Evaluating Architectural Safeguards for Uncertain AI Black-Box Components

Zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

von der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Max Scheerer

aus Kehl

Tag der mündlichen Prüfung: 09. Mai 2023
Erster Gutachter: Prof. Dr. Ralf Reussner
Zweiter Gutachter: Prof. Dr. J. Marius Zöllner

Abstract

There have been enormous achievements in the field of *Artificial Intelligence* (AI) which has attracted a lot of attention. Especially, *Deep Learning* (a subfield of AI) employs so-called *Deep Neural Networks* (DNNs) that have been successfully applied to various complex learning tasks, e.g. autonomous driving or human-robot-interaction. However, the tremendous data dependency and complexity of DNNs revealed significant vulnerabilities. More specifically, DNNs react sensitively to particular environmental factors (e.g. brightness or contrast variations in input images) which can result in incorrect predictions. However, since AI (and especially DNNs) is applied in safety-critical systems, such erroneous behaviour may result in physical or economical damage. As a result, research branches have emerged that approach the unreliable nature of AI.

One of the major issues of AI models is that they reached a high complexity which makes it either impossible to understand their internals or to explain why particular predictions have been made. Thus, they are also referred to as *Black-Boxes*. Existing works address this problem by runtime approaches that can detect either potentially malicious input data or wrong predictions made by the AI model. Although such approaches enable the detection of possible unsafe states, they do not discuss any countermeasures. Consequently, several approaches at the architectural or system level have been elaborated that deal with the unreliable or uncertain nature of AI (e.g. *N-Version Programming Pattern* or *Simplex Architectures*). Moreover, there is a growing requirement for AI-enabled systems to adapt at runtime in order to deal with changing environmental conditions. Systems with such capabilities are known as *Self-Adaptive Systems*. We denote such architectural or system-level approaches (e.g. n-version programming or self-adaptive systems) as *Architectural Safeguards*. Software engineers are now facing the challenge to identify the architectural safeguard that satisfies the non-functional requirements best. Each architectural safeguard, however, impacts the quality attributes of the system differently. It is crucial to resolve such design decisions as early as possible in the development process (i.e. at design-time) to avoid changes after the system has been implemented as they are associated with high costs. In addition, safety-critical systems in particular must satisfy strict (quality) requirements that need to be addressed at the architecture level of the software system.

This thesis presents a model-based approach that supports software engineers in the development of AI-enabled systems. More specifically, the approach allows the evaluation of architectural design decisions specifically dealing with AI-induced uncertainties (i.e. architectural safeguards). In particular, an approach for reliability prediction of AI-enabled systems based on established model-based techniques is presented. In the next step, we describe how the reliability prediction approach is generalised to self-adaptive systems.

The core of the approach is an environment model to describe (*i*) AI-specific uncertainties and (*ii*) the operating environment of a self-adaptive system. Finally, a classification structure or taxonomy is presented which, based on various dimensions, classifies AI-enabled systems into four possible classes. Each class is associated with a certain degree of dependability assurance that can be made for the given system.

The thesis encompasses four central contributions:

1. **Domain-agnostic modelling of AI-specific environments:** In this contribution, a metamodel was elaborated for the modelling of AI-specific uncertainties and their temporal expansion which form the operative environment of a self-adaptive system.
2. **Reliability prediction of AI-enabled systems:** The presented approach extends an existing *Architectural Description Language* (namely the *Palladio Component Model*) for modelling component-based software architectures and an associated reliability prediction tool (for classical software systems). The problem of the black-box property of an AI component is addressed by a sensitivity model that, depending on various uncertainty factors, models the *Predictive Uncertainty* of an AI component.
3. **Evaluation of self-adaptive systems:** This contribution presents a framework for evaluating self-adaptive systems that act as architectural safeguards, i.e. they safeguard an AI component. The concepts presented in this contribution generalises the concepts of contribution 2.
4. **Classes of architectural dependability assurance:** The contribution describes a classification structure that describes the extent to which assurances (w.r.t. a dependability-related system-level property) can be made for a given AI-enabled system.

Contribution 2 was validated in the context of a case study from automated driving. More precisely, we validated whether our reliability prediction approach preserves plausibility assertions that can be observed in the case study. Moreover, we demonstrated the general possibility to evaluate design decisions at design-time. For the validation of contribution 3, plausibility assertions were validated in the context of the aforementioned case study and a case study from the field of human-robot-interaction. In addition, two further community case studies have been considered, in which (based on simulators) quality attributes of self-adaptive systems were evaluated and compared with the results of our framework. In both cases, it could be shown that on the one hand all plausibility assertions are preserved and on the other hand, our approach produces the same results as the domain-specific simulators. Furthermore, we could demonstrate that our approach supports software engineers in evaluating design decisions that are relevant when developing self-adaptive systems. Contribution 1 was implicitly validated with contribution 2 and 3. For the fourth contribution, the classification structure is applied to well-known and representative AI systems. We were able to classify each AI system into one of the classes such that the general applicability of the classification structure was shown.

Zusammenfassung

Künstliche Intelligenz (KI) hat in den vergangenen Jahren große Erfolge erzielt und ist immer stärker in den Fokus geraten. Insbesondere Methoden des *Deep Learning* (ein Teilgebiet der KI), in dem *Tiefe Neuronale Netze* (TNN) zum Einsatz kommen, haben beeindruckende Ergebnisse erzielt, z.B. im autonomen Fahren oder der Mensch-Roboter-Interaktion. Die immense Datenabhängigkeit und Komplexität von TNN haben jedoch gravierende Schwachstellen offenbart. So reagieren TNN sensitiv auf bestimmte Einflussfaktoren der Umwelt (z.B. Helligkeits- oder Kontraständerungen in Bildern) und führen zu falschen Vorhersagen. Da KI (und insbesondere TNN) in sicherheitskritischen Systemen eingesetzt werden, kann solch ein Verhalten zu lebensbedrohlichen Situationen führen. Folglich haben sich neue Forschungspotenziale entwickelt, die sich explizit der Absicherung von KI-Verfahren widmen.

Ein wesentliches Problem bei vielen KI-Verfahren besteht darin, dass ihr Verhalten oder Vorhersagen auf Grund ihrer hohen Komplexität nicht erklärt bzw. nachvollzogen werden können. Solche KI-Modelle werden auch als Black-Box bezeichnet. Bestehende Arbeiten adressieren dieses Problem, in dem zur Laufzeit "bösartige" Eingabedaten identifiziert oder auf Basis von Ein- und Ausgaben potenziell falsche Vorhersagen erkannt werden. Arbeiten in diesem Bereich erlauben es zwar potenziell unsichere Zustände zu erkennen, machen allerdings keine Aussagen, inwiefern mit solchen Situationen umzugehen ist. Somit haben sich eine Reihe von Ansätzen auf Architektur- bzw. Systemebene etabliert, um mit KI-induzierten Unsicherheiten umzugehen (z.B. *N-Version-Programming-Muster* oder *Simplex Architekturen*). Darüber hinaus wächst die Anforderung an KI-basierte Systeme sich zur Laufzeit anzupassen, um mit sich verändernden Bedingungen der Umwelt umgehen zu können. Systeme mit solchen Fähigkeiten sind bekannt als *Selbst-Adaptive Systeme*. Software-Ingenieure stehen nun vor der Herausforderung, aus einer Menge von *Architekturellen Sicherheitsmechanismen*, den Ansatz zu identifizieren, der die nicht-funktionalen Anforderungen bestmöglich erfüllt. Jeder Ansatz hat jedoch unterschiedliche Auswirkungen auf die Qualitätsattribute des Systems. Architekturelle Entwurfsentscheidungen gilt es so früh wie möglich (d.h. zur Entwurfszeit) aufzulösen, um nach der Implementierung des Systems Änderungen zu vermeiden, die mit hohen Kosten verbunden sind. Darüber hinaus müssen insbesondere sicherheitskritische Systeme den strengen (Qualitäts-) Anforderungen gerecht werden, die bereits auf Architektur-Ebene des Software-Systems adressiert werden müssen.

Diese Arbeit befasst sich mit einem modellbasierten Ansatz, der Software-Ingenieure bei der Entwicklung von KI-basierten System unterstützt, um architekturelle Entwurfsentscheidungen (bzw. architekturellen Sicherheitsmechanismen) zum Umgang mit KI-induzierten

Unsicherheiten zu bewerten. Insbesondere wird eine Methode zur Zuverlässigkeitsvorhersage von KI-basierten Systemen auf Basis von etablierten modellbasierten Techniken erforscht. In einem weiteren Schritt wird die Erweiterbarkeit/Verallgemeinerbarkeit der Zuverlässigkeitsvorhersage für Selbst-Adaptive Systeme betrachtet. Der Kern beider Ansätze ist ein Umweltmodell zur Modellierung (i) von KI-spezifischen Unsicherheiten und (ii) der operativen Umwelt des Selbst-Adaptiven Systems. Zuletzt wird eine Klassifikationsstruktur bzw. Taxonomie vorgestellt, welche, auf Basis von verschiedenen Dimensionen, KI-basierte Systeme in unterschiedliche Klassen einteilt. Jede Klasse ist mit einem bestimmten Grad an Verlässlichkeitszusicherungen assoziiert, die für das gegebene System gemacht werden können.

Die Dissertation umfasst vier zentrale Beiträge.

1. **Domänenunabhängige Modellierung von KI-spezifischen Umwelten:** In diesem Beitrag wurde ein Metamodell zur Modellierung von KI-spezifischen Unsicherheiten und ihrer zeitlichen Ausdehnung entwickelt, welche die operative Umgebung eines selbstadaptiven Systems bilden.

2. **Zuverlässigkeitsvorhersage von KI-basierten Systemen:** Der vorgestellte Ansatz erweitert eine existierende Architekturbeschreibungssprache (genauer: *Palladio Component Model*) zur Modellierung von Komponenten-basierten Software-Architekturen sowie einem dazugehörigen Werkzeug zur Zuverlässigkeitsvorhersage (für klassische Software-Systeme). Das Problem der Black-Box-Eigenschaft einer KI-Komponente wird durch ein Sensitivitätsmodell adressiert, das, in Abhängigkeit zu verschiedenen Unsicherheitsfaktoren, die *Prädektive Unsicherheit* einer KI-Komponente modelliert.

3. **Evaluation von Selbst-Adaptiven Systemen:** Dieser Beitrag befasst sich mit einem Rahmenwerk für die Evaluation von Selbst-Adaptiven Systemen, welche für die Absicherung von KI-Komponenten vorgesehen sind. Die Arbeiten zu diesem Beitrag verallgemeinern/erweitern die Konzepte von Beitrag 2 für Selbst-Adaptive Systeme.

4. **Klassen der Verlässlichkeitszusicherungen:** Der Beitrag beschreibt eine Klassifikationsstruktur, die den Grad der Zusicherung (in Bezug auf bestimmte Systemeigenschaften) eines KI-basierten Systems bewertet.

Der zweite Beitrag wurde im Rahmen einer Fallstudie aus dem Bereich des Autonomen Fahrens validiert. Es wurde geprüft, ob Plausibilitätseigenschaften bei der Zuverlässigkeitsvorhersage erhalten bleiben. Hierbei konnte nicht nur die Plausibilität des Ansatzes nachgewiesen werden sondern auch die generelle Möglichkeit Entwurfsentscheidungen zur Entwurfszeit zu bewerten. Für die Validierung des dritten Beitrags wurden ebenfalls Plausibilitätseigenschaften geprüft (im Rahmen der eben genannten Fallstudie und einer Fallstudie aus dem Bereich der Mensch-Roboter-Interaktion). Darüber hinaus wurden zwei weitere Community-Fallstudien betrachtet, bei denen (auf Basis von Simulatoren) Selbst-Adaptive Systeme bewertet und mit den Ergebnissen unseres Ansatzes verglichen wurden. In beiden Fällen konnte gezeigt werden, dass zum einen alle Plausibilitätseigenschaft erhalten werden und zum anderen, der Ansatz die selben Ergebnisse erzeugt, wie die Domänen-spezifischen Simulatoren. Darüber hinaus konnten wir zeigen, dass unser Ansatz Software-Ingenieure bzgl. der Bewertung von Entwurfsentscheidungen, die für die Entwicklung von Selbst-Adaptiven Systemen relevant sind, unterstützt. Der erste Beitrag wurde implizit mit Beitrag 2 und mit 3 validiert. Für den vierten Beitrag wurde die Klassifikationsstruktur auf bekannte und repräsentative KI-Systeme angewandt und diskutiert. Es konnte jedes KI-System in eine der Klassen eingeordnet werden, so dass die generelle Anwendbarkeit der Klassifikationsstruktur gezeigt wurde.

Danksagungen

Im Laufe meiner Dissertation haben mich viele Menschen begleitet und unterstützt. An dieser Stelle möchte ich die Gelegenheit nutzen, um meine Dankbarkeit auszudrücken.

Zu Beginn möchte ich mich bei meinem Doktorvater Prof. Dr. Ralf Reussner bedanken, der mir es nicht nur ermöglicht hat diese Dissertation zu verfassen, sondern mich auch während meiner ganzen Zeit fachlich und vor allem mental unterstützt hat. Insbesondere seine freundliche, offene und empathische Art hat maßgeblich zum Erfolg dieser Dissertation beigetragen. Ebenfalls möchte ich mich bei Herrn Prof. Dr. J. Marius Zöllner für die Übernahme des Korreferats meiner Arbeit bedanken. Sein konstruktives Feedback hat mir dabei sehr geholfen, meine Arbeit aus einer anderen Perspektive zu betrachten.

Des Weiteren möchte ich Axel Busch für die tolle Betreuung meiner Masterarbeit und Emre Taspolatoglu für die Betreuung während meiner Zeit als Hiwi am FZI Forschungszentrum Informatik danken. Beide haben mein Interesse am wissenschaftlichen Arbeiten geweckt und letztlich dazu beigetragen, dass ich die Promotion begonnen habe.

Ein großes Dankeschön geht an meine Kolleginnen und Kollegen am FZI sowie an den Lehrstuhl Software Design and Quality für die gute Zusammenarbeit über die Jahre. Insbesondere möchte ich hierbei vom FZI Jonas Klamroth, Martina Rapp und Sebastian Weber danken, die mich durch Ihre zahlreichen Anmerkungen zu dieser Arbeit unterstützt haben. Ebenfalls möchte ich mich bei den von mir betreuten Student Dennis Bäuml bedanken für seine Unterstützung bei verschiedenen Implementierungsarbeiten. Ich möchte auch meinem Abteilungsleiter Jörg Henß und meinem Bereichsleiter Oliver Denninger meinen Dank aussprechen. Sie haben es mir ermöglicht, während meiner Arbeitszeit am FZI Raum für meine Dissertation zu schaffen.

Mein größter Dank gilt meiner Familie. Meinen Eltern Sybille und Heinz danke ich für ihre unermessliche Unterstützung und Liebe. Schließlich möchte ich meiner Ehefrau Michaela danken, die mir in meinem Leben (und insbesondere im Laufe der Dissertation) so viel Kraft, Liebe und Stabilität gibt. Ohne euch wäre ich niemals an dem Punkt in meinem Leben angelangt, an dem ich jetzt stehe; ich bin unendlich dankbar, euch in meinem Leben zu haben.

Contents

List of Figures

List of Tables

List of Listings

Notations

This section gives an overview of the most common notations used in this thesis. In some places, we deviate from the notations to introduce context-specific notations.

Set Theory

A or $\mathcal{A}$	Capitalised letters denote a set of elements
$\{a, b, c\}$	A set consisting of the elements a,b and c
$\|X\|$	The cardinality of set X, e.g. $\|\{a, b, c\}\| = 3$
$\mathbb{N}$	The set of natural numbers
$\mathbb{Z}$	The set of natural numbers
$\mathbb{R}$	The set of real numbers
$[a, b]$	The real interval: $\{x \in \mathbb{R} \mid a \leq x \leq b\}$
$\{x \mid \Phi(x)\}$	Set-builder notation: the set of all values of x that satisfy predicate $\Phi(x)$
$(x_1, ..., x_n)$	A tuple consisting of n elements
$\sim$	Equivalence relation
$[x]_\sim$	Equivalence class of $x \in X$, i.e. $\{y \in X \mid y \sim x\}$
$X/\sim$	Quotient set, i.e. $\{[x]_\sim \mid x \in X\}$

Model & Graph Theory

$\mathcal{G}$	A graph
M	Capitalised m denotes a model
M_X	Model-based representation of the set X
$\mathcal{MM}(M)$	Defines the metamodel of model M

Probability Theory

X	A random variable (always written with a capital x)
$\mathbf{X} := \{X_1, ..., X_n\}$	Bold X defines a set of random variables
X_A	Relates a set A with a random variable such that $Val(X_A) = Val(A)$
$X_A \perp\!\!\!\perp X_B$	The random variables X_A and X_B are independent
$(X_A \perp\!\!\!\perp X_B \mid X_C)$	The random variables X_A and X_B are conditionally independent given X_C
$Pr(X_A = a)$	The probability of event a
$Pr(X_A = a \mid X_B = b)$	The conditional probability of event a given b
$P(X_A)$	A probability distribution over a discrete random variable X_A
$P(X_A \mid X_B)$	A conditional probability distribution over the discrete random variables X_A and X_B
$\mathbb{E}[X]$	The expected value of a random variable X
$(X_t)_{t \in \mathbb{N}}$	A stochastic process for a family of random variables
$x \sim P(X)$	Indicates that value x is generated or sampled from probability distribution P defined over random variable X
$P \models \mathcal{G}$	Indicates that distribution P satisfies the independence assumptions encoded by graph $\mathcal{G}$

Miscellaneous

$\mathbb{1}_c$	The indicator function evaluates to 1 if condition c is true, 0 otherwise
$Val(\cdot)$	Returns the value space of a function or set
$\underset{x \in X}{max} f(x)$	Returns the value of $f(x)$ maximizing function f
$\underset{x \in X}{\text{argmax}} f(x)$	Returns the argument $x \in X$ maximizing $f(x)$
$\exists_{=1}$	Restricts the existential quantifier $\exists$ to exactly one element
$max(a, b)$	Returns a if $a \geq b$ and b otherwise.

$(f \circ g)(x)$	Function composition, i.e. $(f \circ g)(x) = f(g(x))$

Part I.

Prologue

1. Introduction

This thesis presents an approach to evaluate *Architectural Safeguards* for dealing with uncertainty induced by *Artificial Intelligence* (AI). With AI-induced uncertainty, we primarily mean *Predictive Uncertainty*, i.e. the confidence associated with a prediction of an AI component itself [89]. Moreover, we consider architectural or system-level approaches (e.g. architectural patterns or self-adaptive systems) that aim to reduce the predictive uncertainty of AI components as architectural safeguards. Our approach supports software engineers in the decision-making process of selecting appropriate architectural safeguards for AI-enabled systems w.r.t. non-functional requirements (e.g. reliability and performance). AI-enabled systems refer to software systems that, in addition to classic software components, include AI components that have been trained for specific tasks (e.g. object detection). In this chapter, we motivate the importance of safeguarding AI components and the capability of assessing and comparing architectural safeguards in early development stages. We provide a broad overview of the current state of research in this area before discussing the research gap we have identified. Furthermore, we enumerate several challenges one must take into consideration to close this gap and formulate the research questions of this thesis accordingly. On this basis, we provide an overview of our contributions. Before presenting the outline of this work, we discuss some example systems that we use as running examples in this thesis.

1.1. Motivation

In the past years, AI made tremendous progress. Especially, methods from the field of *Machine Learning/Deep Learning* (a subfield of AI) gained much attention and opened a multitude of application scenarios. Deep learning methods employ so-called *Deep Neural Networks* (DNNs) that learn various concepts or tasks based on training examples. For example, there is successful work on the application of AI in autonomous driving (e.g. [43]), robot manipulation (e.g. [74]), or smart manufacturing (e.g. [196]).

AI (and ML in particular) is often used to learn complex behaviours or to approximate functions that are difficult to program explicitly using large datasets of training examples. Depending on the learning scenario, the function to be approximated becomes very complex (e.g. in self-driving cars or robot manipulation scenarios). Consequently, the capacity of the learned AI models goes hand in hand with increasing complexity. However, this has serious consequences and comes at the expense of the interpretability and transparency of the AI models. As a result, the internal behaviour of an AI model can neither be understood nor can it be explained why certain predictions were made. For this reason, AI models with high complexity are also considered *Black-Boxes* [76].

Moreover, the increased complexity of AI models demands more training data to learn specific concepts accurately. In practice, it is quite difficult to obtain large datasets that are of high quality, e.g. including representative examples (i.e. cover sufficient training examples that represent the concepts to be learned). However, the quality of datasets is crucial because it strongly correlates with the accuracy of the trained AI models. Consequently, the inherent dependency of AI models on training data, coupled with their high complexity and black-box nature, can potentially lead to undesirable behaviours. For instance, so-called adversarial examples exploit small variations of the input space along the decision boundary of an AI model to force misclassification, while appearing unmodified to human observers [135, 71]. According to Hanif et al. [79] small errors of individual weights (simulated by injected bit flips) in the first layer of a DNN can result in unacceptable accuracy loss. Moreover, it has also been observed that AI models make wrong predictions for certain input data but are still fairly confident in their prediction [89]. Tian et al. [186] have shown that artificially inserting different environmental conditions (such as changes in brightness or harsh weather conditions) into images that were correctly classified before may drastically change the prediction result.

The unreliable nature of AI components is of particular concern in safety-critical applications. Tian et al. [186] enumerate three reported incidents in automated driving that resulted in crashes due to incorrect predictions of DNNs. One of the incidents even ended fatally because the AI component used was unable to recognise a white truck in a scene with bright contrast [210]. A promising approach is therefore to verify or certify AI components by applying formal verification approaches. However, providing formal guarantees for AI components is challenging and hardly scales for complex AI models [99]. Although various approaches aim to improve properties such as safety (e.g. [195]) or robustness (e.g. [99]) of AI models, it is arguably challenging to fully verify AI models due to their inherently probabilistic and nonlinear nature.

To deal with AI-related uncertainties, various approaches have been developed in the past years. For example, [99, 100, 172] represent approaches for verifying properties in DNNs (e.g. robustness properties) to deal with adversarial examples. [76] describes a class of approaches that generate explanations for predictions made by an AI component. The explanations, for example, can be checked against some formal constraints at runtime and suitable countermeasures can be taken in case of violation. In [1, 81] out-of-distribution approaches are presented that determine whether a new input data (observed at runtime) is likely to be generated by the same probability distribution as the training data. If this is not the case, the input potentially leads to wrong predictions. Similarly, Cheng et al. [46] introduce an approach to monitor neuron activation patterns of neural networks at runtime. Activation patterns for newly arriving input data are generated. The patterns are compared to the ones generated from the training data (by an upstream pattern generation process). Thus, input for which no similarity exists is considered to be potentially malicious. What all these approaches have in common is that they provide means to detect potentially unsafe states of AI components but do not define what to do in case of detection, i.e. no countermeasures are discussed. Additionally, in the context of ML testing (see Zhang et al. [213] which provides a comprehensive overview of machine learning testing approaches) several methods are discussed that try to identify corner cases for which ML models tend to produce wrong predictions. However, although ML testing methods provide insights to specifically retrain the ML model, they still do not guarantee that the model operates reliably.

Therefore, new research directions emerged in the field of software engineering that focused on applying or adopting established methods to deal with AI-induced uncertainties at the architecture or system level. Work in this line of research does not only describe countermeasures or architectural safeguards in general but also approaches that incorporate methods outlined before (e.g. neuron activation patterns or out-of-distribution approaches to detect unsafe states). For instance, Shafaei et al. [167] outline an architectural pattern that relies on an input checker component to detect potentially malicious inputs. The input checker can be implemented according to one of the approaches described above. Moreover, [77, 211, 119] adopt the *N-Version Programming* pattern to reduce predictive uncertainty, i.e. *N* distinct AI models are used, and a more reliable output is generated by combining the *N* results (e.g. majority voting). Another example is provided by Musau et al. [131], that adopt *Simplex Architectures* to deal with unreliable AI models. In AI-enabled systems, however, not only the requirement to deal with AI-related uncertainties at the architectural level emerged but also the requirement of self-adaptation [114, 216, 11, 128, 41]. More specifically, so-called *Self-Adaptive Systems* provide an established software engineering approach that is primarily concerned with maintaining quality objectives (or non-functional requirements such as performance or reliability) [153]. Self-adaptive systems generalise traditional static software systems in that they allow the structure or behaviour of a system to be adapted at runtime in response to environmental conditions that the current system configuration cannot cope with. This makes them predestined safeguarding mechanisms for AI components, as they can intervene in safety-critical states at runtime, e.g. by transitioning the system into a fail-safe mode or switching the AI component with a more conservative but functionally equivalent component.

When engineering AI-enabled systems, however, software engineers are facing various design options related to architectural safeguards, which are preferably resolved at design time, i.e. before the system is implemented. In concrete terms, different design decisions have to be compared and evaluated in the decision-making process. For example, should one use a self-adaptive system as an architectural safeguard or is there an architectural pattern that meets the requirements just as well? Although self-adaptive systems are quite powerful in terms of their adaptive capabilities, they are also more complex compared to static software systems because they are associated with more uncertainties [57] and their temporal evolution corresponds to stochastic processes (see e.g. [126, 40]). Therefore, software engineers are better advised with static architectural safeguards. At design-time, however, it is hard to assess whether for a given application context a self-adaptive system or a non-adaptive solution is more advisable. Assuming the software engineers have chosen a static solution, there are still numerous decisions and options to consider. For example, shall one use an n-version programming pattern or rather consider a simplex architecture? Both solutions affect the quality attributes of the system differently. The n-version programming pattern arguably improves the reliability of the system but degrades performance attributes at the same time. In some application contexts, the performance loss might be not acceptable such that n-version programming is not applicable (e.g. autonomous driving). For software engineers, however, this is hard to assess at design-time because quality attributes are usually only observable at runtime. The same applies to self-adaptive systems in which various design decisions exist, each of which effecting the quality attributes differently.

The lack of tools to evaluate design decisions of software architectures is not a new problem in software engineering and has already been addressed in some domains by using techniques of *Model-driven Software Development* (MDSD). In performance engineering, for example, Reussner et al. [149] describe a model-based approach in which component-based software architectures are modelled and simulated to predict performance attributes (e.g. response time). Based on the predictions, design decisions can be evaluated and compared at design-time. The goal of this thesis follows the same underlying concern, i.e. to support software engineers in the decision-making process by evaluating and comparing architectural safeguards w.r.t. their impact on system-level quality attributes. Moreover, we provide a model-based reliability prediction approach for AI-enabled software systems which accounts for the predictive uncertainty of an AI component. Our approach can be complemented by existing prediction approaches (e.g. [149]) to assess architectural safeguards under the perspective of multiple quality attributes. Additionally, we generalise the approach to self-adaptive systems. More precisely, software engineers can evaluate (i) whether self-adaptation is necessary at all (as opposed to a static solution), (ii) design decisions within an adaptation strategy family and (iii) distinct adaptation strategies. Complementing our tool support, we present a classification structure that allows software engineers to assess the level of assurance that can be assigned to an AI-enabled system w.r.t. a specific system-level property. The assessment takes place prior to the design and development of the system and not only provides intuition about the engineering and domain problem itself but also indicates whether systems can be realistically engineered given the system-level property under consideration.

1.2. Research Gaps

We have identified four research gaps that are not addressed at all or not sufficiently by the current state of research. With this thesis, we contribute to the current state of research by addressing the following research gaps:

Analysing the impact of architectural safeguards on reliability attributes considering the predictive uncertainty of AI components: Architectural safeguards have various manifestations, e.g. n-version programming pattern, simplex architecture, filtering methods, self-adaptive systems, etc. What they have in common is that they aim to reduce the predictive uncertainty of an AI component. Predictive uncertainty in turn is affected by other uncertainty factors or environmental variables, e.g. brightness variations in images (or disturbances in input data in general), lack of knowledge of the AI model (due to insufficient training data), etc. Depending on the application context, some architectural safeguards are better suited than others (e.g. filtering approaches to reduce input disturbances or n-version programming to compensate for the lack of knowledge of an AI model). At design-time, however, it is hard to determine which safeguard is better suited in a given context. In the current research, there is currently no approach that connects the impact of architectural safeguards to predictive uncertainty. Thus, it is not even hard to determine how an architectural safeguard acts on predictive uncertainty but also how it impacts the overall reliability of the AI-enabled system, i.e. a system which includes an AI component with high predictive uncertainty is arguably operating less reliably.

Decision support regarding the use of self-adaptive or static software systems: Complementary to the first research gap, the question arises if a self-adaptive system or, say, an architectural pattern (and thus a non-adaptive solution) should be used as an architectural safeguard. The great advantage of self-adaptive systems is (in case of quality objective violation) to dynamically change the system configurations whenever the system cannot deal with the current state of the operating environment (harsh weather conditions, sensor noise, hardware failure and so forth). On the other hand, self-adaptive systems are more complex compared to static software systems (due to the temporal aspect). Therefore, software engineers tend to prefer static software systems. If software engineers opt for a static system solution, they implicitly assume that there must be a single system configuration (or software architecture) for the given operating environment that meets all non-functional (or quality) requirements. At design-time, this is difficult to assess, and the assumption needs to be well justified. For example, if one decides to implement a self-adaptive system, it must be convincingly demonstrated that there is no static solution that performs comparably and whether the self-adaptive system performs significantly better (in terms of quality objectives) such that the higher effort is justified. However, such decisions are crucial and, if taken incorrectly, can lead to a massive waste of human resources or costs. Moreover, it seems fairly challenging to compare the two types of systems which are inherently different. To the best of our knowledge, no work supports software engineers in making such well-informed design decisions which in turn is of great importance when engineering software systems.

Evaluation of MAPE-K-based adaptation strategies at design-time: In addition to the second research gap, there remains the challenge to evaluate the quality of adaptation strategies when software engineers have decided to deploy self-adaptive systems as an architectural safeguard. In this thesis, we focus on so-called *MAPE-K*-based self-adaptive systems. In section 2.1 we discuss the general framework of MAPE-K feedback loops; for the moment, however, it is sufficient to know that MAPE-K-based self-adaptive systems decompose the adaptation process into the phases *Monitor*, *Analyse*, *Plan* and *Execute*, which are continuously run through. After each cycle either an adaptation is made or not. Although there are approaches that allow for design-time evaluation of self-adaptive systems, they tend to focus on evaluating specific scenarios for a given domain (e.g. [15]), the impact of adaptations itself (e.g. [38]), synthesise adaptation strategy repertoires at design-time (e.g. [39]) or do not aim to evaluate MAPE-K-based adaptation strategies (e.g. [20]). However, there are no approaches that evaluate adaptation strategies more comprehensively. More specifically, the dynamics of self-adaptive systems induce a multitude of different state sequences, each of which represents a particular trajectory of how a self-adaptive system moves through the space w.r.t. the operating environment and the adaptations made by the strategy. Analysing the distinct trajectories, however, is crucial to determine the quality of the adaptation logic which is implemented by the adaptation strategy.

Moreover, according to Esfahani and Malek [57], self-adaptive systems must account for the uncertainty *Parameter over time* which relates to the uncertainty of adaptation impact in the long-term. To evaluate the quality of an adaptation strategy, it is crucial to do so in terms of *Parameter over time* because strategies that select adaptations that seem to work well in a given state may perform poorly in the long run.

Cross-domain analysis of architectural safeguards: In practice, AI models are applied in various domains. The used AI models or components vary in their predictive uncertainty (e.g. due to the amount and quality of available training data) in each domain. Moreover, also the environmental variables (which affect predictive uncertainty) are different, e.g. in some domains harsh weather conditions are jeopardizing the prediction accuracy; other domains may indicate various levels of sensor noise. From an analytical point of view, the cross-domain application of AI is challenging because the environmental variables of each domain are different. Therefore, a domain-independent approach is required to analyse architectural safeguards independent of the application context and to causally relate domain-specific environmental variables to the predictive uncertainty of an AI component. With such a modelling approach, it is possible to analyse how an architectural safeguard impacts predictive uncertainty. Furthermore, as we consider self-adaptive systems as architectural safeguards, we also need to consider a temporal component. That is, the modelling approach must not only take into account the causal relationships of the environmental variables and predictive uncertainty but also how the variables evolve. Modelling and describing the temporal evolution of the environmental variables is crucial to analyse how the adaptation strategy responds to different environmental states. To the best of our knowledge, there exists no modelling approach that allows for the description of environmental variables, their impact on predictive uncertainty and their temporal expansion. Although several approaches in the literature allow modelling the operating environment of self-adaptive systems in general, they are either applicable to a specific domain or not suitable for modelling environmental state spaces flexibly and compactly. Addressing this research gap must be seen as a prerequisite for addressing the research gaps mentioned above.

1.3. Challenges and Research Questions

To support software engineers in evaluating distinct architectural safeguards at design-time and to address the research gaps outlined in section 1.2, we identified three challenges. In the following, we discuss the challenges and the related research questions.

1.3.1. Modelling and Simulating Adaptation Strategies of Self-Adaptive Systems

As we already pointed out in section 1.2, there are currently no approaches that evaluate adaptation strategies of MAPE-K-based self-adaptive system sufficiently, i.e. in terms of multiple quality objectives and the uncertainty *Parameter over time*. Note, however, that we aim to evaluate three aspects when considering self-adaptive systems as architectural safeguards, namely (i) design decisions within adaptation strategies, (ii) the quality of distinct adaptation strategies for comparison, and (iii) the potential advantage/disadvantage of using self-adaptive systems as opposed to static architectural safeguards. Therefore, the first challenge that emerges relates to the modelling and simulation of self-adaptive systems w.r.t. several quality objectives. We formulate the first main research question as follows:

Research Question 1: How to evaluate adaptation strategies of self-adaptive systems at design-time regarding the ability to meet quality objectives?

To answer research question **RQ1**, several sub-questions need to be considered which, if answered individually, will allow **RQ1** to be answered.

Self-adaptive systems operate in dynamic environments that transition into states the system is not able to deal with or for which the current system configuration violates the quality objectives (e.g. the response time of the system exceeds a given threshold). Thus, it is essential to model the operating environment to simulate and evaluate adaptation strategies. Moreover, as we discussed in section 1.2, there is a requirement for cross-domain analysis. More specifically, the environments of self-adaptive systems differ depending on the application context, e.g. in some applications hardware failures and the number of user requests determine the main variables of the environment, while others are characterised by sensor noise or harsh weather conditions. Consequently, the dynamics of the operating environment must be modelled domain-independently:

Research Question 1.1: How can environmental dynamics be formalised domain-independently at design-time?

A second issue refers to the potential complexity of such environments. In addition to the temporal aspect that must be described, environments may also exhibit large state spaces that must be modelled flexibly and compactly:

Research Question 1.2: What is an appropriate level of abstraction to represent the environmental dynamics domain independently? By appropriateness, we mean that

- adaptation strategies can be analysed at design-time with sufficient accuracy.
- environmental state spaces can be described flexibly and compactly.

Finally, the question of an appropriate analytical model arises which serves as a foundation to evaluate adaptation strategies. For example, Reussner et al. [149] transform architecture models to *Queuing Networks* to predict performance attributes. Finding proper analytical models for self-adaptive systems, however, is especially challenging because the analytical model must capture the adaptation process (i.e. the various changes of the system configuration), account for the uncertainty *Parameter over time* and the inclusion of multiple quality objectives as basis of quality assessment:

Research Question 1.3: What is an appropriate analytical model to enable design-time analyses of self-adaptive systems?

Research Question 1.4: Are the predictions sufficiently accurate to yield plausible results?

Note that we have deliberately formulated research question **RQ1** (and its associated sub-questions) in general terms. More precisely, we have formulated the questions independently of the actual intention to evaluate self-adaptive systems as architectural safeguards. Suppose there is an approach that allows the evaluation of self-adaptive systems as architectural safeguards, how far is this approach from being a generic method for evaluating any type of self-adaptive system (or its strategy) regardless of its actual intended use? Therefore, answering the research questions not only provides the foundation for evaluating self-adaptive systems as architectural safeguards but also examines how such an approach can be generalised.

1.3.2. Evaluation of Architectural Safeguards Regarding Reliability Attributes

The next central challenge relates to the evaluation of architectural safeguards. Because we focus in this thesis on the evaluation of reliability attributes, we aim to predict reliability for AI-enabled systems. We formulate the corresponding research question as follows:

Research Question 2: How can software systems that contain AI black-box components be evaluated in terms of meeting reliability attributes at design-time?

There already exist approaches that enable reliability prediction of modelled software systems at design-time, e.g. Brosch [33]. However, there is no approach taking into account systems with uncertain AI black-box components and their predictive uncertainties. The systematic consideration of AI components in reliability prediction is associated with two sub-challenges.

The first challenge refers to the black-box property of AI components themselves. Not knowing the true state of an AI component is not only an issue at runtime but also at design-time because without being able to determine whether the AI component exhibits erroneous behaviour, we cannot draw any conclusions about predictive uncertainty. In the further course of this work, we refer to the problem of not being able to determine the true state of an AI component (induced by the black-box property) as the *Hidden State Problem*. We consider the following sub-question:

Research Question 2.1: How to deal with the hidden state problem of AI black-box components?

The second challenge refers to the systematic consideration of environmental variables that are correlated with the predictive uncertainty of an AI component in the reliability prediction process. Seshia et al. [165] enumerate several challenges to achieving formally-verified AI-enabled systems. Hereby, the authors pointed out that the modelling of the AI system and its operating environment is challenging due to potentially large input spaces AI components may encounter (e.g. the pixel space). However, this applies not only to verification but to any model-based analysis, since both concepts must be taken into account to some extent in reliability prediction. We tackle this problem by abstracting away the irrelevant details and focusing only on the variables in the environment which are causally related to the predictive uncertainty of an AI component. Eventually, these correlations must be systematically integrated into the reliability prediction. We consider the following sub-questions:

Research Question 2.2: How to systematically consider the influence of predictive uncertainty and causally related environmental variables in the reliability prediction?

Finally, the last research question of this section is about how to evaluate self-adaptive systems for safeguarding uncertain AI components at design-time and generalises the reliability prediction approach of **RQ2** to self-adaptive systems:

Research Question 3: How can adaptation strategies of self-adaptive systems that safeguard uncertain AI black-box components be evaluated in terms of reliability at design-time?

In principle, however, the research question is mainly concerned with how the insights gained from **RQ2** (and its sub-questions) can be combined with the results of **RQ1** from section 1.3.1.

1.3.3. Dependability Assurance of AI-enabled Systems

Our research focus is on the evaluation of architectural safeguards. In doing so, we consider reliability (more precisely, the success probability of the system) as system-level property. Nonetheless, there are other system properties of AI-enabled systems for which software engineers would like to give assurances. More generally, software dependability encompasses a wide range of system-level properties. According to Sommerville [173], dependability encompasses four principle dimensions, namely *Safety*, *Reliability*, *Security* and *Availability*. However, is it always possible to give assurances for any system property? And at which level can they be given? Clearly, it is desirable to provide assurances at design-time so that we can ensure that the modelled software architecture of the AI-enabled system satisfies the system property. However, this seems difficult to achieve in practice, as there might be various possible system-level properties which are more or less difficult to assure. Instead, it seems to make more sense that some properties can be assured at design-time, while others can only be assured at runtime and still others cannot be assured at all.

The last challenge and research focus of this thesis address exactly the previous discussion. More specifically, the idea is to assess an AI-enabled system, its operating environment, used AI component and other factors to reason about the extent of assurances that can be given regarding a particular dependability-related system-level property:

Research Question 4: How to assess the extent to which dependability assurances can be given for an AI-enabled system?

Before that, however, there must be classes into which an AI-enabled system can be classified. We call these *Classes of Architectural Dependability Assurance* because each of which determines whether assurances can be given at design-time, runtime or not at all w.r.t. a system-level dependability property:

Research Question 4.1: What are appropriate classes of architectural dependability assurances?

There are various possible factors one can take into account to assess an AI-enabled system. However, these factors may also vary depending on the considered application context. Therefore, appropriate classification dimensions need to be identified which are sufficiently generic to classify a broad class of AI systems but also concise enough to allow classification into one of the classes of architectural dependability assurance:

Research Question 4.2: What are the suitable dimensions for classification?

In summary, the result of **RQ4** is a classification structure which classifies AI-enabled systems w.r.t. a system-level property. The classification is conducted by considering the dimensions of **RQ4.2** based on which an AI system is assigned to a particular class of architectural dependability assurance from **RQ4.1**.

1.4. Contributions

The contribution of this thesis results directly from the identified research gaps. We make four contributions to the current state of research:

Contribution 1: Domain-agnostic instantiation of probabilistic environment models. Recall that AI is applicable in many contexts or use cases. Consequently, our approaches must be applicable regardless of the domain. To describe the architecture of a software system, we use the *Palladio Component Model* (PCM) as modelling language from Reussner et al. [149]. PCM is a very powerful and expressive formal language for component-based software architectures which allows modelling a broad class of cross-domain systems and which we consider sufficient for our purposes. When evaluating architectural safeguards, the most crucial aspect that makes the application domain-specific is, on the one hand, the environmental variables that affect the predictive uncertainty of an AI component and, on the other hand, the operating environment if we consider a self-adaptive system as an architectural safeguard. In terms of self-adaptive systems, however, the operating environment is strongly connected to the environmental variables that affect the predictive uncertainty. In effect, they describe the same concept with the difference that their temporal expansion must be added to fully describe the operating environment.

To model and instantiate environmental variables (that form the operating environment) in a domain-agnostic way, we provide a metamodel of a formal modelling language. The modelling language unifies the requirements of describing environmental variables and operating environments of a self-adaptive system into a single environment model which can be instantiated in any domain. We consider a probabilistic relationship between the environmental variables and the predictive uncertainty of an AI component. We employ *Probabilistic Graphical Models* (see section 2.6) to describe the relationship by a network of connected discrete random variables (representing the environmental variables and predictive uncertainty) and a set of multinomial probability distributions. Again, we reuse probabilistic graphical models to describe how the variables probabilistically evolve. The temporal expansion allows the modelling of the *Environmental Dynamics* a system encounters and represents what we consider to be the operating environment of a self-adaptive system. Finally, to account for the domain-independent application, the formal semantics of our modelling language build upon the formal semantics of *Template-based Probabilistic Models* (see section 2.6.3) which describe a general framework to instantiate probabilistic structures domain-independently.

Contribution 2: Reliability prediction of AI-enabled systems at design-time. Our next contribution refers to the design-time support for software engineers to make well-informed design decisions when choosing between multiple architectural safeguards. Therefore, we introduce an approach which allows predicting reliability attributes of an AI-enabled system such that multiple variants of the system (each including a different architectural design choice) can be evaluated and compared. In this contribution, we focus on static software systems but describe in the next contribution how the approach is generalised to self-adaptive systems.

Predicting quality attributes at design-time is, in fact, not new in research. There are various approaches (e.g. [149, 15, 33, 106]) that make use of model-based techniques to abstract the software system and to predict quality attributes such as performance, reliability or costs. Therefore, we build upon these concepts and work that exists so far in research. More specifically, our reliability prediction approach extends an existing approach from Brosch [33] for predicting reliability attributes of software systems not including AI components. The approach of Brosch builds upon PCM, i.e. the formal language that we mentioned in the first contribution to model component-based software architectures. In our extension, we represent an AI component by a sensitivity model which is obtained by an upstream sensitivity analysis. The environment model of our first contribution provides the required means to model such sensitivity models. Based on the sensitivity model, the reliability of the system is evaluated by considering different manifestations of the environmental variables (which affect predictive uncertainty).

Contribution 3: Evaluation of adaptation strategies of self-adaptive systems at design-time. The third contribution of this thesis is twofold: The first part of the contribution generalises the concepts from the second contribution to self-adaptive systems; the second part relates to the general ability to evaluate adaptation strategies for any type of self-adaptive systems (not only considering AI safeguards). The foundation forms the environment model from the first contribution that allows modelling the stochastic dynamics of the operating environment. We consider a self-adaptive system as an *Agent* responding to changes in the environment. More specifically, we define a self-adaptive system as *Markov Decision Process* (MDP) which is a prevalent theoretical concept to view self-adaptive systems more formally (e.g. [126, 40, 55]). In simple terms, MDPs are stochastic processes satisfying the *Markov Assumption* (see section 2.4). The mathematical framework of MDPs provides the necessary building blocks for evaluating adaptation strategies. More precisely, we employ *Dynamic Programming* and *Monte Carlo Methods* (which builds upon MDPs) to evaluate adaptation strategies.

In the first part of the contribution, we investigate whether there is a general approach for evaluating adaptation strategies of any type of self-adaptive system in any domain. The second part of the contribution integrates the concepts of our reliability prediction approach of the second contribution to evaluate adaptation strategies that are specifically engineered to safeguard AI components. In doing so, we embed the reliability prediction into the reward function (which resembles a utility function for evaluating decisions made by the adaptation strategy) of the MDP to account for reliability attributes in the overall assessment of a strategy.

Contribution 4: Classification structure to assess AI-enabled systems regarding assurances that can be given for system-level dependability properties. The last contribution of this thesis relates to the classes of architectural dependability assurance and its corresponding classification structure. The contribution supplements the other contributions by assessing AI-enabled systems (that are to be engineered) before the design and implementation w.r.t. a particular system-level property. The classification structure not only provides an initial intuition about the problem domain and the assurances that can be given but also guides software engineers in system design and during the assurance process.

We elaborated four classes of architectural dependability assurance into which a system can be classified. Moreover, we identified several classification dimensions to classify (self-adaptive and static) AI-enabled systems into one of the classes. However, it should be mentioned here that the classes are highly subjective. Therefore, we could not fully evaluate their appropriateness. Considering the amount of work that has been done in the other contributions, a comprehensive evaluation of the classes was not possible. Therefore, a more comprehensive evaluation is planned for future work.

1.5. Example Systems

In this thesis, we illustrate complex and theoretical concepts with examples to make them more amenable. Therefore, this section presents the example systems that we refer to repeatedly in this work. We consider several example systems because a single example system is not sufficient to support all concepts with illustrations. Moreover, some example systems are classical software systems (i.e. without AI components), which might seem somewhat contradictory concerning our planned contributions. However, some parts of the first and third contributions are not limited to AI-enabled systems (in particular, in the third contribution we investigate how the approach of evaluating adaptation strategies can be generalised to arbitrary application contexts). Therefore, we also consider software systems without AI components, which are more suitable for other concepts of our approach.

1.5.1. Load Balancer

As the first running example, we consider the Znn.com system [48] which is a prevalent exemplar in the self-adaptive system research community. As shown in Figure 1.1 the system comprises three components, namely a load balancer component deployed on a web server node and two application server components deployed on two application server nodes. In principle, the system is exposed to varying amounts of user requests. Consequently, the system might experience overload scenarios in which the number of user requests cannot be handled by the system, i.e. the performance of the system (measured by the response time) degrades. However, the load balancer component controls the distribution factor (i.e. the factor which is responsible for distributing the incoming load on the available servers; see $\alpha \in [0, 1]$ in Figure 1.1) which can be adjusted by a self-adaptive system. Besides overload scenarios, hardware failures can arise, e.g. a server node is temporarily not available.

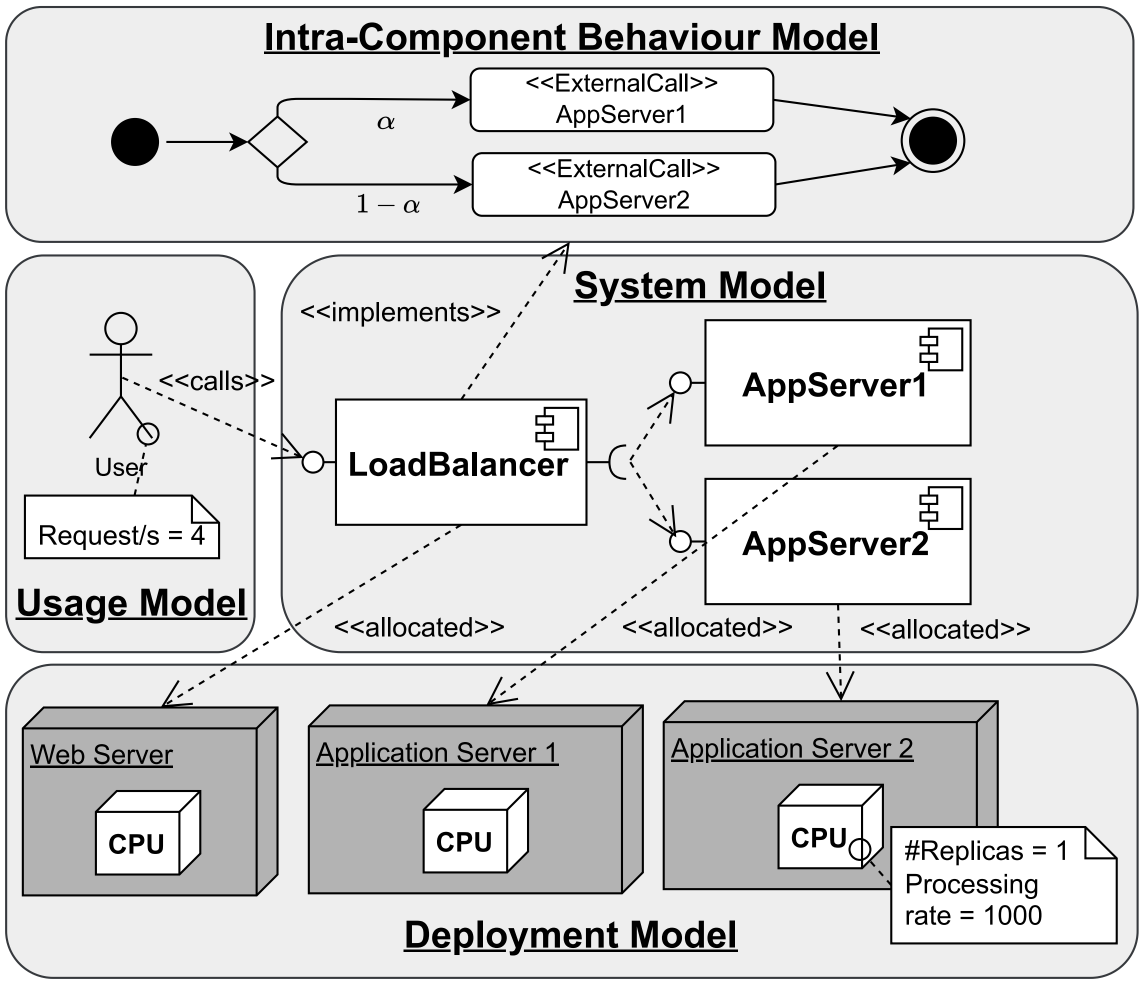


Figure 1.1.: PCM instance of the load balancer system in a UML-based notation (taken from [158]).

In the case of the Znn.com system, we consider a self-adaptive system which adapts the system configuration by varying the distribution factor of the load balancer component. That is, the self-adaptive system can distribute the incoming load evenly in terms of high-load scenarios or to a single node if the other server node is not available. More specifically, the adaptation problem is to keep the system responsive in the presence of high user loads and potential hardware failures while minimising the number of utilised servers over time.

1.5.2. DeltaIoT

The DeltaIoT system [92, 168] is a widely used case study in the self-adaptive system community and originates from the *Internet of Things* (IoT) domain. The DeltaIoT system is a multi-hop network consisting of 15 motes. The system is deployed at the Department of Computer Science at KU Leuven in Belgium. The motes are distributed in various buildings and communicate with each other based on *LoRa* communication, a low-power

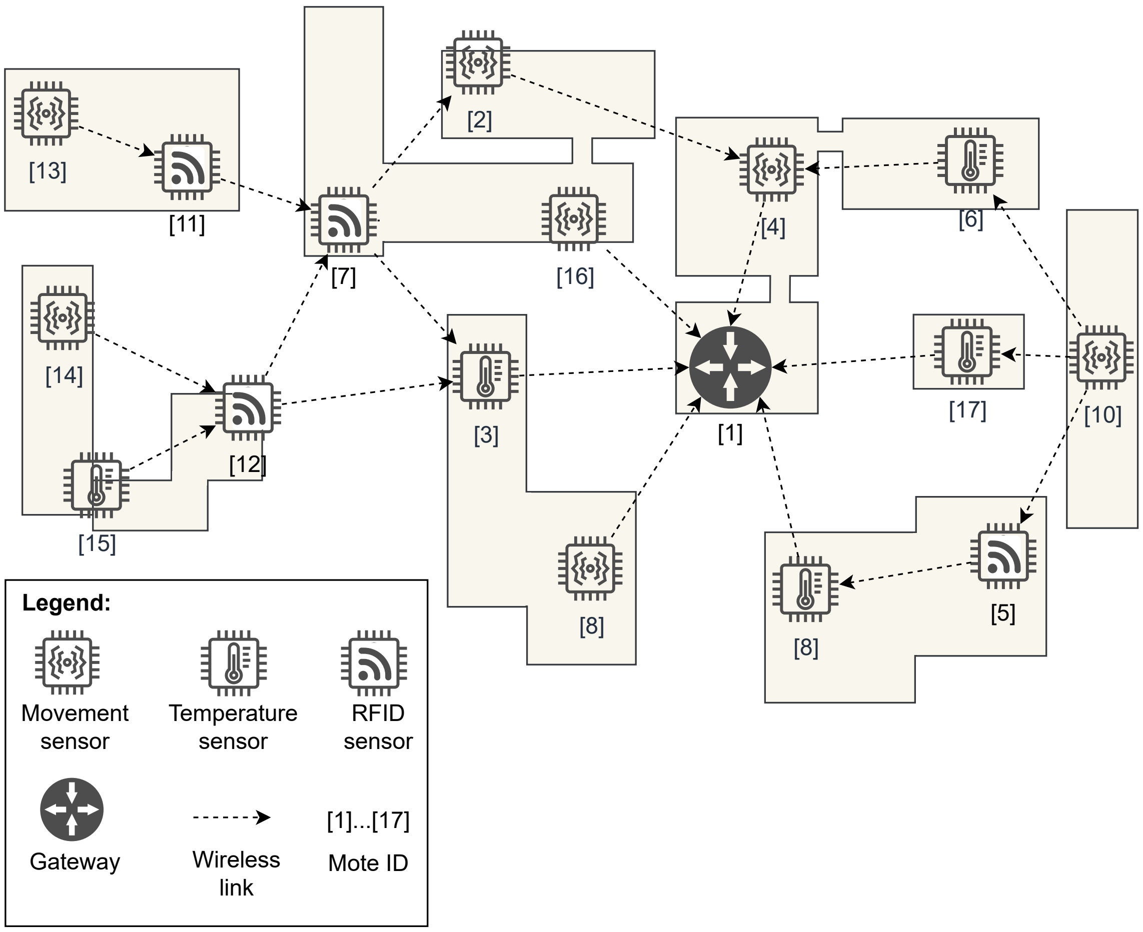


Figure 1.2.: Network topology of the DeltaIoT system (roughly sketched from [168]).

wide area network modulation technique suitable for low-power devices such as those found in IoT networks. Each mote is equipped with a single sensor where three sensor types are considered: RFID sensor to provide access control to the labs, passive infrared sensor to monitor the occupancy status of buildings and temperature sensor to sense the temperature. The emitted sensor data of the motes are transmitted to a central gateway where the data is aggregated and made available. Thus, actions can be taken by the Campus security personnel in case of unusual behaviour. The DeltaIoT system is depicted on Figure 1.2.

As shown in Figure 1.2, some motes cannot directly send their data to the central gateway but send the data to an intermediate or adjacent mote; also known as multi-hop communication. Hereby, the communication is considered to be unicast. Thus, for each mote, there has to be at least one path (via some adjacent motes) to the gateway. This means that a mote might have more than one adjacent mote and thus the possibility to send a data packet towards the gateway.

A mote is individually configurable. More specifically, a mote is associated with an adaptable transmission power. The higher the transmission power the lower the probability of packet loss, but at the cost of higher energy consumption. Moreover, the proportion for motes with more than one option to send data can be adapted as well. Thus, traffic can be sent to motes with lower packet loss potential, taking into account their transmission power to balance the energy consumption. The challenge now is to find an optimal configuration of the motes such that packet losses and energy consumption are minimised.

However, some uncertainties make it difficult for static systems to find an individual system configuration such that the reliability of the system in terms of packet loss and energy consumption is maintained at the same time. These uncertainties refer to fluctuations in traffic load and wireless interference. Fluctuations in traffic loads emerge from the varying number of data packets that can be emitted by a mote. Each sensor deployed on a mote emits a certain number of data packets in equidistant time cycles; more specifically, between 0 and 10 data packets per cycle. The probability that a mote produces data packets during a cycle is captured by its activation probability, e.g. if the activation probability of a mote is 0.6, 6 data packets are produced and sent within a cycle. Fluctuations in traffic loads are dependent on the type of sensor. For instance, while temperature sensors produce sensor data in equidistant time steps, passive infrared and RFID sensors are more active during the daytime operation of the university and thus more frequently emitting sensor data.

Wireless interference refers to disturbances in the environment or rather during the communication between motes that can cause a failure in the transmission of data packets and increases the overall packet loss of the system. More specifically, the packet loss depends on the *Signal-to-Noise Ratio* (SNR) of a wireless link, i.e. the communication link between two motes. SNR is defined as the ratio between the level of a mote's signal when sending data and the level of a noise signal from the environment (e.g. wireless interference). The SNR of a wireless link is increased when the transmission power of a mote is increased; that is, the higher the SNR the lower the probability of packet loss and vice versa.

Fluctuation in traffic load and wireless interference are two variables that change over time and make it quite challenging to adjust the configurations of the individual motes to maintain a certain level of packet loss and energy consumption. Especially for IoT networks such as DeltaIoT where the network topology encompasses a lot of configurable motes. Therefore, self-adaptive capabilities are required to compensate for such uncertainties. More specifically, the adaptation problem is to adapt the transmission power and data distribution of motes to maintain packet loss and energy consumption in the presence of wireless interference and fluctuations in the traffic load.

1.5.3. Human-Robot-Interaction

The last example system is taken from the domain of *Human-Robot-Interaction* (HRI) and is based on the work of Timmermann et al. [188]. The HRI system was designed as part of the *CyberProtect*-project[1] in which parts of this thesis were also elaborated.

[1] https://www.cyberprotect-bw.de/

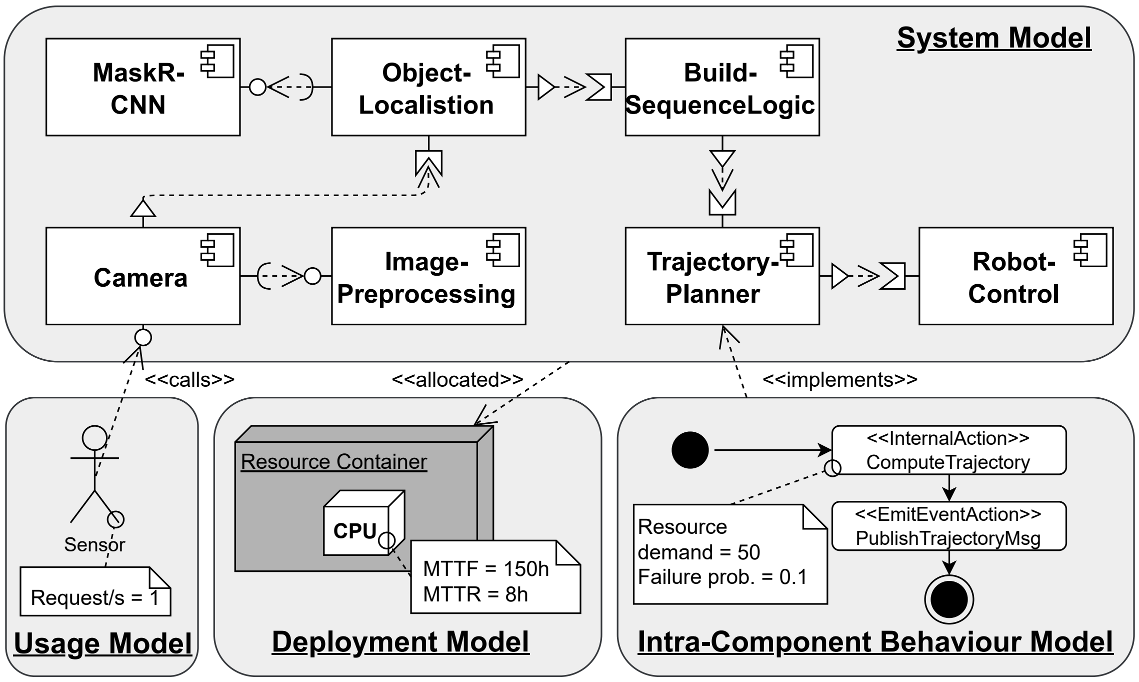


Figure 1.3.: PCM instance of the HRI system [188] in a UML-based notation (taken from [159]).

It represents a robot system that controls a robot arm which is attached to a camera to recognise and assemble object parts. Figure 1.3 shows the simplified architecture of the HRI system. It builds upon the *Robot Operating System* (ROS) [142] and thus follows a message-based communication. We assume that the system perceives one image per second from the camera. The central part of the system forms the `ObjectLocalisation` component (to which the image is forwarded). In the HRI example, object localisation is implemented by a hybrid approach where computer vision and deep learning techniques are combined to achieve a fast and robust object localisation. More specifically, the deep learning part of the approach encompasses a DNN which detects all object parts within an image; so-called MaskR-CNN's have been used [80, 2]. The results of the MaskR-CNN are used by a computer vision component for object localisation to determine further information such as the orientation or the exact position of the object parts. Based on the localised objects, the `BuildSequenceLogic` component determines the assembling order. Finally, the trajectory of the robot is planned and translated into control signals.

Because the use case involves human interaction (human workers may operate in the same workspace as the robot arm or directly interact with the system), a specific level of safety must be maintained, e.g. a collision with the robot arm can cause injuries such as squeezing. However, the safety of human workers is highly dependent on the reliable detection of the object parts by the AI model (i.e. the MaskR-CNN). For example, if a body part is not detected while assembling object parts, the system computes a trajectory which can directly collide with the human worker. Therefore, it is paramount that the AI model is working accurately and reliably.

For the sake of illustration, we consider two external factors which we assume to have a direct influence on the predictive uncertainty of the AI model, namely brightness variations and sensor noise. Originally, Timmermann et al. [188] used as an AI model a pre-trained MaskR-CNN which they re-trained based on a limited training dataset of the object parts. Therefore, we assume that not sufficient data examples are included in the dataset which covers data examples under various brightness conditions or sensor noise levels. Consequently, the AI model is likely to produce wrong predictions in scenes with varying brightness conditions or sensor noise.

To ensure safety, however, an architectural or system-level approach is supposed to be considered. For the HRI system, we consider the use of a self-adaptive system as an architectural safeguard. Note that also non-adaptive solutions could be taken into consideration; however, due to illustration purposes, we consider a self-adaptive system which can apply two adaptations. The first adaptation activates (or deactivates) an additional preprocessing component to deal with sensor noise. For instance, in Figure 1.3 the `ImagePreprocessing` component can be activated or deactivated. For the second adaptation, we assume (in addition to the MaskR-CNN) another AI model which is more robust but computationally expensive. Thus, the self-adaptive system can switch between the two AI models at runtime. However, both adaptations degrade the performance of the system to some extent, i.e. a particular execution time can not be guaranteed, and the system is no longer able to react to new events in time. Thus, the challenge or adaptation problem of the self-adaptive system is to adapt the system in such a way that the performance and reliability of the system are balanced as well as possible, i.e. preventing unsafe states while keeping the system responsive.

1.6. Outline

Before briefly outlining the individual chapters of this thesis, we would like to point out that the structure of the thesis does not directly follow the order of the enumerated contributions. Although we build upon existing model-based approaches, further model-based approaches had to be developed, which are necessary for a design-time evaluation of architectural safeguards. In particular, this concerns the evaluation of adaptation strategies of self-adaptive systems (acting as designated architectural safeguards) and the probabilistic modelling of environments (i.e. the environmental dynamics). Therefore, the contribution-related chapters (i.e. chapters 4-8) are divided into two parts. The first part (i.e. chapters 4-6) generally presents the fundamental concepts and approaches that are independent of the evaluation and analysis of AI-related architectural safeguards. This includes the formal or mathematical framework on which this thesis is based and which runs through all chapters, the domain-agnostic environment model and our *SimExp* method for design-time evaluation of adaptation strategies. The second part (i.e. chapters 7 and 8) takes these concepts and extends them for the evaluation of architectural safeguards of AI components. That being said, we structured the thesis as follows:

In **Chapter 2**, we introduce all relevant foundations necessary to understand the concepts and terminology of this thesis.

In **Chapter 3**, we review and distinguish ourselves from numerous scientific works related to our research.

In **Chapter 4** we discuss the formal framework of our approach, namely MDPs. More specifically, we present the basic concepts and building blocks which map directly to equivalent concepts in MDPs and on which the subsequent approaches are based. In addition, based on the semantics of MDPs, we prove that self-adaptive systems follow a dynamic behaviour under certain assumptions.

In **Chapter 5**, we present the metamodel of our environment model. For the sake of generality, we discuss the environment (or the environmental dynamics) from the perspective of self-adaptive systems. At its core, however, the metamodel is based on the concepts of *Dynamic Bayesian Networks* that extends *Bayesian Networks.* We use Bayesian networks to describe the relationships between environmental variables and the predictive uncertainty of an AI component.

In **Chapter 6**, we present our *SimExp* method. The *SimExp* method encompasses a framework one can use to evaluate adaptation strategies of any self-adaptive system in any domain. *SimExp*, however, represents not a ready-to-use tool but rather a method (or framework) one must instantiate and, if necessary, complement with domain-specific elements.

In **Chapter 7**, we expand the concepts elaborated in the previous chapters to evaluate architectural safeguards. Hereby, we start to present our reliability prediction approach for AI-enabled systems. Afterwards, we discuss how to evaluate self-adaptive systems as architectural safeguards for AI components by combining our reliability prediction approach with the *SimExp* method presented in chapter 6.

In **Chapter 8**, we introduce our classification structure for assessing AI-enabled systems in terms of making assurances for a particular system-level property. First, we start to discuss our classes of architectural dependability assurance and the classification dimensions we identified. Finally, we apply the structure to a representative set of AI systems and discuss the results afterwards.

We validate our approaches in **Chapter 9**. Our validation is driven by the *Goal-Question-Metric* approach that we present in the beginning. We validate our approaches by considering four case study systems. In the end, we discuss the results and answer the research questions.

Finally, we conclude the thesis in **Chapter 10** by providing a summary and discussion of future work.

Part II.

Foundations and Related Work

2. Foundations

In this section, we review all relevant foundations that are needed to understand the concepts presented in this thesis.

2.1. Self-Adaptive Software Systems

Opposed to static software systems, self-adaptive systems adapt their structure and behaviour at runtime. Thus, self-adaptive systems can operate in dynamic environments where a static software system is not able to satisfy the quality requirements of the system. In this thesis, we focus on self-adaptive systems that follow the MAPE-K paradigm [101]. The MAPE-K paradigm follows a feedback loop. Figure 2.1 depicts the main elements or generic structure of a MAPE-K feedback loop.

Basically, the MAPE-K paradigm distinguishes between *Managed Elements* and the *Autonomic Manager*. The former corresponds to the elements of a software system which are adapted at runtime. The autonomic manager in turn is responsible for adapting the managed elements. Both together form a self-adaptive system.

More specifically, the autonomic manager defines four phases: *Monitor*, *Analyse*, *Plan* and *Execute* (this explains the first four letters of the acronym MAPE). In the monitor-phase, the autonomic manager gathers data on the environment or system variables which are necessary for determining the state of the system. Moreover, it serves as a basis for the next phase. The analyse-phase checks based on the collected data of the monitor-phase whether

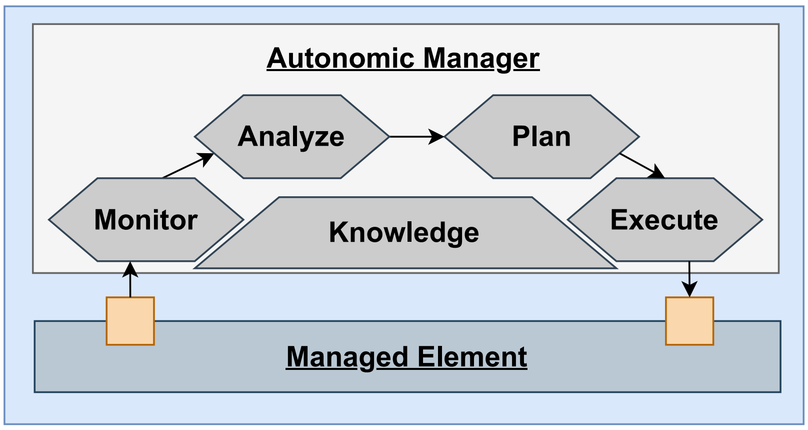


Figure 2.1.: Basic structure of the MAPE-K feedback loop based on [101].

an adaption of the system is necessary, e.g. by checking whether quality objectives are not satisfied anymore. If so, the analyse-phase triggers the plan-phase. The plan-phase determines a specific adaptation to cope with the current situation and to restore a stable system state. Finally, the execute-phase applies the planned adaptation to the managed elements. Note that we use the term quality objectives in the context of a self-adaptive system to refer to a set of non-functional (or quality) requirements that are maintained by a self-adaptive system.

The concept of the *Knowledge* refers to a knowledge base that encompasses information on the system structure, domain information, assumptions, etc. necessary for the individual MAPE phases. The MAPE phases as well as the knowledge base constitute a MAPE-K feedback loop.

2.2. Model-driven Software Development

Model-Driven Software Development (MDSD) [198] defines a collection of techniques and methods centred around models. A model provides an abstract point of view of a software system. Thus, the complexity is reduced and provides developers with good means for communication, discussion and documentation. Strongly related to MDSD is an approach known as *Model-Driven Architecture* (MDA) [103] which was introduced by the *Object Management Group* (OMG). The very basic idea of MDA is to use models gradually refined with more details in later development stages. In the beginning, a *Platform Independent Model* (PIM) is generated which is exclusively concerned with platform-independent system operations. Based on the PIM, the *Platform Specific Model* (PSM) is developed to complement the PIM with platform-specific information. Based on the PSM model, code can be generated. Essentially, MDA is about using abstraction at the early development stages of a software product which is iteratively refined as soon as more requirements and information are getting more concrete.

2.2.1. Models and Metamodels

In this section, we discuss the foundations of *Models* and *Metamodels* as both concepts are of paramount importance for this thesis. The central element of MDSD is a model. Therefore, the concept of a model must be defined, which we will do in the following based on the work of Stachowiak [175] on general model theory:

Definition 1 (Model based on Stachowiak [175]). *A model is a formal representation of real-world entities and their relationships in which*

- *only the details relevant for understanding are captured (abstraction)*
- *a certain correspondence is maintained (homomorphism)*
- *a specific intention is to be illustrated (pragmatics)*

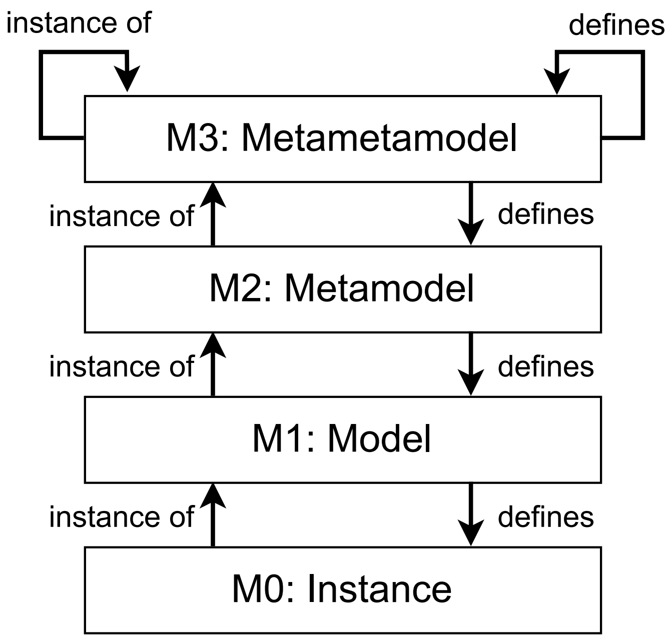


Figure 2.2.: The meta-levels in models adopted from [176].

A model is associated with a set of rules that define how a model is to be created, i.e. which elements the model contains and how the elements relate to each other. Such structure-defining rules are defined within a so-called metamodel. In terms of metamodels, we reuse the definition provided by Koziolek [106, P.43] which is based on Stahl and Völter [176].

Definition 2 (Metamodel)**.** *"A metamodel is a formal model that describes the possible models for a domain by defining the constructs of a modelling language and their relationships (abstract syntax) as well as constraints and modelling rules (static semantics)." [106, P.43] (adapted from Stahl and Völter [176]).*

A model is considered to be an instance of its corresponding metamodel. Also, a metamodel may have a metametamodel while this can again be described by a metametametamodel. Thus, from a theoretical point of view, there exist infinite meta-levels. In practice, however, only a few meta-levels are considered. More concretely, the Object Management Facility (OMG) and their Meta Object Facility (MOF) consider four meta-levels (see Figure 2.2).

The idea of MOF is to provide a standardised metametamodel. Based on the standardised metametamodel at meta-level M_3, metamodels (M_2) can be created which in turn allow the instantiation of models (M_1) capturing real-world entities and their relationships (M_0).

2.2.2. Model Transformation

In this section, we explain *Model Transformations* which we mainly use within this thesis to abstract adaptations of self-adaptive systems. Informally, a model transformation transforms a model into another model, e.g. it transforms a state machine model into a Petri-Net model. However, this requires that both models have common semantics, otherwise a transformation would not be possible. Transformations of this kind are called *Model-to-Model Transformations.*

Definition 3 (Model-to-Model Transformation)**.** *A model-to-model transformation defines a set of rules that map elements from a source metamodel to a target metamodel. [198]*

Generally, there is a distinction between *Exogenous* and *Endogenous* transformations. Transformations that transform models from one metamodel into another are called exogenous. More formally, let M and M' be two models where $\mathcal{MM}(M)$ corresponds to the source and $\mathcal{MM}(M')$ to the target metamodel, a transformation is exogenous, if and only if $\mathcal{MM}(M) \neq \mathcal{MM}(M')$. On the contrary, a model transformation is called endogenous or *in-place* model transformation, if $\mathcal{MM}(M) = \mathcal{MM}(M')$ holds. In the context of this thesis, we are mainly concerned with endogenous or in-place transformations.

2.2.3. EMF Profiles

Metamodels can be very expressive and grow over time. To some extent, the extension of a complex and large metamodel requires a lot of effort. For this reason, a new extension method has been established that applies so-called *Profiles* at metamodel level to the respective metaclasses to be extended. Profiles define additional attributes or references for which the metaclass is to be extended. In the context of the *Unified Modeling Language* (UML) [133], for example, *UML Profiles* have been established. In this work, however, we consider *EMF Profiles* that correspond to the equivalent concept to UML profiles in the context of the *Eclipse Modeling Framework* (EMF) [177]. EMF provides a framework to develop metamodels based on the Ecore notation and is fully integrated into the Eclipse development environment. The presented concepts of this thesis are mainly implemented in EMF. Therefore, we also make use of EMF profiles. However, we are not using EMF profiles to extend existing metamodels but to annotate them. In later chapters, we will see that using EMF profiles as annotations enable us to apply our approach independent of the considered modelling language for describing software architectures.

2.3. The Palladio Approach

In this section, we introduce the *Palladio* approach [149]. The Palladio approach provides an ADL (architectural description language) to model component-based software architectures, namely the *Palladio Component Model* (PCM) (see section 2.3.1). Complementary to PCM, the Palladio approach provides a collection of simulation and analysis tools which can be used to predict quality attributes based on the modelled PCM instances (see section 2.3.2). However, before we delve into the details of PCM and its simulation/analysis tools, we introduce two more definitions which are of paramount importance, namely the concept of *Software Architecture* and *Software Component*. In literature, there are many definitions and understandings of both concepts. For clarification, however, we introduce the respective definitions of what we define as software architecture and software components.

In this thesis, we focus on component-based software architectures, where software components are the central building blocks. Therefore, we first define software components from Reussner et al. [149]:

Definition 4 (Software Component)**.** *"A software component is a contractually specified building block for software, which can be composed, deployed and adapted without understanding its internals." [149, P.47]*

Based on the definition, the software architecture is defined by Reussner et al. [149] as follows:

Definition 5 (Software Architecture)**.** *"A software architecture is the result of a set of design decisions relating to the structure of a system with components and their relationships as well as their mapping to execution environments." [149]*

2.3.1. Modelling Component-based Software Architectures

In this section, we describe the PCM. The PCM is an ADL for component-based software architectures which we use in the context of this thesis as modelling language to describe software architectures. In the following, we introduce the main elements of the language. Moreover, we describe the template language *Architectural Templates* (ATs) which is based on the PCM. ATs describe a language to model architectural knowledge (e.g. architectural patterns or reference architectures). Because we use ATs in this thesis to describe architectural patterns, we introduce the main concepts as well.

2.3.1.1. The Palladio Component Model

In principle, the PCM comprises a collection of distinct models where each of which describes a specific view or aspect of the entire software architecture. There are three viewpoints within the PCM, namely *Structural Viewpoint*, *Behavioural Viewpoint* and *Deployment Viewpoint*. Each viewpoint is associated with a set of models. All models together form a PCM model instance that describes the architecture of a software system.

Structural Viewpoint The structural viewpoint is associated with two models, namely the *Repository Model* and *System model.* The repository model describes all *Software Components* of a software system (which form the main building blocks of the software architecture) and their dependency structure. Each component can provide and require an *Interface.* An interface describes a set of services which is either provided or required by a component. A service is represented by its service or function signature which encompasses the service name, input arguments and the corresponding output value. The requiring and providing semantics connect components where one component provides an interface which is required by another component. Moreover, it also dictates synchronous communication between two components, e.g. consider the load balancing system where the load balancer component is synchronously communicating with the application server. In some scenarios, however, software systems rather follow a different type of communication, i.e. event-based communication. Recall, for example, the HRI system using the ROS (robot operating system) which dictates message-based communication. In such settings, there is no direct communication between components, only an event channel where events are published and consumed by other components. In event-based communication, components are not connected by interfaces but by *Event Groups*, which in turn do not specify function signatures but event types. Each event type can define a set of input arguments. In contrast to providing and requiring components, one component can act as a source (i.e. an event-producing component), while another component acts as a sink (i.e. event-consuming component).

The system model defines the runtime model. More specifically, it defines a set of *Assembly Contexts* which instantiates components of the repository model. Consequently, not every component of the repository model must be instantiated in the system model but only a subset. For example, if there exist several components providing the same interface (i.e. design or implementation options), only one is eventually selected and considered in the system model. Furthermore, the system model describes the concrete connection of components based on the providing/requiring or source/sink semantics (which depends on the type of communication technology). More precisely, one may connect components by creating so-called *Assembly Connectors.* All assembly contexts and connectors are maintained within a *System* which represents the whole software system. A system provides one or more interfaces that are visible to the environment in which the system operates, i.e. they describe the services that the system provides and that can be requested by users.

Behavioural Viewpoint The behavioural viewpoint comprises two models, namely the *Service Effect Specification* (SEFF) and *Usage Model.*

In PCM, the intracomponent and intercomponent behaviour is distinguished. The intra-component behaviour refers to the internal behaviour of the component itself which is modelled by the SEFF. More specifically, for each component operation or service, there exists a SEFF describing the internals of the component when invoking the respective operation provided by the component. SEFFs define *Abstract Actions* which models the control flow of a component based on finite state machines. An abstract action can have different subtypes. For example, an *Internal Action* represents a processing task which is associated with a certain resource demand. *External Call Actions* model external service calls of a requiring component to a providing component. An *Emit Event Action* is the counterpart of an external call action in terms of event-based communication. Moreover, there are further actions such as *Branch Actions* and *Loop Actions* to model the abstract control flow of a component. On the contrary, intercomponent behaviour models the interaction between models. However, because the intercomponent behaviour is implicitly modelled by the system model and the external calls of the intracomponent behaviour, there is no dedicated model.

The usage model captures the usage behaviour of the users interacting with the system. The usage model allows domain experts to model so-called *Usage Scenarios*. In a usage scenario, the user's interaction with the system is captured in a way that resembles activity diagrams. In this context, a domain expert must model a *Workload* specification. There are two kinds of workload: *Open Workloads* and *Closed Workload*. Closed workloads models a population of users (i.e. a fixed number of users) that circulate in the system (i.e. who are repeatedly interacting with the system). In contrast, open workloads capture the frequency with which the system is requested by a user. Hereby, either the *Arrival Rate* (i.e. the number of requests per time unit) or *Interarrival time* (i.e. the time between user arrival which is derived by the arrival rate: $1/ArrivalRate$) of the system can be modelled. Finally, a domain expert can model the distribution defined over the distinct interarrival times to capture the user behaviour more accurately.

Deployment Viewpoint The deployment viewpoint subsumes all models which provide information about the deployment of the system components. More specifically, two models include deployment information, namely the *Allocation Model* and *Resource Environment*.

The resource model defines the available *Resource Containers* (e.g. physical nodes such as servers) on which software components can be deployed. A resource container consists of several processing resources (e.g. CPU or HDD) which can be requested by the software components (and which have a tremendous effect on the system performance). Moreover, the resource environment models *Linking Resources* (i.e. network connection such as LAN or WAN) between the distinct resource containers.

Based on the resource environment, the allocation model describes the allocation structure of the assembly contexts of the system model. More specifically, it models the mapping of an assembly context to a specific resource container.

2.3.1.2. Architectural Templates

In this section, we present the *Architectural Template* (AT) approach of Lehrig [113] for templating *Architectural Knowledge*. Architectural knowledge is defined by Kruchten et al. [108] as design decisions and the design of the software architecture, i.e. not only the design but also the involved design decisions. In some domains, however, gained architectural knowledge is transferable to related domains which face comparable design problems. Therefore, it is of tremendous importance to document architectural knowledge to make them reusable in various application contexts. Examples of reusable architectural knowledge are *Architectural Patterns*, *Architectural Styles* and *Reference Architectures*. In this thesis, we focus on architectural patterns which are defined as follows:

Definition 6 (Architectural Pattern)**.** *"An architectural pattern is a named collection of architectural design decisions that are applicable to a recurring design problem, parametrized to account for different software development contexts in which that problem appears." [122, P.73]*

Lehrig's AT approach accommodates the reuse of architectural knowledge by providing a formal modelling language for describing ATs, i.e. for modelling architectural patterns, architectural styles and reference architectures. The approach has been initially implemented in the context of Palladio and PCM; thus, it is fully compatible with PCM. The definition of an AT involves three steps. First, an AT must be defined and added to a central *Architectural Catalog* that can be browsed by a software engineer and contains all applicable ATs. An AT itself contains a set of *AT Roles* and a *Quality Completion*. The AT roles correspond to EMF profiles which are applied to annotate the elements of the architecture model (or PCM model) that are affected by the AT. More precisely, the roles specify the place or part of the architecture to which the AT is to be applied. In addition, they can also introduce architectural constraints that must not be violated. For example, in a three-tier architecture, components assigned to one of the tiers (i.e. presentation, application and data access tiers) may only request components from lower tiers. Finally, to evaluate the effect of an AT on the quality attributes, a quality completion must be implemented. Quality completion refers to an in-place model transformation that completes the PCM model with AT-specific elements. Recall the load balancer example system from section 1.5.1. The concept of a load balancer can also be modelled as an architectural pattern (see [113]). In this case, the corresponding model elements of the PCM model (concerning the load balancing system of the `AppServer`) have to be annotated in such a way that the quality completion transforms the model by inserting the load balancer component and replications of the `AppServer`. Depending on the AT used, the quality completions are different. In this thesis, we use ATs to model architectural patterns to safeguard or improve the reliability of AI-enabled software systems.

2.3.2. Simulating Component-based Software Architectures

In this section, we briefly provide an overview of the simulation and analysis capabilities of the Palladio framework.

2.3.2.1. Predicting Software Quality Attributes

Originally, the idea of the Palladio approach was to predict performance attributes of modelled PCM instances. This idea was generalised by allowing the prediction of other quality attributes (e.g. reliability) and not only performance. In the process, the PCM was extended by further concepts to enable the prediction of further quality attributes. Thus, the Palladio approach defines an overall procedure that can be extended by further simulation and analysis tools to predict a whole range of quality attributes.

In the context of this thesis, we will reuse the performance simulator *SimuLizar* ([16, 17]) to predict performance attributes (more specifically, the response time of the system). We reuse, however, SimuLizar not as part of our approach but as a tool to quantify a system state. In addition to SimuLizar, there is also the PCM-Rel analysis tool [33] to predict reliability attributes of PCM instances, which we extend in this work to enable the prediction of reliability attributes for AI-enabled software systems. Since the internals and semantics of PCM-Rel are of greater importance, PCM-Rel is discussed in more detail in the next section. Note, however, that there are more simulation and analysis tools (we refer to reference [149] for more details).

2.3.2.2. Architecture-based Software Reliability Prediction

Before we explain the details of the PCM-Rel approach for predicting the reliability attributes of a PCM model, we first define what we consider under the term software reliability. Therefore, we consider the IEEE 1633 standard in which software reliability is defined as follows:

> "The probability that software will not cause the failure of a system for a specified time under specified conditions." [90]

Based on this definition, we present the PCM-Rel approach [33, 34].

In essence, PCM-Rel enables the prediction of the success probability of a PCM-modelled software system. More specifically, the success probability is determined for the execution of a service. The service execution depends on the execution paths induced by the intra- and intercomponent behaviour of the distinct components of the software architecture. Moreover, the success probability depends on the given usage behaviour, i.e. the sequences of service calls modelled by a usage scenario.

PCM-Rel extends the PCM by annotating particular model elements with failure types. More specifically, there are three distinct failure types, namely *Software-*, *Hardware-* and *Network Failure Type*. Software failure types describe the failure potential due to faults in the implementation of software components. In terms of PCM, internal action elements are annotated by failure probabilities because they refer to internal processing units (i.e. implemented code). Hardware failure types represent the failure potential of observing hardware failures of hardware resources (i.e. the probability that at a certain time instance a server is unavailable). In the context of PCM, resource containers are annotated by

Mean-Time-To-Failure (MTTF) and *Mean-Time-To-Repair* (MTTR). Based on the MTTF and MTTR specifications, the availability of a resource container i is calculated as follows:

$$Av(i) = \frac{MTTF_i}{MTTF_i + MTTR_i} \tag{2.1}$$

Let X_i denote the random variable describing the state of a resource container i which is either available (*OK*) or not available (*NA*), i.e. $Val(X_i) = \{OK, NA\}$. The probability of failure/success of resource container i is equivalent to the probability of having i either in state OK or NA:

$$\begin{aligned} Pr(X_i = OK) &= Av(i) \\ Pr(X_i = NA) &= 1 - Av(i) \end{aligned} \tag{2.2}$$

Finally, a network failure type refers to the failure probability observed during the communication of components (i.e. intercomponent behaviour). In this case, a linking resource of the PCM model is annotated with a corresponding failure probability.

Recall that the resource environment of PCM models all resource containers. Whenever we use multiple resource containers on which software components are allocated, we also observe distinct resource failure patterns. For instance, for two resource containers, four different failure situations are possible. More generally, for l available resource containers, 2^l different resource failure patterns are possible. Based on the hardware failure annotations all possible *Physical System States* (i.e. the distinct resource failure patterns) are determined. The probability of observing a particular pattern of physical states $\psi \in \Psi := \{OK, NA\}^l$ is $Pr(X_\Psi = \psi) = Pr(X_1 = \psi_1, ..., X_l = \psi_l)$. Moreover, it is assumed that the individual resource failures are stochastically independent, i.e. $Pr(X_1 = \psi_1, ..., X_l = \psi_l) = \prod_{i=1}^{l} Pr(X_i = \psi_i)$. The main reason why we consider different physical state patterns ψ is that each of which has a different effect on the system's success probability $Pr(X_{Sys} = Success)$. Recall that each usage model U defines a set of usage scenarios. Each usage scenario is associated with a different user behaviour (i.e. different interaction with the system) that results in different execution or service invocation paths. For a given physical state pattern, the service invocation path may contain services that are provided only on available and functioning servers. If, on the other hand, the path of service invocations contains services that are provided on non-functioning servers, a system failure is observed. Therefore, the effect of ψ on $Pr(X_{Sys} = Success)$ is evaluated w.r.t. U. Without going into too much detail, this is achieved by generating and evaluating an absorbing *Discrete-Time Markov Chain* (DTMC) (see section 2.4.1). For each ψ a DTMC is generated w.r.t. the internal software failure of the intracomponent behaviour and failures of intercomponent behaviour descriptions (i.e. network failures). Finally, the overall system's success probability is evaluated by determining all physical states and their weighted effect on the system's success probability:

$$P(X_{Sys} \mid X_U) = \sum_{\psi \in \Psi} P(X_{Sys} \mid X_\Psi = \psi, X_U) \cdot Pr(X_\Psi = \psi) \tag{2.3}$$

In chapter 7, we will discuss how we extend PCM-Rel (based on equation (2.3)) to predict the reliability attributes of AI-enabled software systems.

2.4. Markov Models

In this section, we introduce *Markovian Processes* or Markov models. More specifically, we discuss three well-known Markov models, namely *Discrete-time Markov Chains* (DTMCs), *Markov Decision Processes* (MDPs) and *Partially Observable Markov Decision Processes* (POMDPs).

2.4.1. Discrete-time Markov Chain

The most basic Markov model forms *Markov Chains*. Markov chains are stochastic processes where the *Markov Assumption* applies:

Definition 7 (Markov Chain)**.** *A Markov chain is defined by a family of random variables* $(X_t)_{t\in\mathbb{N}}$ *where the Markov assumption applies:*

$$Pr(X_{t+1} = x_{t+1} \mid X_t = x_t, \ldots, X_1 = x_1, X_0 = x_0) = Pr(X_{t+1} = x_{t+1} \mid X_t = x_t) \tag{2.4}$$

In other words, the Markov assumption states that the probability of transitioning to a state x_{t+1} at time $t+1$ depends exclusively on the last state x_t at time t but not on the entire history, i.e. $x_0, \ldots, x_t$. Generally, it is distinguished between *Discrete-time Markov Chains* (DTMCs) and *Continuous-time Markov Chains*. For DTMCs, discrete-time instances t are considered in which the stochastic process evolves. On the contrary, continuous-time Markov chains move in continuous time through the state space. In the context of this work, however, we only focus on DTMCs. Moreover, we consider DTMCs where the *Stationary Assumption* [105, P. 202] (also called homogeneous or time-invariant) holds:

$$\begin{aligned} &\forall t, t' \in \{0, 1, \ldots, T\} : \\ &\qquad Pr(X_{t+1} = x_j \mid X_t = x_i) = Pr(X_{t'+1} = x_j \mid X_{t'} = x_i) \end{aligned} \tag{2.5}$$

That is, the probability of evaluating how the system transitions from x_i to x_j is independent of the current time instance.

Throughout this work, we consider a DTMC as tuple (S, S_0, t) consisting of three elements, namely state space S, initial distribution S_0 and transition function $t : S \times S \longrightarrow [0, 1]$. Hereby, the transition function t refers to the probability distribution of transitioning from state s at time t to state s' at time $t+1$, i.e. $(s, s') \mapsto Pr(X_{t+1} = s' \mid X_t = s)$. Finally, the set $S_0 \subseteq S$ defines the set of initial states (following a particular probability distribution) in which a DTMC may start.

2.4.2. Markov Decision Process

This section introduces *Markov Decision Processes* (MDPs), which form an elementary concept of this thesis. In section 2.4.1, we introduced DTMCs, i.e. stochastic processes where the Markov assumption applies. MDPs extend DTMC (or Markov chains in general) by taking into account two further concepts: *Actions* and *Rewards.* In the following, we briefly discuss the main elements of MDPs but refer to [180, P.37] for more details.

MDPs are commonly explained by an agent that interacts with an environment. More specifically, at some point in time t, the agent receives the current state $s_t \in S$ of the environment and selects an action $a_t \in A$ from an action set A w.r.t. s_t. Afterwards, at time $t+1$, the agent receives a new state s_{t+1} and a reward $r_{t+1} \in R$ evaluating the decision of selecting action a_t in state s_t. As a result of the interaction between agent and environment, one can observe a *Trajectory* (also referred to as *Episode*) that has the following form:

$$s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, \dots \tag{2.6}$$

The trajectory illustrates how stochastic processes described by MDPs are different from DTMCs. The state transitions are additionally affected by the choice of an action selected at time t. Probabilistically speaking, this is expressed by the transition function $t : S \times A \times S \longrightarrow [0, 1]$, i.e. $(s, a, s') \mapsto Pr(X_{S_{t+1}} = s' \mid X_{S_t} = s, X_{A_t} = a)$. Each decision of selecting an action in a given state is evaluated by a reward $r \in R$ which is a numerical value, i.e. $R \subset \mathbb{R}$. This idea is captured by the reward function $r : S \times A \times S \longrightarrow R$ that determines the expected reward, i.e. $(s, a, s') \mapsto \mathbb{E}[X_{R_{t+1}} \mid X_{S_t} = s', X_{A_t} = a, X_{S_t} = s]$. In summary, an MDP is defined as follows:

Definition 8 (Markov Decision Process). *A Markov decision process (MDP) is defined by the tuple $\lambda := (S, A, t, r)$ where*

- *S is a finite set of states.*
- *A is a finite set of actions.*
- *t is a conditional probability distribution determining the probability of transitioning to state s' at time $t+1$ given the current state s and the selected action a at time t.*
- *r is the reward function determining the expected reward after transitioning from state s w.r.t. action a at time t to s' at time $t+1$.*

The last concept related to MDPs is the so-called *Policy.* A policy implements the decision procedure that determines the action to be taken in a given state. More formally, a policy $\pi : A \times S \longrightarrow [0, 1]$ is a conditional probability distribution $(s, a) \mapsto \pi(a \mid s) = Pr(X_A = s \mid X_S = s)$ that evaluates the probability of selecting an action in a given state. The primary objective in MDPs is to develop such a decision procedure or policy π that maximises the expected reward over time. This idea is captured by so-called *Value Functions* $v_\pi(s)$

which are defined to be the expected reward when starting in state s and following policy π (taken from [180, P.46]):

$$\forall s \in S : v_\pi(s) = \mathbb{E}_\pi[X_{G_t} \mid X_{S_t} = s] = \mathbb{E}_\pi\Big[\sum_{k=0}^{T} \gamma^k X_{R_{t+k+1}} \Big| X_{S_t} = s\Big] \tag{2.7}$$

X_{G_t} is used as a notation to abbreviate the accumulated reward over time, i.e. $X_{G_t} := \sum_{k=0}^{T} \gamma^k X_{R_{t+k+1}}$. Hereby, T denotes the final time step. The parameter γ, $0 \leq \gamma \leq 1$, is called *Discount Rate* and has two main purposes. First, it ensures that in situations where $T = \infty$ the accumulated reward is not infinite (when $\gamma < 1$). Second, it determines how strong future rewards are taken into account. For instance, for values of γ close to 0, future rewards are considered less strongly (as γ^k decreases for increasing k's). On the contrary, for values of γ close to 1, future rewards are taken into account more strongly.

2.4.3. Partially Observable Markov Decision Process

An even more general family of Markov models are so-called *Partially Observable Markov Decision Processes* (POMDPs) which expand MDPs. In MDPs, it is implicitly assumed that each state s is fully observable. However, this might not be true in some settings where the state is hidden, i.e. state s can not directly be observed. In such situations, POMDPs are considered which expand MDPs $\lambda := (S, A, t, r)$ by a set of observations Ω and an observation model $o : S \times \Omega \to [0, 1]$ which evaluates the probability to observe $\omega \in \Omega$ in state s. Note that in many other POMDP definitions the observation model also considers the last action. However, the last action is directly reflected by the resulting state such that it can be excluded by the observation model [174].

Definition 9 (Partially Observable Markov Decision Process)**.** *A partially observable Markov decision process (POMDP) is defined as a tuple* (λ, Ω, o)*:*

- $\lambda := (S, A, t, r)$ *describes a Markov decision process.*
- Ω *is a set of observations.*
- o *is a probability distribution determining the probability of observing* $\omega \in \Omega$ *in state* $s \in S$ *at some time instance* t*, i.e.* $(s, \omega) \mapsto Pr(X_\Omega = \omega \mid X_S = s)$*.*

We use POMDPs in later chapters to formalise the hidden state problem induced by AI black-box components.

2.5. Dynamic Programming

In this section, we briefly discuss *Dynamic Programming* (DP). For more details on DP, we refer to [19, 180] (notations are reused from [180]).

Recall from section 2.4.2 the concept of MDPs $\lambda := (S, A, t, r)$ and policies that need to be implemented in a way that maximises the cumulative reward over time. DP refers to a set of algorithms that enable computing an optimal policy π given a perfect model of the environment represented as an MDP. However, before finding an optimal policy, one has to define when a policy π is considered to be better than any other policy π'. Hereby, the value function of equation (2.7) is used which induces a partial ordering over policies:

$$\pi \geq \pi' \Leftrightarrow \forall s \in S : v_\pi(s) \geq v_{\pi'}(s) \tag{2.8}$$

Roughly speaking, DP can be seen as an iterative process that repeatedly goes through two sub-processes to find optimal policies. The first sub-process is called *Policy Evaluation* and computes the value function $v_\pi(s)$ of a policy π. The second sub-process is called *Policy Improvement* and identifies individual changes of a policy π that lead to a better policy π' (according to (2.8)). Policy evaluation and improvement are constantly repeated until the process converges to an optimal policy; this whole iteration process is called *policy iteration*. However, the main focus of this paper is evaluation rather than optimisation. Therefore, the concept of policy evaluation in DP is briefly discussed.

2.5.1. Policy Evaluation

Policy evaluation computes the value function $v_\pi(s)$ of a policy π; this is also referred to as the *Prediction Problem*. The core of policy evaluation forms an iterative update approach of a set of equations that are known as *Bellman Equations* (taken from [180, P.47]):

$$\begin{aligned}
\forall s \in S : v_\pi(s) =& \mathbb{E}_\pi[X_{G_t} \mid X_{S_t} = s] \\
=& \mathbb{E}_\pi[X_{R_{t+1}} + \gamma X_{G_t} \mid X_{S_t} = s] \\
=& \sum_a \pi(a \mid s) \sum_{s'} \sum_r p(s', r \mid s, a) \\
& \cdot \Big(r + \gamma \mathbb{E}_\pi[X_{R_{t+1}} + \gamma X_{G_{t+1}} \mid X_{S_{t+1}} = s']\Big) \\
=& \sum_a \pi(a \mid s) \sum_{s', r} p(s', r \mid s, a)\Big(r + \gamma v_\pi(s')\Big)
\end{aligned} \tag{2.9}$$

Note that function $p(s', r|s, a)$ is used to abbreviate the conditional probability of observing state s' and reward r at time $t + 1$ given state s and action a at time t, i.e. $p(s', r|s, a) := Pr(X_{S_{t+1}} = s', X_{R_{t+1}} = r \mid X_{S_t} = s, X_{A_t} = a)$. The existence and uniqueness of value function $v_\pi(s)$ of policy π is guaranteed if $\gamma < 1$ or termination is guaranteed (i.e. the final time step T from the value function (2.7) is finite $T < \infty$). An update rule for a given state s is defined as follows (taken from [180, P.60]):

$$\begin{aligned}
v_{k+1}(s) &= \mathbb{E}_\pi[X_{R_{t+1}} + \gamma v_k(X_{S_{t+1}}) \mid X_{S_t} = s] \\
&= \sum_a \pi(a \mid s) \sum_{s', r} p(s', r \mid s, a)\Big(r + \gamma v_k(s')\Big)
\end{aligned}$$

The update rules are implemented w.r.t. the Bellman equations, where v_k denotes the last updated value; such updates are also denoted as *Expected Updates*. The iterative policy evaluation algorithm is shown in algorithm 2.1.

Algorithm 2.1: The iterative policy evaluation algorithm adopted from [180, P.61]

```
Input: The policy to be evaluated $\pi$
Output: Value function $V \approx v_\pi$
$V(s) \leftarrow initialise()$ // e.g. $\forall s \in S : V(s) = 0$
repeat
    $\Delta \leftarrow 0$
    foreach $s \in S$ do
        $v \leftarrow V(s)$
        $V(s) \leftarrow \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a)\Big(r + \gamma V(s')\Big)$
        $\Delta \leftarrow max(\Delta, |v - V(s)|)$
    end
until $\Delta < \epsilon \in \mathbb{R}_{>0}$
```

2.5.2. Monte Carlo Prediction

Generally, *Monte Carlo Methods* refer to a collection of estimation methods that involve probabilistic sampling from a distribution. Regarding MDPs, Monte Carlo methods are commonly used to randomly sample trajectories from the actual or simulated environment which are (e.g.) used to estimate the value function $v_\pi(s)$. One problem with DP is that it requires complete knowledge of the dynamics of the environment (i.e. $p(s', r \mid s, a)$), which, however, might not be known in advance. In some cases, it is possible to draw samples from $p(s', r \mid s, a)$ without knowing the distribution which is sufficient to compute value functions $v_\pi(s)$. This is where *Monte Carlo Prediction* comes in, which estimates value functions $v_\pi(s)$ by sampling from the environment and following the policy π (as dictated by the Bellman equation). If the interaction with the environment is simulated, the generated samples are called *Simulated Experience*.

Monte Carlo prediction is a quite simple approach as it generates trajectories by repeatedly drawing samples from $p(s', r \mid s, a)$ and following π until termination. Now recall that the value of a state s corresponds to the expected reward (or rather expected accumulated and discounted reward) by starting from s and following π. For each generated trajectory, the value of a state s can be estimated by averaging the expected reward observed after the first visit of s. The more trajectories are sampled, the more the estimated value for each state converges towards its true value.

There are two Monte Carlo prediction approaches, namely the *First-Visit Monte Carlo Method* and *Every-Visit Monte Carlo Method*. Generally, when traversing over a trajectory, one encounters several states s which is denoted as a visit to s. The first-visit Monte Carlo method estimates $v_\pi(s)$ by averaging the accumulated rewards observed after the first visit of s in each trajectory. In contrast, the every-visit Monte Carlo method averages the accumulated rewards following all visits of s in each trajectory. The first-visit Monte Carlo prediction algorithm is illustrated in algorithm 2.2.

Algorithm 2.2: The first-visit Monte Carlo prediction algorithm based on [180, P.76]

```
Input: The policy to be evaluated $\pi$
Output: Estimated value function $V \approx v_\pi$
$V(s) \leftarrow initialiseV()$ // e.g.  $\forall s \in S : V(s) = 0$
$R(s) \leftarrow initialiseR()$ // an empty list, for all $s \in S$
repeat
    $\tau \leftarrow sample()$ // Generates a trajectory w.r.t $\pi$
    foreach $s \in S$ do
        if $s \in \tau$ then
            $G \leftarrow acc(s, \tau)$ // The accumulated reward that follows after
              first occurence of $s$
            $append(R(s), G)$ // Appends $G$ to $R(s)$
            $V(s) \leftarrow average(R(s))$ // Averages the returns of $s$
        end
    end
```

2.6. Probabilistic Graphical Models

In this section, we discuss the framework of *Probabilistic Graphical Models* that describe graph-based probabilistic structures. Basically, probabilistic graphical models are used to describe large and complex probability spaces which adhere to or encode specific properties in a graph-based way, e.g. *Bayesian Models* or *Markov Random Fields.* The framework encompasses entire families of probabilistic models; in this section, however, we discuss Bayesian models (i.e. *Bayesian Networks* (BN) and *Dynamic Bayesian Networks* (DBN) and *Template-based Probabilistic Models.* Template-based probabilistic models provide a generic framework which generalises Bayesian models by defining templates of random variables.

In this section, only the concepts necessary for the understanding of this work are explained. For further details, we refer to the reference [105] on which this section is based. Also, all definitions and notations are based on [105].

2.6.1. Bayesian Networks

A BN consists of a set of random variables $\{X_1, ..., X_n\}$. The network encodes conditional independence assumptions of the random variables by a *Directed Acyclic Graph* (DAG) $\mathcal{G}$.

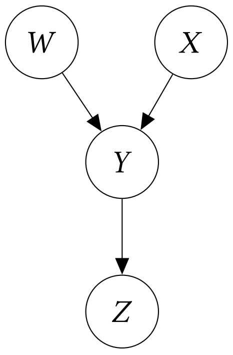


Figure 2.3.: Example Bayesian network consisting of the random variables W, X, Y and Z.

The nodes of graph $\mathcal{G}$ represent the random variables of the network; the edges represent the direct effect of one random variable on another random variable. For example, consider Figure 2.3 which depicts an example BN including four random variables: $\{W, X, Y, Z\}$.

The resulting graph includes four nodes (i.e. $\{W, X, Y, Z\}$) and the edges $W \longrightarrow Y$, $X \longrightarrow Y$, $Y \longrightarrow Z$. The semantics of an edge (e.g.) $Y \longrightarrow Z$ is to be understood as node Z depends solely on its parent Y. In principle, however, an edge indicates conditional independence, e.g. $(Z \perp\!\!\!\perp W, X \mid Y)$. Additionally, nodes of a graph $\mathcal{G}$ that have no parents are stochastically independent, e.g. $(W \perp\!\!\!\perp X)$ and $(X \perp\!\!\!\perp W)$. This encoding of conditional independence assumptions can be generalised for any $\mathcal{G}$. Let $Pa_{\mathcal{G}}(X_i)$ denote the parents of X_i in $\mathcal{G}$ and $NonDescendants(X_i)$ denotes the variables in $\mathcal{G}$ that are non descendants of X_i. A DAG $\mathcal{G}$ representing the structure of a BN with nodes (or random variables) $\{X_1, ..., X_n\}$ encodes the following independence assumptions:

$$\forall i \in \{1, ..., n\} : (X_i \perp\!\!\!\perp NonDescendants(X_i) \mid Pa_{\mathcal{G}}(X_i)). \tag{2.10}$$

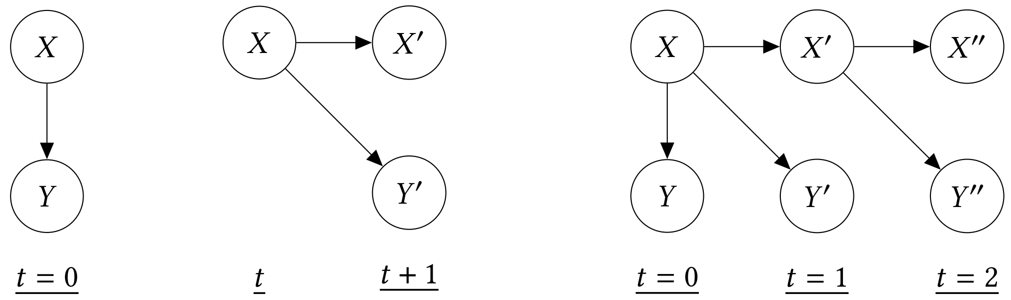
Conditional independence assumptions have favourable ramifications on the joint distribution $P(X_1, ..., X_n)$. More specifically, if distribution P satisfies the conditional independence assumptions encoded by $\mathcal{G}$ and according to equation (2.10) (which we write $P \models \mathcal{G}$), then it can be shown that P factorises over $\mathcal{G}$ (see [105, P. 62]):

$$P(X_1, ..., X_n) = \prod_{i=1}^{n} P(X_i \mid Pa_{\mathcal{G}}(X_i)). \tag{2.11}$$

The individual *Conditional Probability Distributions* (CPDs) $P(X_i \mid Pa_{\mathcal{G}}(X_i))$ complements the structure of a BN. This factorisation property is called *Decomposability*. Decomposability not only allows estimating each $P(X_i|Pa_{X_i})$ individually but also reduces the complexity of the joint distribution. In terms of the example BN in Figure 2.3, the distribution $P(W, X, Y, Z)$ factorises as follows:

$$\begin{aligned} P(W, X, Y, Z) =& P(W) \cdot P(X \mid W) \cdot P(Y \mid W, X) \cdot P(Z \mid W, X, Y) \\ =& P(W) \cdot P(X) \cdot P(Y \mid W, X) \cdot P(Z \mid Y). \end{aligned}$$

Finally, BNs are defined as follows:



(a) Initial BN $\mathcal{B}_0$ **(b)** 2-TBN $\mathcal{B}_{\rightarrow}$ **(c)** Unrolled DBN

Figure 2.4.: Example DBN: (a) represents the initial BN $\mathcal{B}_0$, (b) depicts the 2-TBN $\mathcal{B}_{\rightarrow}$ and (c) shows the resulting DBN unrolled over 3 steps.

Definition 10 (Bayesian Network). *A Bayesian network $\mathcal{B}$ is a tuple $\mathcal{B} := (\mathcal{G}, P)$ where probability distribution P factorises over $\mathcal{G}$, i.e. $P \models \mathcal{G}$.*

2.6.2. Dynamic Bayesian Networks

While BNs enable the representation of compact joined distributions $P(\mathbf{X})$ with $\mathbf{X} := \{X_1, ..., X_n\}$, DBNs describe the stochastic evolution of $\mathbf{X}$ by a transition model. If the Markov and stationary assumptions hold, the transition model forms a CPD: $P(\mathbf{X'} \mid \mathbf{X})$. DBNs are thus a specialisation of Markov processes.

Roughly speaking, DBNs are temporal extensions of BNs where the temporal extension refers to the transition model $P(\mathbf{X'} \mid \mathbf{X})$. Such transition models are represented by so-called *2-Time-Slice Bayesian Network* (2-TBN).

Definition 11 (2-Time-Slice Bayesian Network). *A 2-time-slice Bayesian network $\mathcal{B}_{\rightarrow}$ for a process over $\mathbf{X}$, is a CPD $P(\mathbf{X'} \mid \mathbf{X_I})$ where $\mathbf{X_I} \subseteq \mathbf{X}$.*

The set $\mathbf{X_I}$ is denoted as *Interface Variables* and refers to the set of random variables that have a direct effect on the random variables in $\mathbf{X'}$. Just as in BNs, this dependency is characterised by an edge in a DAG $\mathcal{G}$. Figure 2.4b illustrates a simple 2-TBN.

In the example, the interface variables of the 2-TBN include only X, i.e. $\mathbf{X_I} = \{X\}$. Therefore, X is the only variable that has a direct effect at time t on $\mathbf{X} = \{X, Y\}$ at time $t + 1$. In 2-TBNs, edges between time slice t and $t + 1$ are denoted as *Inter-Time-Slice Edge*, e.g. $X \longrightarrow X'$ and $X \longrightarrow Y'$. Additionally, there might be edges within a time slice that are denoted as *Intra-Time-Slice Edge*. Intra-time-slice edges indicate dependencies between random variables that have an immediate effect, i.e. much shorter than one would observe between variables that are connected by an inter-time-slice edge. Inter-time-slice edges of the form $X \longrightarrow X'$ are denoted as *Persistence Edges*. A random variable X for which there exists a persistence edge tends to persist over time with high probability. Just like BNs, 2-TBNs factorises w.r.t. a dependency structure encoded by a DAG $\mathcal{G}$.

$$P(\mathbf{X'} \mid \mathbf{X_I}) = \prod_{i=1}^{n} P(X'_i \mid Pa_{\mathcal{G}}(X'_i)) \quad \text{where} \quad Pa_{\mathcal{G}}(X'_i) \subseteq \mathbf{X_I}. \tag{2.12}$$

A DBN consists of a BN that forms the initial distribution and a 2-TBN that inductively describes how states are dynamically changing over time (see Figure 2.4). DBNs are unrolled to generate a trajectory of any length. The Markov and stationary assumptions allow a compact representation of the distribution $P(\mathbf{X}_0, \mathbf{X}_1, \dots, \mathbf{X}_T)$ for any T:

$$P(\mathbf{X}_0, \mathbf{X}_1, \dots, \mathbf{X}_T) = P(\mathbf{X}_0) \prod_{t=0}^{T-1} P(\mathbf{X}_{t+1} \mid \mathbf{X}_t) \tag{2.13}$$

Figure 2.4c depicts and unrolled DBN over three time slices. Finally, we conclude this section by formally defining a DBN.

Definition 12 (Dynamic Bayesian Network)**.** *A dynamic Bayesian network is a tuple* $(\mathcal{B}_0, \mathcal{B}_{\rightarrow})$ *which consists of a Bayesian network* $\mathcal{B}_0$ *and a 2-time-slice Bayesian network* $\mathcal{B}_{\rightarrow}$*.* $\mathcal{B}_0$ *defines the initial distribution over the state space;* $\mathcal{B}_{\rightarrow}$ *inductively defines the transition model.*

2.6.3. Template-based Probabilistic Models

Template-based probabilistic models (or simply template models) provide a generic framework to model and instantiate probability spaces for arbitrary object-relational domains. The concepts of template-based models serve as a basis to develop rich languages. In this section, we give a brief overview of template-based models and their fundamental building blocks.

One of the key concepts of template models is *Template Variables* (or template attributes). Template variables encode random variables that have common (domain-specific) semantics and share the same value space. Applying a template variable to an object of a particular domain is to be considered as an instantiation of the template and turns the template into a random variable. Thus, template variables are defined at the type level while random variables are defined at the instance level. Domains, in which templates are supposed to be instantiated, are viewed as being composed of a set of objects. Objects are divided into a set of mutually exclusive classes $\mathcal{Q} = Q_1, ..., Q_n$, i.e. equivalence classes. Template variables have a tuple of *Arguments* where each argument is associated with a specific class Q_i. The arguments of a template restrict the set of objects for which the template can be instantiated (w.r.t. to the class membership of each argument).

Definition 13 (Template Variable)**.** *A template variable* $\mathcal{V}(U_1, ..., U_n)$ *is a function with some range* $Val(\mathcal{V})$*. Each argument* U_i *of* $\mathcal{V}$ *is a typed logical variable where* $Q[U_i] \in \mathcal{Q}$*.*

The tuple $(U_1, ..., U_n)$ is denoted as the *Argument Signature* of $\mathcal{V}$, and abbreviated by $\alpha(\mathcal{V})$.

Template variables generate probability spaces within the domains in which they are instantiated. The probability spaces are induced by a set of objects considered for instantiating the templates. The objects can be divided into mutually exclusive sets (w.r.t. the equivalence classes $\mathcal{Q}$.) and forms an *Object Skeleton.* More formally, let denote $O^{\kappa}[Q]$ the finite set of objects associated with class Q included in object skeleton κ. Template variables are instantiated to the objects considered in skeleton κ by applying them to objects that can be assigned to the logical variables of the argument signature of $\mathcal{V}$. Therefore, the set of possible assignments for a template $\mathcal{V}$ with $\alpha(\mathcal{V}) = (U_1, ..., U_n)$ is defined as follows:

$$O^{\kappa}[(U_1, ..., U_n)] = O^{\kappa}[Q[U_1]] \times \cdots \times O^{\kappa}[Q[U_n]]. \tag{2.14}$$

In some domains, however, specific assignments are not legal. Therefore, $\Gamma_{\kappa}[\mathcal{V}] \subseteq O^{\kappa}[\alpha(U)]$ defines the set of all valid assignments for template $\mathcal{V}$. Based on the previous definitions, for an object skeleton κ and a set $\aleph$ of template variables over $\mathcal{Q}$, the set of instantiations of template variables $\mathcal{X}_{\kappa}[\aleph]$ is defined as follows:

$$\mathcal{X}_{\kappa}[\aleph] = \bigcup_{\mathcal{V} \in \aleph} \{\mathcal{V}(\gamma) | \gamma \in \Gamma_{\kappa}[\mathcal{V}]\}. \tag{2.15}$$

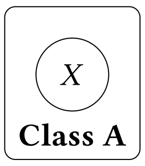
The notation $\mathcal{V}(\gamma)$ indicates an assignment of object tuple $(u_1, ..., u_n)$ to a template $\mathcal{V}$ w.r.t. the argument signature $\alpha(\mathcal{V})$, i.e. $\gamma = (U_1 \mapsto u_1, \ldots, U_k \mapsto u_k)$. An instantiated template variable is also denoted as *Ground Random Variable.*

The second key concept of template models forms *Template Factors.* Template factors complement template variables by a probabilistic specification, i.e. the type of distribution such as multinomial distributions. Just as Template variables, template factors are defined at type-level over a set of template variables. When the set of template variables is instantiated, the template factors can be instantiated to specify the concrete distribution of the ground random variables.

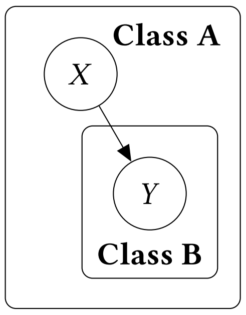
Definition 14 (Template Factor). *A template factor $\xi : Val(\mathcal{V}_1) \times \cdots \times Val(\mathcal{V}_l) \longrightarrow \mathbb{R}$ is a function defined over template variables $(\mathcal{V}_1, \ldots, \mathcal{V}_l)$. Given a tuple of ground random variables $(X_1, ..., X_l)$, if $\forall i \in \{1, ..., l\} : Val(X_i) = Val(\mathcal{V}_i)$ holds true, then $\xi(X_1, ..., X_l)$ defines the instantiated factor from $X_1, ..., X_l$ to $\mathbb{R}$ w.r.t. $(\mathcal{V}_1, ..., \mathcal{V}_l)$.*

2.6.3.1. Plate Models

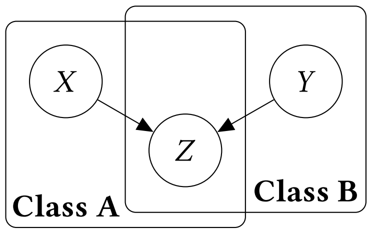
In this section, we briefly discuss a template-based language for probabilistic models, called *Plate Models.* Plate models reuse the key concepts presented in the last section to encode probabilistic structures. The formal semantics of plate models goes beyond the pure template-based concepts and are not in the scope of this work. Therefore, we refer again to reference [105] that discusses plate models in more detail. In the following, we



(a) Simple plate



(b) Nested plate



(c) Intersected plate

Figure 2.5.: Plate model encoding according to [105, P. 219]: (a) represents a simple plate structure, (b) shows a nested plate structure and (c) depicts an intersected plate structure.

introduce basic concepts of plate models (which we reuse in later chapters) and how they are used in modelling Bayesian networks.

In plate models, object types are characterised by *Plates*. Such object types or classes correspond to the equivalence classes $\boldsymbol{Q} = Q_1, \ldots, Q_n$. More specifically, each plate describes a class Q_i. The graphical notation of a plate encloses a random variable with a box. For instance, consider Figure 2.5a which specifies a plate of some object class A for a random variable X. The graphical representation encodes the fact that there are multiple random variables of X in which all share the same template $\mathcal{V}_X$ and are instantiated for several objects of class A.

The example of Figure 2.5a is very simple and can be expanded for multiple random variables and overlapping plates. For instance, the plate structure in Figure 2.5b indicates a *Nested* plate structure. Hereby, the plate of object class B for random variable Y is nested within the plate of object class A. Nested plate structures encode that the template for Y can be instantiated for object pairs of class A and B (as B is embedded in plate A). For each pair, however, there has to be an individual instantiation of $\mathcal{V}_X$ representing the parent of Y.

A third plate structure describes the *Intersection* of plates as depicted in Figure 2.5c. In this case, the template for random variable Z is instantiated for object pairs of class A and B. Similarly to nested plates, for each pair the corresponding parent random variables must be instantiated as well, i.e. $\mathcal{V}_X$ and $\mathcal{V}_Y$, respectively.

Plate models describe probabilistic models with repeated structure and shared parameters. Based on the notion of template variables, plate models induce BNs instantiated in domains with certain object structures. As explained earlier, a set of plates and their associated object classes are semantically the same as the equivalence classes $\boldsymbol{Q}$ that typify the argument signature of a template. Thus, any template variable that is embedded in several plates $Q_1, ..., Q_n$ possesses an argument signature $U_1, ..., U_n$ matching the plate's object class structure, i.e. $Q[U_i] = Q_i$.

Finally, the only missing concept that completes a plate model is the dependency structure of template variables. In plate models, template variables embedded in overlapping plates can depend on template variables in any of these plates. Based on these semantics, the plate model can now be formally defined.

Definition 15 (Plate Model). *For a set of template variables $\mathcal{V} \in \aleph$ with argument signature $\alpha(\mathcal{V}) = U_1, ..., U_n$, let $B_i(\mathbf{U_i})$ denote the variables of the argument signature of parent B_i. A plate model M_{Plate} defines for each template:*

- *A set of template parents $Pa(\mathcal{V}) := \{B_1(\mathbf{U_1}), ..., B_k(\mathbf{U_k})\}$ in which $\forall i \in \{1, ..., k\}$: $B_i(\mathbf{U_i}) \Rightarrow \mathbf{U_i} \subseteq \{U_1, ..., U_n\}$.*
- *A template CPD $P(\mathcal{V} \mid Pa(\mathcal{V}))$.*

Finally, a plate model M_{Plate} and object skeleton κ generate a *Ground Bayesian Network*:

Definition 16 (Ground Bayesian Network). *A Ground Bayesian Network $\mathcal{B}_\kappa^{M_{Plate}}$ is generated by a plate model M_{Plate} and object skeleton κ as follows:*

$$\forall \mathcal{V}(U_1, ..., U_n) \in \aleph, \forall \gamma \in \Gamma_\kappa[\mathcal{V}] : \exists_{=1} \mathcal{V}(\gamma) \in \mathcal{X}_\kappa[\aleph] \tag{2.16}$$

where $\gamma := (U_1 \mapsto u_1, \ldots, U_n \mapsto u_n)$ and for all template parents $\mathcal{V}_{Pa} \in Pa(\mathcal{V})$ of ground random variable $\mathcal{V}(\gamma)$ there exist an instantiated CPD: $P(\mathcal{V}(\gamma) \mid \mathcal{V}_{Pa_1}(\gamma), \ldots, \mathcal{V}_{Pa_k}(\gamma))$.

Note that $\mathcal{V}_{Pa}(\gamma)$ is a shorthand notation for ground random variables of template parents which argument signature is only a subset of $\mathcal{V}(\gamma)$ so that the parent template is only instantiated for a subset of tuple γ. The ground Bayesian network $\mathcal{B}_\kappa^{M_{Plate}}$ forms a joint distribution over $\mathcal{X}_\kappa[\aleph]$ (see (2.15)).

2.7. A Brief Introduction to Artificial Intelligence

In this section, we provide a brief overview of the broad field of AI (artificial intelligence). More specifically, we give a brief overview of the subfields of AI, namely machine learning and deep learning (a subfield of machine learning). It is worth noting, that AI is a fairly large field that goes beyond machine learning and deep learning, e.g. propositional logic or first-order predicate logic (see [56] for a broader introduction). However, machine learning and deep learning are currently the most popular methods associated with AI and embody the current state of the art. In particular, DNNs (deep neural networks) and their inherent complexity lead to today's challenges and the need to safeguard AI. Although our presented concepts generalise to all types of AI models, we briefly introduce the notion of machine learning and some well-known DNNs below.

2.7.1. Machine Learning

Goodfellow, Bengio and Courville [69] provide a brief introduction to machine learning based on the definition of Mitchell [124, P.2]: “A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.” For more detailed introductions to machine learning, we refer to [129, 24].

The abstract definition of a task T refers to the learning task particular to the given domain where a machine learning system is applied. For example, a task might simply refer to face recognition, object detection or more complicated tasks in robotics (such as navigating through space). According to Goodfellow et al. [69], most machine learning tasks are generally classified into the following task types: Classification, classification with missing inputs, regression, transcription, machine translation, structured output, anomaly detection, synthesis and sampling, imputation of missing values, denoising and density or probability mass function estimation. The most common tasks are classification and regression. In classification, the task is to classify input values x of some input space $\mathcal{X}$ to some set of categories $1, \dots, k$ by learning a function $f : \mathbb{R}^n \rightarrow \{1, \dots, k\}$. For example, object detection is a quite familiar example for classification tasks. In general, machine learning is intended to approximate functions of the form $f : \mathcal{X} \rightarrow \mathcal{Y}$ based on a dataset D (discussed later) that contains the relationships to be learned. Regarding classification, $\mathcal{X}$ refers to $\mathbb{R}^n$ (e.g. pixel space) and $\mathcal{Y}$ refers to the distinct categories $\{1, \dots, k\}$. In terms of regression, the function to be learned or approximated is of the form $f : \mathbb{R}^n \rightarrow \mathbb{R}$. In such cases, control signals need to be learned for robot navigation or real estate price prediction.

The performance measure P evaluates the extent to which a task T is accomplished. The effectively used performance measure depends on the considered domain. In many cases (and also in the context of this work), however, the accuracy of an AI model is measured. A fairly popular measure in terms of AI model accuracy forms the *Mean-Squared-Error* (MSE) or *Root-Mean-Squared-Error* (RMSE).

$$\begin{aligned} MSE &= \frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2 \\ RMSE &= \sqrt{MSE} \end{aligned} \tag{2.17}$$

Hereby, $\hat{y}_i$ refers to the predictions produced by an AI model and y_i to the actual output. Moreover, N refers to the number of test data included in a *Test Dataset* D_{Test}. Such datasets are commonly used to measure the performance of an AI model. A test dataset consists of labelled example data of the form (x, y) where x describes some input data and y the corresponding (and correct) output. For instance, in classification tasks, y refers to one of the correct categories $\{1, \dots, k\}$.

Finally, experience E refers to *Training Datasets* from which specific patterns or behaviours are to be learned, i.e. based on training examples that have the same form as the test data. Training datasets form the source of what is considered as experience E. In classification tasks, labelled input data serves as experience to generate a learner f for classifying arbitrary input data. However, machine learning is divided into three categories, namely *Supervised Learning*, *Unsupervised Learning* and *Reinforcement Learning*. Depending on the category, the way how experience is incorporated into the learning process is different. In simple terms, supervised learning is about approximating or learning a function w.r.t. the training dataset such that for new inputs (not included in the training dataset) precise predictions or classifications are made. Hereby, the training dataset includes data annotated by labels or targets which serves as the foundation for the learning process. On the contrary, unsupervised learning considered unlabeled training datasets from which structures (e.g. clustering) or probability distributions (e.g. density or probability mass function estimation) are to be learned. The third category refers to reinforcement learning which is based on a trial-and-error approach. In reinforcement learning, the idea is to learn or train an agent that can take actions and interact with an environment. Based on the taken actions and observed responses of the environment, the agent learns how to deal with the environment. Reinforcement learning is often used in robotic tasks or games (e.g. [125]). All three categories have in common that they are highly dependent on the quality of the training datasets. If the concepts (to be learned) are not sufficiently represented in the training datasets, the learned AI model is likely to perform poorly; this induces one of the main problems concerned with machine learning.

2.7.2. Deep Learning

When people talk about AI today, it is usually about deep learning. Nearly any AI-enabled system integrates a DNN. Deep learning is a subfield of machine learning and, in simplified terms, deals with *Neural Networks* with a deeper layer structure. In principle, a neural network defines three layers, namely the *Input*, *Hidden* and *Output* layer. Each layer consists of a set of neurons which are connected with neurons of adjacent layers. Each layer can be viewed as a function $f^{(i)}$ which (when composing the individual layers) transforms input data x into output data by forwarding the results to the next layer, i.e. $(f^{(3)} \circ f^{(2)} \circ f^{(1)})(x)$ (see [69]). The most common types of neural networks refer to *Feedforward Networks* where no data is fed back into some previous layers. Neural networks with higher depths (i.e. many layers or increased length of composed functions) are called DNNs (deep neural networks). Similarly to neural networks, DNNs are mainly used to approximate some function $f : \mathcal{X} \rightarrow \mathcal{Y}$ which is to be learned by a training dataset D including examples of $x \in \mathcal{X}$ annotated by labels or targets.

The learning or training process can be viewed as an optimisation problem where the main target is to estimate a set of parameters θ which is responsible for determining the prediction results of approximated function $f(x)$ (commonly written $f(x;\theta)$). A widely used optimisation or training algorithm is *Back-Propagation* which refers to a gradient-based training method. The main idea is to propagate the error back to the parameters θ that caused a high deviation w.r.t. a loss function (i.e. a function that measures the deviation of the predictions made and the training data) that is to be minimised. In the following, we consider two types of DNNs, namely *Convolutional Neural Networks* (CNNs)

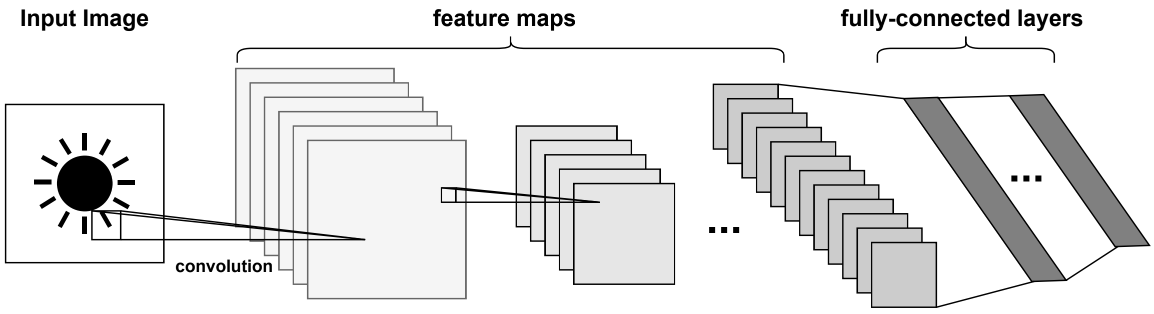


Figure 2.6.: Schematic structure of a CNN adopted from [112].

and *Recurrent Neural Networks* (RNNs). Both DNNs are widely used and also serve as the main object of investigation in the validation part of this work. Again, we refer to reference [69] which discusses both DNNs in more detail.

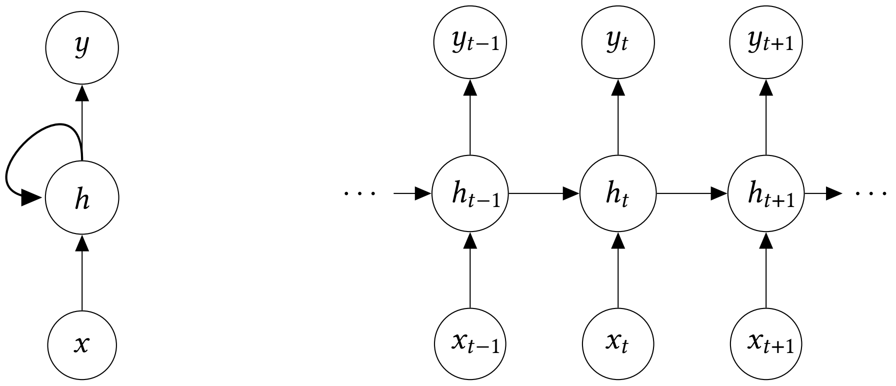
2.7.2.1. Convolutional Neural Networks

Before CNNs gained more attention, software engineers hand-crafted local filters which were applied to extract features from images used for classification. However, hand-crafted features are vulnerable to particular image transformations (e.g. rotation). In CNNs, the entire feature engineering process is part of the learning process in which so-called *Kernels* (representing filters) are learned. In principle, CNNs introduce convolution operations where the input image is convolved by the kernel. The output is denoted as a feature map. The structure of a CNN consists of several convolutional layers followed by a fully-connected layer (see Figure 2.6). The convolutional layers extract features from the input image and are eventually passed on to the fully-connected layer, which is responsible for the actual classification task. Unlike fully connected DNNs, in CNNs some neurons of the convolutional layer are connected to some neurons of the next layer. Moreover, weights or parameters to be learned (i.e. the elements of the kernel) are shared which reduces the number of learned parameters drastically.

The functionality of CNNs is inspired by neuroscience or rather by the visual cortex of the human brain. More specifically, convolutional layers close to the input layer are extracting low-level features (e.g. edges), which become more abstract higher-level features in deeper convolutional layers (i.e. closer to the output layer). Finally, note that CNNs are not only applicable to image data but any kind of data where convolution is applicable (e.g. time-series data).

2.7.2.2. Recurrent Neural Networks

In contrast to feedforward DNNs (which can be viewed graphically as directed acyclic graphs), RNNs allow for feedback or loop-like structures. This means, that outputs or predictions of some layers may be fed back to previous layers (see Figure 2.7). While CNNs



(a) Recurrent structure of an RNN.

(b) RNN unfolded for several time steps.

Figure 2.7.: Example structure of an RNN (based on [69, P.378]) in which x refers to the input, h represents the hidden layers and y the output.

are considered to work on image data, RNNs are rather applied to sequential data. An example application of RNNs is machine translation in which sequences of words (i.e. sentences) are translated from one language to another; also image captioning and action recognition are popular examples.

RNNs are also trained by using the gradient-based back-propagation learning algorithm. However, when unfolding over many time steps, the gradient is also propagated over many time points, which leads to long-term dependencies. In such situations, the gradient can either vanish or explode, affecting the learning process negatively. To circumvent such long-term dependencies, *Long Short-Term Memories* (LSTMs) [84] are leveraged which deal with the problem of observing vanishing or exploding gradients in the learning process.

2.8. Validation Preliminaries

Finally, this section briefly discusses the basic building blocks essential for validation.

2.8.1. Goal-Question-Metric Approach

In the validation chapter of this thesis (see chapter 9), we leverage the so-called *Goal-Question-Metric* (GQM) approach or plan that guides our validation process. The GQM approach was originally introduced by Basili, Caldiera and Rombach [36].

The idea of a GQM plan is to structure the validation or evaluation process by three distinct levels, namely the conceptual level (goals), the operational level (questions) and the quantitative level (metrics). The conceptual level defines the distinct goals, one has to define within the validation or evaluation process.

A goal is formulated for a particular purpose of measurement (e.g. to improve or analyse something), issue (e.g. efficiency or applicability) and object (e.g. software product,

goal	Purpose	Improve
	Issue	the timeliness of
	Object	change request processing
	Viewpoint	from the project manager's viewpoint

Table 2.1.: Example of a formulated goal taken from [36].

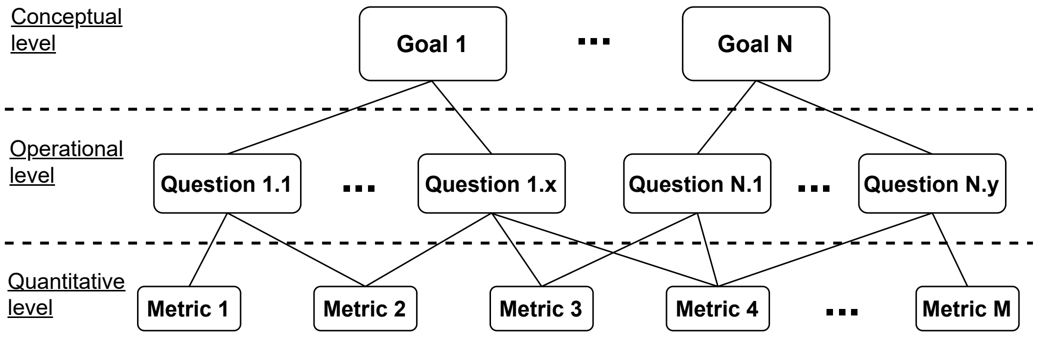


Figure 2.8.: Overview of the GQM approach adapted from [36].

modelling language or process) to be measured and w.r.t. to a certain viewpoint (e.g. from the perspective of a user or software engineer). More specifically, a goal is structured by the very same components, i.e. purpose, issue, object and viewpoint. An example of a formulated goal is shown in Table 2.1.

On the operational level, questions are defined that determine or characterize how the validation is conducted to achieve or assess the goal. Therefore, a goal is associated with a set of questions. Finally, on the quantitative level, metrics are defined. Each question is related to a set of metrics. Metrics define in a quantitative way how measurements can be derived to answer a particular question. An overview of the GQM approach as well as the relationships between the distinct levels is depicted on Figure 2.8.

2.8.2. Validation Levels

Böhme and Reussner [28] define three validation levels to validate analytical metrics. In terms of this work, we assign our validation goals to the validation levels presented below.

An analytical metric is defined as a prediction made for a quality property of a system, e.g. a predictive model. The validation of analytical metrics, however, is more challenging since the predictions must be compared with observations or measurements of actual outcomes. In this context, the authors presented three levels of validation at which an analytical metric can be validated:

- Level I (metric validation): A level I validation refers to the situation where predictions of an analytical metric are compared with observations or measurements. However, Böhme and Reussner [28] point out that this requires the implementation of the analytical metric and the computability of the metric.

- Level II (applicability validation): A level II validation is concerned with the applicability of an analytical metric and prediction approach. Level II validation is about whether the input data can be collected reliably and whether the predictions made can be interpreted meaningfully (e.g. to assess design decisions).
- Level III (benefit validation): A level III validation relates to validating the benefit of an analytical metric or prediction approach as part of a systematic process (e.g. the selection of design options). In such settings, the analytical metric must be compared with competing approaches to validate the benefits. However, level III validation usually involves a lot of effort [28].

2.8.3. Bhattacharyya Distance

In the validation chapter of this thesis, we use the *Bhattacharyya Distance* [22] to measure the similarity of two probability distributions. In order to calculate the Bhattacharyya distance D_B, one has to calculate the Bhattacharyya coefficient $BC(P, P')$. Let P and P' be two probability distributions defined over the random variable X, the Bhattacharyya coefficient $BC(P, P')$ is defined as follows:

$$BC(P, P') = \sum_{x \in Val(X)} \sqrt{P(X = x) \cdot P'(X = x)} \tag{2.18}$$

Based on the Bhattacharyya coefficient, the Bhattacharyya distance can be derived.

$$D_B = -ln(BC(P, P')) \tag{2.19}$$

Note that we formalized the Bhattacharyya coefficient and distance in terms of discrete probability distributions.

3. Related Work

In this chapter, we review related work. We structure related work into three areas. In section 3.1, we first distinguish ourselves from approaches that are dealing with AI-induced uncertainties. In section 3.2, we discuss engineering processes that facilitate quality assurance of AI-enabled systems and review existing taxonomies that classify AI systems. Finally, section 3.3 discusses the whole branch of research analysing self-adaptive systems by either employing MDPs (which also include runtime approaches) or using other model-based approaches for design-time analysis.

3.1. Dealing with AI-induced Uncertainty

In this section, we give an overview of related work that deals either with AI-induced uncertainty at the algorithmic or system level. Especially, on the algorithmic level, there are numerous approaches one can take into account, e.g. approaches that allow identifying vulnerabilities in AI models or proving safety properties. We refer to [86, 58, 195] which provide a broad overview of safety-related algorithmic approaches. In section 3.1.1, we explain how our approach differs from pure algorithmic approaches. In addition, we present several approaches regarding runtime monitoring of AI components as an additional safeguarding mechanism. Afterwards, we present in section 3.1.2 related work of system-level approaches.

3.1.1. Algorithmic Approaches

In this section, we give a brief overview of purely algorithmic approaches. Although all approaches are strongly related to our work (as they aim to deal with AI-specific uncertainty), they differ in that they are purely algorithmic. On the contrary, our approach is based on the architecture or system level and supports software engineers (using models) in the development of reliable AI-enabled software systems. Our approach is rather to be understood as complementary to the approaches at the algorithmic level. On the one hand, algorithmic approaches enable the improvement of AI models by identifying vulnerabilities or data examples for which the AI model does not provide good predictions. However, it is unlikely to obtain an AI model acting perfectly accurately. Therefore, architectural or system-level approaches should be considered as additional safeguards. Such architectural approaches are systematically evaluated using the concepts presented in this work and support software engineers in decision-making. On the other hand, various approaches suggest runtime monitoring, i.e. to detect potential malicious inputs or potential erroneous predictions of AI components. Such approaches differ from our work in that they provide building blocks for architecture- or system-level approaches in which they are embedded. For example, there are architectural patterns that implement measures to deal with potentially unsafe states but require a monitoring procedure to detect such states; our approach enables the evaluation of such architectural patterns.

3.1.1.1. Verification of AI

Great efforts have been made in providing formal guarantees for the behaviour of DNNs (deep neural networks), e.g. robustness properties [13]. In addition, DNNs have been observed to be sensitive to so-called *Adversarial Examples* [71] where inputs (known to produce correct outputs by the DNN) are slightly altered (e.g. by modifying some pixels of the input image) and force the DNN to produce incorrect outputs. Therefore, several verification techniques have been developed to prove robustness or to compute regions of the input space for which one can prove that the DNN produces correct results.

Katz et al. [99] present Reluplex, an SMT (satisfiability modulo theories) solver for verifying DNNs. More specifically, Reluplex allows the verification of certain properties such as robustness properties [13]. Another example for verifying properties in DNNs is provided by the Marabou framework [100] or by the work of Wang et al. [201]. Singh et al. [172] provide an approach to verify the robustness properties of neural networks for more complex perturbations of the input data (e.g. image rotations). The approach allows verifying whether a neural network produces correct predictions for so-called adversarial regions (constructed by considering slight modifications of the original image that are still close to the original image depending on a certain distance norm). Such adversarial regions can also be leveraged at runtime to verify whether new input data are contained in such an adversarial region for which the neural network has been shown to provide correct predictions. Another interesting approach is by Cohen, Rosenfeld and Kolter [50], who use randomised smoothing to generate adversarial robustness of DNNs.

However, the problem with verification techniques is that they lack efficiency because DNNs are large, non-linear and non-convex [99]. Therefore, they tend to be practically inapplicable in situations that require fast reactions, e.g. calculating whether input data belongs to an adversarial region might be too time-consuming in autonomous driving. Therefore, they are more likely to be used at design-time or in offline experiments to enhance the overall robustness of DNNs.

3.1.1.2. Interpretable and Explainable AI

As AI has become more advanced and widely applicable, the challenges it entails have also become apparent. The black-box nature of AI makes the use of AI models in safety-critical applications very difficult due to the lack of comprehensibility and interpretability of their internal behaviour. Therefore, the requirement to understand and interpret the internal behaviour of AI models becomes an interesting branch of research to deal with AI-induced uncertainty.

In terms of interpretable and explainable AI, the survey of Guidotti et al. [76] provides an excellent overview. In their work, they classify black-box explanation approaches into three classes, namely model explanation, outcome explanation and model inspection. Model explanation is about finding an interpretable global predictor for a black-box AI model which generates for each input an explanation. The explanations are used to comprehend the prediction or decision made by the AI model and serve as a basis to determine whether the prediction is correct or not. Outcome explanation is about finding an interpretable local predictor, i.e. for generating explanations for only a subset of the input space. Examples for model explanation are [107, 82]; examples for outcome explanation are [150, 217]. The model inspection problem classifies approaches which are concerned with identifying properties or features of the input space that greatly impact the outcome of an AI model, e.g. a sensitivity analysis. Examples of model inspections provide [179, 171]. However, for a complete overview of all approaches, we refer again to [76].

3.1.1.3. Safe Reinforcement Learning

In supervised and unsupervised learning tasks, a machine learning model learns based on data (either labelled or unlabeled). Reinforcement learning constitutes the third branch into which a machine learning approach can be classified. In contrast to supervised and unsupervised learning, an agent learns how to operate in an environment. Technically speaking, reinforcement learning builds on MDPs, where the main learning task is to learn a strategy π that selects from a set of actions the best possible action by considering its effect on the environment. For example, reinforcement learning is a widely used machine learning approach to motion control in robotics. To learn an optimal policy, the agent must explore possible actions to observe how the environment reacts to them. However, this can cause the agent to enter an unsafe state, which is particularly problematic for safety-critical applications.

Therefore, several works expanded the framework of reinforcement learning by introducing safeguarding mechanisms. For instance, Alshiekh et al.[4] introduce so-called shields which are additionally considered while learning policies. A shield monitors the actions taken by the agent and corrects them whenever safety constraints (expressed in temporal logic) are violated. Moreover, Wachi and Sui [200] propose a safe reinforcement learning approach by considering *Constrained Markov Decision Processes.* In this approach, the agent learns safety constraints by successively expanding a safe state space region. After expansion, the agent learns the policy within the safe region. We refer to [63], which provides an overview of further safe reinforcement learning approaches.

3.1.1.4. Runtime Monitoring of AI components

Runtime Monitoring constitutes another research area for dealing with the uncertain nature of AI components. The basic idea is to enrich an AI-enabled system with additional monitors specifically designed to monitor the properties of an AI component or model. The observed properties can either be used to determine whether the AI component may be making incorrect predictions or to ensure that the individual predictions do not violate certain safety constraints. In principle, some approaches presented earlier can be employed in the context of runtime monitoring. For example, AI model explainers can be leveraged to generate explanations for a prediction. The explanations in turn serve as the basis for checking whether certain safety constraints are maintained.

Salay and Czarnecki [151] propose requirements and types of properties that are important for a partial specification language of AI models. More specifically, one of the main reasons why verification is difficult to apply to AI models is the lack of specification of properties to be verified. Therefore, the authors propose the use of partial specification languages that formalise some types of properties, e.g. pre- and post-conditions or invariance properties. Furthermore, the languages are not only used to specify safety constraints or requirements but also to use formalised specifications for runtime monitoring.

Kang et al. [98] propose so-called model assertions that one can use for runtime monitoring. A model assertion is a function (specifically designed for an AI model to be monitored) that is defined over the input and output space of an AI model. A possible implementation of a model assertion returns a Boolean value indicating that a possible fault might occur. For example, if an AI-based object recognition quickly changes its class in a video.

Another variant of runtime monitoring is proposed by the work of Cheng et al. [46], who describes a runtime monitoring technique based on neuron activation patterns of neural networks. The idea is to construct a runtime monitor based on the trained neural network and the training dataset used. In this process, the training data is fed back to the neural network and (abstract) neural activation patterns are generated and stored in the monitor. At runtime, activation patterns for newly arriving input data are generated and the similarity with the patterns obtained from the training data is compared (using a distance measure). Input data for which the monitor has not stored a similar pattern is potentially malicious and could cause the AI model to behave incorrectly.

Finally, Langford and Cheng [111] describe an approach that allows the learned behaviour of an AI component to be predicted in the presence of uncertainty. Based on such predictive capabilities, one can anticipate potentially erroneous behaviour at runtime and take appropriate countermeasures.

3.1.1.5. Machine Learning Testing

To assure some level of quality of machine learning models, *Machine Learning Testing* has become an interesting research branch. Testing machine learning models, however, poses greater challenges than testing conventional software [213], e.g. the oracle problem

[12]. We refer to Zhang et al. [213] who provides a comprehensive overview of machine learning testing approaches. In the following, however, we enumerate a view testing approaches.

In the work of Thian et al. [186], the authors present the DNN testing tool DeepTest which is specifically designed to test DNN-based self-driving cars. DeepTest leverages neuron coverage to detect input data for which the DNN produces incorrect outputs. More specifically, the tool generates synthetic input data (i.e. synthetic images) based on a set of uncertainties (e.g. raindrops in the image, image blur or various brightness conditions). The generated images are fed back to the DNN, and the neuron coverage is observed (input data which activates only a small amount of neurons of the DNN are likely to cause wrong predictions). In the context of the Udacity self-driving car challenge [192], the DeepTest tool identified several input images for which several DNN-based steering angle prediction models produced wrong predictions. Also in this line of research is the work of [187] which also employs neuron coverage metrics to detect confusion and bias errors in DNN-based visual recognition models for image classification.

Pei, Cao, Yang and Jana propose the DNN testing tool DeepXplore [136] for white-box testing. Similarly to DeepTest, DeepXplore detects input data for which DNNs produce wrong outputs. They make use of neuron coverage to measure the extent to which the internal logic of a DNN has been tested. Moreover, the authors leverage multiple DNNs as cross-referencing oracles to detect incorrect predictions of a single DNN.

Finally, DeepRoad [214]) constitutes another testing tool for DNN-based autonomous driving systems. In contrast to the previously presented testing tools, DeepRoad makes use of a *Generative Adversarial Network* [70] to synthesise driving scenes. This circumvents problems related to synthetic data generated by image transformations, e.g. lack of diversity of driving scenes [214]. Finally, DeepRoad implements an input validation component which detects inconsistent behaviours of the tested DNNs.

3.1.1.6. Using Training Data to Deal with Uncertainty

This section enumerates approaches that make use of training data to (i) identify sub-input spaces for which an AI model may produce incorrect predictions or (ii) to determine whether new input data was generated by the same data generation process as the training data. However, what both have in common is that they identify input data that is not sufficiently represented by the training dataset.

Gu and Easwaran [75] present an approach that partitions the input or feature space of an AI model into several subspaces. Based on the training dataset, the approach identifies those partitions that are not sufficiently represented by training samples. Thus, one can either collect new training data to retrain the AI model or use it at runtime to identify inputs that are included in one of the safety-critical partitions.

Other approaches such as, for example, [1, 81] detect input data which does not belong to the training data distribution of a classifier. Such examples are denoted out-of-distribution inputs which must be detected to prevent potential misclassifications. According to [167], another method to detect out-of-distribution samples provides a family of algorithms that are known as variational inference [26].

3.1.1.7. Safety Assurance of AI Models

In this section, we summarise several methods and approaches for the safety assurance of AI models.

In their survey [161], Schwalbe and Schels enumerate various methods for safety assurance along the lifecycle of a machine learning model. Hereby, the authors discuss distinct safety requirements one must take into consideration during requirements engineering (such as safety-related performance requirements). Moreover, the authors highlight the importance of making well-informed design decisions and enumerate a list of quality criteria and other aspects (such as the inclusion of expert knowledge) software engineers have to account for in the decision-making process. In the end, the authors discuss methods for verifying (e.g. satisfiability modulo theory) and validating (e.g. data validation by using fuzzy testing) AI models.

Burton, Gauerhof and Heinzemann discuss in their work [35] how assurance cases can be used to argue in terms of the safety of machine learning models in the context of autonomous driving. In addition, they present several methods and techniques to substantiate claims made in the assurance case, namely training data coverage, explainability of the learned function, uncertainty calculation, black-box testing and runtime measures. The latter comprises runtime monitoring techniques to observe assumptions made or to check the plausibility of the produced outputs.

Finally, Varshney et al. [195] discuss three strategies for achieving safety in machine learning. The first strategy discusses how machine learning models can be inherently safely designed. According to the authors, an inherently safe designed system excludes potential hazards. In terms of machine learning, such hazards might be the exclusion of features that are not causally related to the outputs of a learned model. The second strategy presents safety reserves. Such safety reserves define safety margins which may, for example, address uncertainty resulting from label noise in classification problems. Finally, the third strategy refers to fail-safe states. Hereby, in situations where a system encounters erroneous behaviour, the system may fail safely. In terms of machine learning, fail-safe states can be realized by using reject options. For example, when the confidence of a prediction made by the machine learning model is not sufficiently high enough, the output is rejected and the system transitions into a fail-safe state, e.g. a human operator takes over control.

3.1.2. System-level Approaches

Now, we give an overview of the approaches that safeguard AI models at the architectural or system level. First, we list approaches that provide architectural knowledge for AI-enabled systems (i.e. architectural patterns, styles or reference architectures). Afterwards, we present model-based approaches similar to the presented approach of this work but different in that we aim to predict the reliability properties of AI-enabled systems. Finally, we present and distinguish ourselves from approaches that make use of self-adaptive systems for safeguarding AI black-box components.

3.1.2.1. Architectural Knowledge for AI-based Systems

There are numerous works which propose various architectural means to deal with AI-induced uncertainties. In the following, we review some of them. Also, we discuss architectural patterns from related domains which are potentially applicable to AI-enabled systems. Note that although our work is about addressing AI-induced uncertainty at the architectural level (i.e. by applying architectural patterns), we do not aim to develop novel architectural patterns but rather provide the means to evaluate existing ones. This forms the main difference between our approach to the approaches presented in this section.

Using architectural means to deal with uncertainty primarily induced by ML (machine learning) components is not new in research. Serban [164], for example, argues that ML-induced uncertainty can be mitigated by architectural patterns. More specifically, Serban argues that using already known architectural patterns from the safety domain (e.g. triple modular redundancy) could be a considerable way to approach uncertainties caused by ML.

In terms of architectural patterns several works apply the well-known *N-Version Programming* pattern to deal with possible false predictions of an AI model, e.g. see [77, 211, 119]. In principle, the idea is to use N distinct AI models for which a newly arriving input is fed into each of the models such that one obtains N outputs. Eventually, the results are used to synthesise a more qualitative output, e.g. by calculating a mean value (in the context of control signal prediction) or by making a majority vote (in terms of classification tasks).

In the context of autonomous driving, Shafaei et al. [167] present four cases to deal with uncertainties related to ML models. For example, one pattern describes the use of variational methods (i.e. variational inference) to filter anomalous inputs. Hereby, the pattern discusses an input checker component (based on variational methods). Whenever the input checker detects potentially malicious inputs, the system transitions into a fail-safe mode. Otherwise, the input is simply forwarded to the ML model.

Biondi et al. [23] propose a novel software architecture for integrating DNN components such that safety, security and time predictability are addressed. The authors suggest the use of hypervisors to isolate the security and safety-critical components. Moreover, the software architecture incorporates redundancy and diversity mechanisms to enhance robustness and fault tolerance. Also, digital twin technologies are discussed for predictive fault detection and fault recovery mechanisms (such as switching the safety-critical components with more conservative components in terms of failure occurrence) are enumerated as well. Finally, a predictable DNN inference engine is discussed for dealing with distinct execution rates of several concurrent DNNs.

Also in that line of research, the work of Cheng, Gulati and Yan [45] outline three architectural approaches for architecting dependable AI-enabled systems (focusing on the autonomous driving domain). The first architectural approach is *Diverse Redundancy* and refers to using diverse algorithms or paradigms. The second approach is *Information Fusion*, i.e. by using several homogeneous or heterogeneous sources and merging them to gain more qualitative information. The third approach discusses *Runtime Monitoring* by using additional monitors to detect potential erroneous behaviour.

Originally, Salay and Czarnecki discussed in [151] a partial specification language for enhanced safety assurance of AI components. However, because the authors also consider the language in combination with runtime monitoring, they discuss three architectural patterns on can take into account when performing runtime monitoring. The first pattern simply describes a fail-safe architecture, i.e. the system enters a fail-safe mode as soon as a monitor signals that the output of an ML component violates a property (specified by their proposed language). The second pattern puts the ML component in a pipeline so that only inputs that cannot be classified in advance are passed to the ML component. Finally, the widely known simplex architecture approach [166] is adopted. In simplex architectures, the idea is to supplement a given algorithm with another, more conservative but verifiable safe algorithm (providing the same functionality) that serves as a backup if the primary algorithm is not trusted for certain outputs. In this case, the ML component is considered the primary algorithm and the specification language serves as the basis to decide when switching to the conservative algorithm. Another example of an approach where simplex architectures are employed for the safety assurance of ML components is provided by Musau et al. [131]. In [152] (also authored by Salay and Czarnecki), the list of architectural patterns is extended by considering three more fault tolerance patterns: Ensemble methods, safety envelope and data harvesting.

In general, there are architectural patterns which are not directly designed to deal with uncertain AI components but are nevertheless applicable in that context. For example, Luo et al. [118] describe a safety channel pattern for automated driving applications which can be potentially adapted to deal with uncertainty in AI components. Also in the domain of safety-critical embedded systems (see [10] for more details) there are plenty of patterns (e.g. voting techniques or recovery block patterns) potentially considerable for safeguarding AI components. Finally, architectural patterns for fault-tolerant systems [52] should be mentioned here as well since they provide great means to handle erroneous behaviour caused by AI components. In fact, we have already discussed approaches that make use of fault-tolerant patterns at the beginning of this section (e.g. [211]).

While not directly related to architectural approaches for dealing with AI-induced uncertainty, approaches (such as [212, 162, 181, 203]) that deal with maintenance issues caused by AI components are still related to this work. Improving the maintenance of AI systems is not only relevant for addressing operational stability [212] but is also of considerable importance when the system structure needs to be revised or redesigned by integrating new (e.g. fault-tolerant) architectural mechanisms. Finally, Lewis, Ozkaya and Xu [115] discuss the challenges and role of software architecture when dealing with model maintenance and evolution.

3.1.2.2. Model-based Approaches

To our knowledge, there is little to no scientific work that uses model-based approaches to analyse AI-enabled software systems at design-time. We found two model-based approaches for design-time analysis.

Dreossi et al. [54] introduce a compositional falsification framework for cyber-physical systems with ML components. The framework identifies inputs for which a modelled cyber-physical system with ML components produces false executions in which a property φ of the system to be falsifying are formalised in signal temporal logic. Specifically, they investigated an *Advanced Emergency Braking System* (AEBS) as a representative of a cyber-physical system that relies on the predictions of an AI-based image classifier for object detection. Simulink models are used to simulate the AEBS (in conjunction with the image classifier) to determine whether the φ property is violated. To identify false executions, two extremes of an image classifier are considered: A perfect classifier (which always makes correct predictions) and an extremely poor classifier (which always makes incorrect predictions). Based on the Simulink simulation, both classifiers are simulated to extract an uncertainty region, i.e. a subspace of the input space for which only the correctness of the classifier's prediction result determines whether a crash is about to happen. From the subspace, inputs are identified (based on sampling techniques) for which the system property φ is violated. Dreossi et al. enable software engineers to identify situations in which the system (and especially the image classifier) behaves unreliable. In contrast to our work, we analyse reliability attributes of modelled software architectures which also guides software engineering during the decision-making process, i.e. finding proper architectural countermeasures to deal with AI-induced uncertainties. Furthermore, we have generalised our approach to self-adaptive systems, which cannot be studied with the falsification framework of Dreossi et al.

Serban, Poll and Visser introduce in their work [163] a method *Modeling Uncertainty During Design* for software architecture evaluation by taking into account the uncertainty of ML components. Their approach enables the evaluation of design decisions (e.g. architectural patterns) to mitigate ML-induced uncertainty. The presented method annotates existing software architectures (or rather the included software components) by the two ML-specific uncertainty types: epistemic and stochastic (or aleatoric) uncertainty. Starting from the annotated components, a BN (Bayesian network) is generated. The annotated components and their annotated uncertainties correspond to the nodes (i.e. the random variables) in the graph of the BN; the connections between the nodes can be interpreted as a kind of control flow that includes all software components to which the uncertainty could propagate. The individual probability distributions of the BN can be investigated by a domain expert or simulation. Afterwards, the overall effect of ML-specific uncertainties can be determined and their impact on particular architectural patterns analysed. Similarly to our approach, the approach of Serban et al. allows design-time evaluation of software architectures. However, we focus on the reliability prediction of the overall system; that is, we do not focus on the individual components that might be affected by ML uncertainties but analyse the system-level effects. Moreover, we do not consider epistemic and stochastic uncertainty as such, but concrete instances or manifestations of uncertainty in the environment that could have an impact on predictive uncertainty. As we will see in later sections, our approach allows us to analyse architectural patterns such as the n-version programming or filtering pattern that either limit the impact of incorrect predictions or contain the occurrence of uncertainty. This is in contrast to the method presented by the authors, where only the sensitivity in terms of uncertainty propagation for specific architecture patterns or styles is analysed. Since our approach is embedded in the Palladio framework, we can also use other simulation and prediction tools to compare design decisions not only in terms of reliability but also in terms of performance, leading to well-informed design decisions.

3.1.2.3. Self-Adaptive Systems to Safeguard AI components

To the best of our knowledge, there is little work using self-adaptive systems as architectural safeguards.

De Lemos and Grzes envisioned the self-adaptive AI approach [114] which discusses the idea of using transparency and interpretation methods of AI to generate explanations that serve as the basis for the self-adaptive system to change the AI model, e.g. via direct manipulation of the model parameters. The authors motivated the idea of self-adaptive AI in terms of dealing with concept drift (i.e. the situation where the distribution of the data changes). While the core idea is the same as what we present in this paper, the authors present a vision rather than an implementation of their approach.

The work of Aniculaesei et al. [7] and Weiss et al. [204] present concepts where self-adaptation is used as the primary means to deal with uncertain AI components. The former presents a holistic approach to engineering dependable autonomous systems which might include AI components. The latter work discusses so-called self-adaptation envelopes which are used to integrate and manage undependable components or subsystems. Both of these works are related to our approach in that self-adaptation is leveraged to deal with unreliable subsystems (which refer in our case to AI black-box components). Overall, however, both approaches are rather considered to either support software engineers in the engineering process (i.e. [7] by proposing a holistic software system engineering approach for dependable autonomous systems) or by providing a concept to integrate undependable self-adaptive systems in safety-critical environments. In contrast, the approach presented in this thesis provides an implemented design-time approach to predict the reliability attributes of AI-enabled software systems.

Another example of a runtime adaptation approach is provided by Zhu et al. [216] that outline approaches where multiple AI-based controllers (for planning tasks in autonomous systems) are employed. The controllers are considered to be diverse, i.e. each controller might be designed by a different team or according to different design methodologies. Due to diversity, some controllers might have advantages (e.g. acting more robust) while having disadvantages (e.g. less efficient). Runtime adaptation is used to switch between the controllers to deal with various system states. A similar approach is described by [128].

3.2. Quality Assurance of AI-enabled Systems

In this section on related work, we review and distinguish ourselves from approaches to quality assurance of AI-enabled systems. Strictly speaking, the approaches listed in section 3.1 are also quality assurance approaches, but directly at the algorithmic or system level. In the following, approaches are discussed (complementary to the approaches from the last sections) that deal with the quality assurance of AI systems by looking at the life cycle of an AI component. Therefore, in section 3.2.1 we provide an overview of approaches that propose or discuss engineering processes around the life cycle of an AI component. Afterwards, we discuss in section 3.2.2 existing classifications for AI-enabled systems and how they are different from our classes of architectural dependability assurance.

3.2.1. Engineering Processes

There are a lot of works that describe how engineering approaches must be adapted to engineer AI or (more specifically) ML systems. Similarly, our assurance classes provide support for assessing an AI-enabled system and its domain (e.g. operating environment). Thus, they also guide the development of the system. Nonetheless, the approaches are rather suggesting further methods or activities that must be considered during all phases of development (e.g. requirement, training or deployment phase). In contrast, our classes give a first intuition of the system and the level of assurance that can be achieved. Based on the classification into one of the classes, a software engineer can take appropriate actions by considering the methods and activities proposed in the work, which we summarise below. Also, we refer to [67] which conducted a systematic literature review on the state-of-the-art of software engineering research for engineering ML-based systems.

In terms of addressing safety issues during the ML lifecycle, Pereira and Thomas [137] discuss safety hazards that can occur during the lifecycle of ML-based cyber-physical systems that potentially impact safety, e.g. incorrect objective function definition during requirements elicitation. Also in this context, the work of Santhanam et al. [154] introduce and discuss the notion of AI engineering which is about building reliable deep learning-based software systems. In particular, the authors discuss the impact of deep learning components on the traditional software lifecycle and highlight challenges that need to be addressed such as the requirement to measure the correctness of a deep learning model across its lifecycle. Ashmore, Calinescu and Paterson [11] provide a survey which discusses state-of-the-art methods that provide proper evidence for assuring ML models at distinct stages of the ML lifecycle. Hereby, the authors considered the stages *Data Management*, *Model Learning*, *Model Verification* and *Model Deployment*. For each phase, the corresponding activities are discussed (e.g. data management includes activities such as data collection and pre-processing), assurance desiderata and the methods to achieve assurance.

Besides adapting traditional software development processes in terms of AI-specific activities to assure safety, some approaches discuss general processes for engineering AI applications. For example, Amershi et al. [5] describe a study for building AI applications based on the experience of several Microsoft software teams. Moreover, they present a nine-step workflow process for developing AI applications which is integrated into agile-like software engineering processes. Also, the authors extracted best practices one can account for when developing AI systems. Finally, Hesenius et al. [83] represent a further work which proposes a software engineering process for ML systems (or data-driven applications).

In addition to approaches that focus purely on the engineering process and the lifecycle of AI-enabled systems, some approaches focus on specific phases within the process, such as requirements engineering [85, 197, 144] or testing [32].

3.2.2. Classifying AI-enabled Systems

In this section, we discuss scientific works (similarly to our classes of dependability assurances) that classify AI-enabled systems w.r.t. some classification structure or taxonomy. To our knowledge, however, no work classifies AI-enabled systems according to the extent to which assurances can be made either at design-time or at runtime. Nonetheless, we discuss some works that propose classification structures, but for a different purpose.

In [199], a taxonomy is established that classifies ML in terms of the types of knowledge used to train the respective ML models. Therefore, the authors introduce the term *Informed Machine Learning*, which refers to the idea of incorporating knowledge gained (e.g. from domain experts) into the training process. For example, knowledge graphs can be used to provide certain relationships between concepts in the domain to be learned. The proposed taxonomy classifies ML approaches based on three categories: The type of knowledge integrated (e.g. domain expert or process flows), the representation of the knowledge (e.g. rules, knowledge graphs or differential equations) and the location where the knowledge is integrated (e.g. training data or hypothesis space). In contrast, our classes of dependability assurance are rather designed to classify ML systems regarding assurance levels. Moreover, the taxonomy presented by the authors aims to classify research activities.

Feldt et al. [59] present the *AI in Software Engineering Application Levels* (AI-SEAL) taxonomy that classifies applications w.r.t. three dimensions. The first dimension is the *Point of Application* and refers to the point in time or location the AI technology is applied, i.e. during execution (runtime), at the process level (during software engineering process) or directly in the product. The second dimension refers to the *Type of AI*, e.g. Bayesian models. The last dimension or facet corresponds to the level of automation, i.e. the extent of human intervention. The authors argue that engineers or software companies can analyse the associated risks and opportunities when applying AI based on the taxonomy. Although this is fairly similar to our classification structure, we reason about the assurance level one can make when engineering AI-enabled systems. Furthermore, we aim to support software engineers to develop reliable systems with AI components.

3.3. Analysing Self-Adaptive Systems

In this section, we provide an overview and distinguish ourselves from related work that (just like us) uses Markov models to analyse self-adaptive systems in terms of decision-making. Note that although we will describe extensively how MDPs are instantiated in the domain of self-adaptive systems, we do not aim to provide a formalism describing the fundamental notion of self-adaptivity (e.g. Petrovska et al. [138]), but use MDPs to predict the quality of an adaptation strategy. We conclude this section by reviewing model-based approaches for analysing self-adaptive systems at design-time and discuss how they differ from our approach.

3.3.1. Using Markov Models for Decision-Making

The use of Markov models in terms of self-adaptive systems is a widely used method. For example, Moreno et al. [126] employ MDPs to determine the best possible adaptation or adaptation tactic from a given set of options. Hereby, the interaction of the environment and the system is translated into an MDP such that different adaptations can be evaluated by applying (e.g.) probabilistic model checking to the MDP. Thus, the best possible adaptation can be determined. Also in this line of research, Camilli, Mirandola and Scandurra [40] employ MDPs to capture self-adaptive systems and their uncertainties mainly induced by the environment and verify whether the system maintains an acceptable behaviour. In this work, we use MDPs as well to simulate the interaction between the environment and the system. However, we conduct a full simulation of the MDP to evaluate entire adaptation strategies (as opposed to analysing individual adaptation options). Moreover, the above approaches are applicable only at runtime, whereas our approach aims to analyse adaptation strategies at design-time. Also, Elrakaiby et al. [55] make use MDPs to formalise self-adaptive systems for the sake of optimisation. The authors provide a framework for model-based (and requirements-driven) synthesis of optimal adaptation strategies for autonomous systems at design-time. Besides the difference that we evaluate adaptation strategies and do not optimise them, the authors focus on behavioural optimisation (i.e. on functional requirements), which is in contrast to our approach that deals exclusively with non-functional requirements.

DTMCs (discrete-time Markov chains) represent Markov models that are also frequently used to model self-adaptive systems and their stochastic dynamics, e.g. [66, 38, 60, 37, 61]. In [38], for example, the authors use DTMCs to describe the impact of adaptations. Based on the DTMC, the impact of adaptation is analysed and the best possible adaptation is selected. Further, Filieri et al. [60] use DTMCs to model different behavioural variants of a self-adaptive system and use control theory approaches to maintain the reliability properties. Equivalently, Cámara and de Lemos [37] employ DTMCs to model self-adaptive systems and to check resilience properties. The resilience properties are specified by using probabilistic computation tree logic. In the last step, the authors apply probabilistic model checking to verify whether the properties are satisfied. Finally, Franco et al. [61] translate individual system configurations into DTMCs to predict reliability properties. At runtime, the prediction results are used to find optimal adaptations to which the current configuration of the system can transition. As with MDP-based approaches, the same reasoning applies to the distinction with our work: All approaches are used at runtime to support decision-making (e.g. selecting the best possible adaptation) or to verify certain properties when transitioning to another state. In this work, however, we focus entirely on design-time analysis, i.e. we enable the evaluation of design decisions before a single line of code of the system is implemented. Moreover, we evaluate entire adaptation strategies.

3.3.2. Model-based Analysis of Self-Adaptive Systems

In this section, we review related work that is either purely model-based or heavily build upon model-based techniques.

3.3.2.1. Architecture-based Self-Adaptation

Architecture-based self-adaptation [64] refers to a research area in which models (abstracting the managed system) are used at runtime to evaluate changes made to the managed system using model-based analysis techniques. In such settings, abstract models of the system are complemented by data monitored at runtime. Afterwards, analysis techniques are applied to the complemented models in order to predict or analyse system properties such as the response time. Based on the predicted properties, adaptations can be triggered, or they are used to determine the effects of some adaptations. In this section, we give a brief overview of some approaches using architecture-based self-adaptation. However, the difference between architecture-based self-adaptation approaches and our approach is always the same. While architecture-based self-adaptation uses models at runtime (for decision-making), we use models at design-time for the analysis of self-adaptive systems (e.g. the evaluation design decisions).

Cámara et al. [38] use DTMCs to abstract the managed system and to predict quality attributes. The prediction results are used to determine at runtime the effects of particular adaptations; that is, for a set of adaptations, the individual impacts are evaluated such that the best possible adaptation is selected.

In [87], the authors present a model-based approach using the *Descartes Modeling Language* (DML). DML is an architecture-level modelling language for online performance and resource management in self-adaptive systems. The approach allows the prediction of performance attributes of the system such that adaptation can be planned proactively w.r.t. the predicted performance attributes.

Finally, Weyns and Iftikhar [208] present an approach for model-based simulation at runtime. Hereby, stochastic timed automata are used to abstract the managed system and the environment. At runtime, the models are first complemented with runtime data and then simulated. Based on the simulation results, the adaptation (from a set of adaptations) that best meets the quality objectives can be determined.

3.3.2.2. Formal Verification of Functional Correctness

Model-based approaches are also leveraged in terms of showing the correctness of the adaptation logic itself. In this context, MAPE-K-based self-adaptive systems are analysed (based on models such as state machines) to verify the correctness of the adaptation behaviour, e.g. [8, 91]. Although such approaches aim to ensure the adaptation behaviour at design-time, our model-based approach is concerned with evaluating the quality or effectiveness of a strategy. That is to say, our approach supports software engineers during the design of an adaptation strategy, i.e. to evaluate distinct adaptation strategies or design decisions within a strategy family. Once an appropriate adaptation strategy has been identified and designed (in terms of the different quality objectives), the approaches to verify the correctness of the strategy can be applied afterwards.

3.3.2.3. Model-based Testing

Model-based testing [193] is a widely used approach to generate test cases or input data for a system under test. In the context of self-adaptive systems, there are also several approaches which make use of models to generate test cases to validate the adaptation logic, e.g. [141, 9]. For example, Arcaini et al. [8] uses a domain-specific modelling language called *MAPE Specification Language* and abstract state machines (to represent the adaptation logic) to generate test cases for which the resulting MAPE-K feedback loop implementation can be tested.

In contrast to our approach, however, we do not aim to test adaptation logic, but to evaluate the quality of an adaptation strategy w.r.t. various quality objectives. More specifically, we address the uncertainty *Parameter over time*, which is arguably difficult to tackle by using model-based testing. We also aim to compare distinct strategies or design decisions within families of strategies. In summary, our approach can be seen as a starting point for developing an appropriate strategy. Afterwards, model-based testing can be used to test the adaptation logic and avoid implementation errors.

3.3.2.4. Scenario-based Analysis

In this section, we review several approaches which we consider to be scenario-based, i.e. the adaptation logic or adaptation strategy is explicitly validated or tested against certain scenarios. However, scenario-based analysis is not suitable for evaluating a general quality measure of a strategy; this forms the main difference to our approach. We will see in chapter 4 that it is necessary to analyse the trajectory space of a self-adaptive system to determine the quality of its adaptation strategy. Scenario-based approaches focus only on evaluating specific trajectories of the space. For instance, let π, π' be two strategies, scenario-based analysis evaluates $\pi > \pi'$ based on a single (or a few) trajectories (or scenarios). However, it does not ensure the general case, i.e. whether π outperforms π' when considering all trajectories. Finally, scenario-based analysis neglects the uncertainty *Parameter over time.*

An example of a scenario-based approach is the performance simulator called SimuLizar [16, 15]. SimuLizar enables the simulation of performance attributes of self-adaptive systems by simulating distinct workload scenarios. More specifically, by using an approach called *Usage Evolution* [31], one can model the evolution of the workload over time which is used as a foundation to analyse the adaptive behaviour (represented by model transformations) w.r.t. performance attributes. There are also extensions for SimuLizar that addresses the consideration of other quality attributes, e.g. energy efficiency [178].

Another example of a scenario-based analysis approach is *SLAstic.SIM* [121] for performance simulation of reconfigurable component-based software systems. Similarly to SimuLizar, SLAstic.SIM uses PCM to represent the managed system. In addition, the simulation is driven by workload traces (either recorded or generated) that represent distinct scenarios against which the adaptation logic is evaluated.

3.3.2.5. Evaluating Adaptation Strategies

In the following, we review approaches similar to our approach to enable the evaluation of adaptation strategies at design-time.

Berardinelli et al. [20] provide an approach which models the context of an adaptive system by using continuous-time Markov chains. Based on the continuous-time Markov chain it can be determined how well an adaptive system satisfies certain quality attributes. In contrast to our approach, however, Berardinelli et al. do not focus on MAPE-K-based self-adaptive systems. Moreover, since we make use of MDPs, we can encode multiple quality attributes within a reward; thus, we can compare strategies w.r.t. various quality attributes.

Grassi, Mirandola and Sabetta [72] discuss a model-based approach of performability analysis for dynamically reconfigurable component-based systems. Just like our approach, the work is applicable at design-time to evaluate dynamically acting systems. However, there are several differences. First, the approach of Grassi et al. is domain-specific, i.e. it deals with the analysis of performability properties. Second, although the approach enables evaluating dynamically reconfigurable systems, they do not focus on MAPE-K-based self-adaptive systems (which is contrary to our work). Thus, they do not analyse what we define as an adaptation strategy. Third, the approach of Grassi et al. requires that the state space (or all possible system configurations) can be completely unfolded. However, due to state space explosion, this is not always possible and addressed in our approach by using Monte Carlo methods.

Cámara et al. [39] describe an approach for offline (i.e. design-time) synthesis of adaptation strategy repertoires w.r.t. a utility profile (system qualities). More specifically, the authors use a discrete abstraction of the state space, MDPs and probabilistic model checking techniques to synthesise a repertoire of adaptation strategies used at runtime. In particular, the offline synthesis identifies those strategies that are more suitable for certain regions of the state space. Based on the offline analysis, the runtime overhead of the online synthesis process is eliminated and near-optimal solutions are provided. To our understanding, however, the term adaptation strategy is treated differently by Cámara et al. More precisely, they define an adaptation strategy as a set of tactics representing a primitive or atomic adaptation step within a strategy. As we will see later, the term adaptation strategy is defined differently in this thesis. In our definition, an adaptation strategy includes all activities that influence the decision to select a particular action in a given state (including activities of all MAPE phases). That is, we evaluate the adaptation logic more comprehensively. Furthermore, our main intention is to evaluate adaptation strategies in order to support software engineers in decision-making, i.e. comparing strategies or design decisions within strategy families.

3.3.2.6. Environmental Modelling

In terms of environment modelling, there are two works [104, 169] which conducted literature research on modelling the environment of self-adaptive systems. According to the results, various modelling approaches capture the environment in which self-adaptive systems operate, e.g. state machines, DTMCs, MDPs, and UML (unified modelling language) class diagrams. The selection of the concrete model type highly depends on the purpose of the environment model. The environment models, for example, reviewed in [104] are considered from the perspective of requirements engineering. The environment models discussed in [169] are mainly intended to support runtime decision-making or for model-based testing, which is different from what we want to achieve, i.e. design-time evaluation of adaptation strategies.

When reviewing the various model-based approaches for analysing self-adaptive systems, we also noticed that different representations have been used to capture the environment, ranging from stochastic timed automata to Markov models to traces of recorded data. In virtually every approach, the purpose of the environment model is distinct from our approach. Moreover, most design-time approaches are applicable in a domain-specific way. Thus, we can barely compare our environmental modelling approach with other approaches.

However, we found at least one approach which supports desing-time analysis by using an environment model representing the context of an adaptive system domain independently. Berardinelli et al. [20] present a state machine-based *Context Evolution Model* which captures the context of an adaptive system. More specifically, the authors define so-called *Context Attributes* (CA) that represent what we consider to be an environmental variable. Each CA consists of a finite set of values and is described by an individual state machine. A set of CA is transformed into an overall state machine by composing the individual

CAs. Interestingly, a context evolution model can be instantiated domain independently (just as our environmental modelling approach as we will see in chapter 5). However, the algorithm composing the single CAs suffers from scalability and memory limitations because the number of composed states grows exponentially. We circumvent this problem by compactly encoding the state space by using Bayesian models. Finally, we argue that the modelling of complex state spaces and complex stochastic dependencies is rather hard to conduct with a state machine-based model because for each CA a state machine must be described and also the dependencies between the distinct state machines. In contrast, our approach allows the modelling of state spaces flexibly and compactly by using discrete Bayesian networks. Again, we defer the discussion to chapter 5.

Part III.

Design-time Evaluation of Self-Adaptive System

4. The Dynamics of Self-Adaptive Systems: A Theoretical Perspective

We consider self-adaptive systems as architectural safeguards that we aim to evaluate at design-time. For design-time analysis, however, an analytical model is required that abstracts the behaviour of a real system and that provides the fundamental means to predict quality attributes. In this chapter, we discuss the theoretical framework or model that underlies the approaches of chapter 6 and section 7.2 and that forms the basis for evaluating adaptation strategies of self-adaptive systems at design-time. More specifically, we formalise the stochastic dynamics of self-adaptive systems. Stochastic dynamics refer to the various states that a self-adaptive system can potentially transition to over time. We consider the process of how a self-adaptive system moves through the state space as a stochastic process which is induced by two components, namely the *Environmental Dynamics* and the *Deterministic Adaptation Process*. Environmental dynamics refers to the stochastic evolution of the environment that must be taken into account by a self-adaptive system to maintain quality objectives. The deterministic adaptation process describes adaptations of system configurations in response to changes in the environment. Based on these concepts, we map the formal elements of a self-adaptive system into the domain of MDPs (Markov decision processes) which we consider as the underlying analytical model of self-adaptive systems. We use MDPs to formulate the challenges and problems arising when engineering self-adaptive systems from the perspective of a software engineer.

Using MDPs as the theoretical foundation has several advantages:

- MDPs are prevalent models in the self-adaptive system community and have proven to be successful as theoretical foundation to capture and analyse self-adaptive systems (e.g. [6, 127, 215]).
- Using MDPs as an underlying framework shows the formal semantics of our presented approach.
- Many other mathematical methods (e.g. dynamic programming, reinforcement learning [180] or stochastic stability analyses [123]) build upon MDPs and offer approaches to deal with the challenges this work addresses. For instance, we discuss in chapter 6 how we make use of dynamic programming to evaluate and assess the quality of an adaptation strategy.

This chapter is structured as follows: Section 4.1 introduces the environmental dynamics. Section 4.2 describes the deterministic adaptation process. The concepts presented in the latter two sections are brought together in section 4.3 to formalise self-adaptive systems as MDPs. Having established the formal apparatus of self-adaptive systems, we formalise in section 4.4 what we consider to be the *Engineering Problem* concerning self-adaptive systems, namely the design of an adaptation strategy for maintaining various quality objectives of a system. Finally, we discuss made assumptions in section 4.5 and summarise the chapter in section 4.6.

4.1. Environmental Dynamics

Self-adaptive systems are expected to maintain quality objectives under changing conditions or *Uncertainties* [206, p. 1]. Uncertainties are an important concept, as they are the source of adaptation. In literature, however, there are various definitions and understandings. For instance, Weyns [206, p. 1] enumerates changes in the operational environment, dynamic resource availability and variations of user goals as possible uncertain conditions. Salehie and Tahvildari [153] consider changes in the *Self* and *Context* of a software system as the main reason for adaptation. The term *Self* refers to the whole body of the software and *Context* includes everything in the operational environment that affects system properties and behaviour. As the last example, Oreizy et al. [134] consider the operating environment as the source of adaptation in which the operating environment is considered as anything that can be observed by the system (e.g. user input), sensor data or hardware devices. We agree with all of these definitions. In this thesis, however, we generalise the concept of changing conditions or uncertainties (w.r.t. the definitions) and define it from a quality-oriented perspective.

More specifically, we denote the source of adaptation as *Environment*. Before defining the environment more formally, we introduce *Environmental States* that make up the environment.

Definition 17 (Environmental State). *An environmental state encompasses all variables or factors whose behaviour cannot be controlled directly but have an impact on the quality objectives of a software system. More formally, an environmental state $E := (e_1, ..., e_n)$ is a tuple consisting of instances of the aforementioned variables.*

To limit the effects of the state space explosion problem, we assume that each $e_i \in E$ is discrete. Based on the definition of environmental states, the environment $\mathcal{E}$ is defined as follows:

Definition 18 (Environment). *The environment $\mathcal{E} := \{E_1, ..., E_m\}$ is a set of discrete environmental states. More specifically, $\mathcal{E}$ is spanned by all the variable realizations or instances of each environmental state E_i.*

The definition of discrete environmental states E follows from the assumption that each E consists of discrete variables.

Note how definitions 17 and 18 abstract from the definitions enumerated at the beginning of this section. An environmental state E includes all variables $e_1, ..., e_n$ that affect quality objectives which subsumes (depending on the domain) variables like resource availability/failures or harsh weather conditions. More specifically, no distinction is made between intrinsic (e.g. resource availability) or extrinsic (e.g. harsh weather conditions) variables, as only their effect on quality objectives is of relevance. The main concern of this thesis is how quality objectives (w.r.t. the quality requirements) are affected in the presence of certain environmental states and how self-adaptation compensates for these effects. The concrete classification or meta-information of the environmental state itself is therefore of secondary importance but only their effect on quality objectives matters. Self-adaptation is highly related to quality requirements and meeting them is the primary trigger for adaptation [153].

The distinct states of the environment evolve; that is, for each time instance t, we observe the environment to be in a particular state E_t. We denote the evolution of the environmental states as *Environmental Dynamics* which we consider as stochastic process $(X_{\mathcal{E}_t})_{t \in I\!N}$. In addition, we assume that the Markov assumption (see section 2.4.1) applies to $(X_{\mathcal{E}_t})_{t \in I\!N}$.

Definition 19 (Environmental Dynamics)**.** *The environmental dynamics are a stochastic process $(X_{\mathcal{E}_t})_{t \in I\!N}$ for which the Markov assumption holds. More precisely, the environmental dynamics are described as stationary Discrete-Time Markov Chain $(\mathcal{E}, \mathcal{E}_0, t_{\mathcal{E}})$ where*

- *$\mathcal{E}$ corresponds to the set of environmental states.*
- *$\mathcal{E}_0$ corresponds to the set of initial environmental states: $\mathcal{E}_0 \subseteq \mathcal{E}$*
- *$t_{\mathcal{E}} : \mathcal{E} \times \mathcal{E} \longrightarrow [0, 1]$ corresponds to the transition function that evaluates the probability to transition to state E_i given E_j, i.e. $Pr(X_{\mathcal{E}_{t+1}} = E_i \mid X_{\mathcal{E}_t} = E_j)$.*

We consider the stochastic process of environmental dynamics to be discrete (which follows from the definition of discrete environmental states).

The use of DTMCs or Markov models is an accepted and widely used approach in the self-adaptive system community. For example, many works (e.g. [127, 55, 37]) use Markov models to capture the stochastic nature of self-adaptive systems. In contrast to our work, we generalise the concept of the environment and consider it as the source or trigger of adaptation. The dynamics of the environment (see definition 19) is represented by a DTMC and is considered the main component responsible for the stochastic behaviour of self-adaptive systems.

4.2. The Deterministic Adaptation Process

In the last section, the concept of the environment and its dynamics was introduced. Whenever the environment transitions to a state that can no longer be handled by the current configuration of the software system (due to violations of the quality objectives), an adaptation is triggered. The goal is to adapt the system to compensate for changes in the environment to maintain quality objectives. The process of adapting the system from a configuration C_i to C_j is called *Adaptation Process* and is explained in more detail in the following section.

Before formally defining the adaptation process, we introduce basic terminology. We consider adaptation from an architecture-driven perspective. That is to say, adaptations and system configurations are described at the architectural level.

Definition 20 (Architectural Configuration). *An architectural configuration C includes all structural, behavioural and deployment-specific elements as well as their relationships to describe the software architecture (according to definition 5) of the system at a given time instance.*

We intentionally do not specify the concrete structure of an architectural configuration C to emphasise that no assumptions are made and to maintain generality. It is only required that the effect of adaptations to configurations are represented in C. Similarly to the definition of system configurations, adaptations are also defined from an architectural perspective.

Definition 21 (Architectural Adaptation). *An architectural adaptation $\delta \in \Delta$ (where Δ refers to the set of available adaptations) is applied to an architectural configuration of a system to change its structure or behaviour.*

The set of available adaptations Δ includes at least one adaptation $\delta_\emptyset$ which we denote as the *Empty Adaptation*. The empty adaptation applied to a configuration does not change the configuration and indicates that the self-adaptive system has not taken any action at all, i.e. applying $\delta_\emptyset$ to an architectural configuration C results in C again. For the sake of completeness, the architectural configuration space is defined as follows:

Definition 22 (Architectural Configuration Space). *The architectural configuration space $\mathcal{C}$ consists of all architectural configurations that are reachable from an initial architectural configuration C_0 by applying a sequence of adaptations $(\delta_1, ..., \delta_n)$.*

Based on the previous definitions, the adaptation process can be defined. The adaptation process refers to the transition of an architectural configuration C_i to C_j by applying an architectural adaptation δ.

Definition 23 (The Deterministic Adaptation Process). *The deterministic adaptation process is described by the function $\phi : \mathcal{C} \times \Delta \rightarrow \mathcal{C}$ that uniquely maps an architectural configuration C_i and adaptation δ to an architectural configuration C_j, i.e. $\phi(C_i, \delta) = C_j$.*

Based on the definition of the deterministic adaptation process, we now catch up with the formalisation of the empty adaptation property introduced earlier.

Property 1. *The empty adaptation $\delta_\emptyset$ applied to any architectural configuration C results always in C:*

$$\forall C \in \mathcal{C} : \phi(C, \delta_\emptyset) = C \tag{4.1}$$

Additionally, we require that ϕ satisfies the following property:

Property 2. *For any pair of architectural configurations C and C' for which there exist an adaptation δ (i.e. $\phi(C, \delta) = C'$), the adaptation δ is unique:*

$$\forall \delta, \delta' \in \Delta : \phi(C, \delta) = \phi(C, \delta') = C' \Leftrightarrow \delta = \delta' \tag{4.2}$$

Basically, the property states that an adaptation δ is derived uniquely from C and C', if $\phi(C, \delta) = C'$ applies. The properties 1 and 2 play an essential role in section 4.3.2, which is why we have formalised them explicitly.

The mechanism governing the adaptation process is the adaptation strategy. The strategy implements the decision logic that determines which adaptation should be executed. The selected adaptation is passed to the adaptation process to apply the adaptation. We assume that the adaptation strategy does not select adaptations leading to invalid architectural configurations. For example, consider adaptations adjusting parameters by adding values or incrementing the parameter itself. When the parameter reaches the maximum value, increasing the value again will result in an invalid configuration. Ensuring the correctness of MAPE-K-based self-adaptive systems can be achieved by using approaches such as [8, 91]. We will not formally introduce here what we consider an adaptation strategy; however, this is made up for in chapter 6. For the remainder of this chapter, it is sufficient to think of an adaptation strategy as a function that determines the next adaptation given the current state.

4.3. Considering Self-Adaptive Systems as Stochastic Processes

As discussed at the beginning of this chapter, we consider the dynamics of a self-adaptive system as a stochastic process induced by two components, namely the environmental dynamics and the deterministic adaptation process. Both were introduced in section 4.1 and 4.2, respectively. In this section, we discuss how environmental dynamics and the deterministic adaptation process are correlated and mapped onto MDPs (Markov decision processes).

4.3.1. Mapping Self-Adaptive Systems to Markov Decision Processes

In this section, we describe how the mathematical framework of MDPs is instantiated in the domain of self-adaptive systems. For this, the elements that make up a self-adaptive system (i.e. the self-adaptive system state space, adaptation space, self-adaptive system dynamics, quality objectives and adaptation strategy) are mapped to the basic elements of an MDP, namely the set of states S, set of actions A, transition function t and reward function r: $\lambda := (S, A, t, r)$ (see section 2.4.2). Before we define the mapping, we need to introduce the concepts of *Self-Adaptive System State* and *Self-Adaptive System State Space.*

Whenever the environment transitions to a state that cannot be handled by the current system configuration, an adaptation is triggered to adapt the system accordingly. As described in section 4.1, we assume that environmental states are discrete.

Definition 24 (Self-Adaptive System State)**.** *A self-adaptive system state S consists of an architectural configuration $C \in \mathcal{C}$ and an environmental state $E \in \mathcal{E}$, described by the tuple: $S := (C, E)$.*

The combination of an environmental state and architectural configuration defines a self-adaptive system state. The state changes over time, e.g. when an adaptation is made or the environment transitions to a different state. For completeness, the self-adaptive system state space is defined as follows:

Definition 25 (Self-Adaptive System State Space)**.** *The self-adaptive system state space $\mathcal{S}$ is a set that encompasses all self-adaptive system states, i.e. $\mathcal{S} := \mathcal{C} \times \mathcal{E}$.*

Based on definitions 24 and 25, the mapping of the elements of a self-adaptive system to the elements of an MDP can now be discussed.

As described at the beginning of this section, an MDP comprises four elements captured by the tuple $\lambda := (S, A, t, r)$. In the remainder of this section, we refer to the set of states S of an MDP as S_λ to distinguish the set from a self-adaptive system state $S := (C, E)$. The most trivial mappings refer to the self-adaptive system space $\mathcal{S}$ which represents S_λ and the adaptation space Δ which corresponds to the set of actions A in the context of MDPs.

In MDPs, the transition function t determines how states transition over time, i.e. it captures the dynamics. Therefore, we consider $t_\mathcal{S}$ as an instantiated version of t that captures the transition function or stochastic dynamics of a self-adaptive system. More precisely, we define the function $t_\mathcal{S} : \mathcal{S} \times \Delta \times \mathcal{S} \to [0, 1]$ as the equivalent concept to t from MDPs. Equally, to t, $t_\mathcal{S}$ determines the probability distribution to transition to state S_j given the present state S_i and adaptation δ, i.e. $t_\mathcal{S} = P(X_{\mathcal{S}_{t+1}} \mid X_{\mathcal{S}_t}, X_{\Delta_t})$, $(S_j, S_i, \delta) \mapsto Pr(X_{\mathcal{S}_{t+1}} = S_j \mid X_{\mathcal{S}_t} = S_i, X_{\Delta_t} = \delta)$. Additionally, $t_\mathcal{S}$ specifies the correlation of the environmental dynamics and the adaptation process. More specifically, let $(X_{\mathcal{S}_t})_{t \in \mathbb{N}}$ be a stochastic process (where the Markov assumption applies) describing the stochastic dynamics of a self-adaptive system. $t_{\mathcal{S}}$ encodes all information necessary to determine all possible state sequences of $(X_{\mathcal{S}_t})_{t\in I\!N}$:

$$X_{\mathcal{S}_0} = S_0 \to \cdots \to X_{\mathcal{S}_t} = S_i \overset{t_{\mathcal{S}}(S_i,\delta,S_j)}{\longrightarrow} X_{\mathcal{S}_{t+1}} = S_j \to \cdots \to X_{\mathcal{S}_\infty} = S_k \tag{4.3}$$

$$\cdots \to X_{\mathcal{S}_t} = S_i := (C,E) \overset{t_{\mathcal{S}}(S_i,\delta,S_j)}{\longrightarrow} X_{\mathcal{S}_{t+1}} = S_j := (C',E') \to \cdots \tag{4.4}$$

Hereby, $X_{\mathcal{S}_t} = S_i \overset{t_{\mathcal{S}}(S_i,\delta,S_j)}{\longrightarrow} X_{\mathcal{S}_{t+1}} = S_j$ denotes a specific transition from state S_i and δ to S_j w.r.t. $t_{\mathcal{S}}(S_i,\delta,S_j)$. As shown in sequence 4.4, a state $S := (C,E)$ consists of an architectural configuration C and environmental state E. Both components underlie a change process: the deterministic adaptation process and the stochastic environmental dynamics. Intuitively, one would think that the adaptation process is mainly driven by the environmental dynamics; that is, whenever the environmental dynamics transitions from state E at time t to state E' at time $t+1$ and the current configuration C is not able to satisfy quality objectives in state E', a new configuration is observed at time $t+2$. However, this is not necessarily the case because the environmental dynamics may also change during the adaptation process (e.g. in the sequence from $t+1$ to $t+2$). In principle, there are four ways in which $t_{\mathcal{S}}$ can transition to a particular self-adaptive system state which are mainly due to environmental changes, changes in architectural configuration, both or no changes. More formally, let $S_i := (C,E)$ be a state and $S_j := (C',E')$ be the state after the transition w.r.t. $t_{\mathcal{S}}$, the four possible transitions are as follows:

$$S_j = \begin{cases} (\phi(C,\delta),E), & \delta \neq \delta_\emptyset \land E = E' \\ (C,E), & \delta = \delta_\emptyset \land E = E' \\ (\phi(C,\delta),E'), & \delta \neq \delta_\emptyset \land E \neq E' \\ (C,E'), & \delta = \delta_\emptyset \land E \neq E' \end{cases} \tag{4.5}$$

Recall from section 2.4.2 function r of an MDP referring to the reward function that evaluates the decision of selecting an action in a given state considering the state after the transition. In self-adaptive systems, the reward function is represented by the quality objectives or utility functions that must be maintained by adaptation strategies. Therefore, the reward function $r_{\mathcal{S}} : \mathcal{S} \times \Delta \times \mathcal{S} \to I\!R$ in self-adaptive systems reflects exactly these quality objectives. The quality objectives (and preferences of particular quality attributes, if any) are encoded in the resulting rewards generated for each decision that a self-adaptive system makes (i.e. by applying adaptations). Although in later chapters implementations are presented, we do not focus on the best possible way to implement a reward function. While this is an important topic, it is highly domain and application-dependent. Note that the reward function $r_{\mathcal{S}}$ is not limited to represent solely quality objectives. It is also possible to encode other qualities in the reward function (e.g. stability properties). For example, each time an adaptation is triggered, a negative reward can be added to the resulting reward to punish strategies that frequently adapt the system. In this thesis, however, we solely focus on reward functions that reflect quality objectives.

Finally, the last concept of MDPs that must be mapped to a corresponding concept in self-adaptive systems is the policy π. Recall that a policy π determines the action to be taken in a given state. In self-adaptive systems, the policy π is represented by the adaptation strategy that decides whether an adaptation is triggered or not and governs the adaptation process. We are not going into more detail in this section as the concept of adaptation strategies and their role in MDPs is discussed in chapter 6.

As a last remark, note that the definitions of the reward function $r_{\mathcal{S}}$ and policy π (i.e. the adaptation strategy) deviate from the definitions in the literature. For instance, Sutton and Barto [180] define the reward function as the expected reward for a given state-action pair, i.e. $\mathbb{E}[X_{R_{t+1}} \mid X_{S_{\lambda_t}} = s, X_{A_t} = a]$; the policy $\pi(a|s)$ is defined as a conditional probability distribution of selecting an action in a given state, i.e. $\pi(a|s) = P(X_{A_t} \mid X_{S_{\lambda_t}})$. We deviate from the literature because the underlying concern is different. In the work of Sutton and Barto, the underlying concern of MDPs is optimisation in the context of reinforcement learning. More specifically, the primary goal is to learn the distribution $\pi(a|s)$. From the moment $\pi(a|s)$ is learned the resulting mechanism selects an action by querying the learned policy, i.e. $\pi^* := \operatorname{argmax}_{a \in A} \pi(a|s)$. Based on the probabilistic definition of π, the reward function is also defined from a probabilistic perspective, as the policy must be learned in a way that maximises the expected reward. In the context of this thesis, however, the underlying concern is the evaluation of adaptation strategies. That is, the policy, or rather the adaptation strategy, is a fixed and non-probabilistic function $\pi : S \longrightarrow A$, which is plugged into the MDP framework to assess the quality of the strategy by observing the reward generated. Consequently, there is no need to define the reward function from a probabilistic point of view because sequences of states are simulated/sampled and evaluated by the reward function. Based on the generated rewards of each sequence, the expected reward could be computed anyway.

4.3.2. The Interdependency of Software Architecture and Environment

In this section, we discuss the interdependency of the software architecture (i.e. the various architectural configurations) and the environment. With the term interdependency, we refer to the mutual interaction of the two concepts, i.e. the effect of the environment on the software architecture and vice versa. Intuitively, one may argue that there is only a unidirectional dependency where only the environment forces changes in the architectural configuration. However, there are also cases in which the architectural configuration of the system affects how the environment evolves. The interdependency of architecture and environment now refers to whether the environment drives the stochastic process of a self-adaptive system solely or whether the architecture configuration also has a non-negligible effect on the environment and thus on the entire process. Let $S := (C, E,)$, $S' := (C', E',)$ be self-adaptive system states at time t and $t+1$, respectively. The probability of transitioning to S' given S is $Pr(X_{\mathcal{S}_{t+1}} = S' \mid X_{\mathcal{S}_t} = S)$ or $Pr(X_{\mathcal{C}_{t+1}} = C', X_{\mathcal{E}_{t+1}} = E' \mid X_{\mathcal{C}_t} = C, X_{\mathcal{E}_t} = E)$. From a formal perspective, the interdependency of architecture and environment is about the stochastic (in-)dependencies of the random variables $X_{\mathcal{C}_{t+1}}$, $X_{\mathcal{E}_{t+1}}$, $X_{\mathcal{C}_t}$ and $X_{\mathcal{E}_t}$. By knowing the dependency structure of the random variables, the distribution $P(X_{\mathcal{S}_{t+1}} \mid X_{\mathcal{S}_t})$ may factorise to a product of eligible distributions that provides a better understanding of the stochastic process itself. In addition, the knowledge of how a self-adaptive system moves through the state space (encoded by $P(X_{\mathcal{S}_{t+1}} \mid X_{\mathcal{S}_t})$) is of paramount importance for analysis and decision-making at runtime and design-time.

Example 1. Before discussing the interdependency of architecture and environment in more detail, we provide an example of a real use case in which the software architecture affects the environment. Consider the DeltaIoT example system presented in section 1.5.2. The system consists of a set of motes where a single mote transmits data packets to other motes over unidirectional communication channels. The probability that a data packet will be lost during transmission depends on the current SNR (signal-to-noise) level of the environment. For the sake of illustration, we neglect the activation probability and focus only on a single mote of the network. The SNR level is dependent on the current wireless interference level, i.e. the higher the wireless interference the higher the probability of packet loss. In addition, the SNR level is also dependent on the current transmission power of the given mote, i.e. the higher the transmission power the lower the probability of packet loss. The transmission power, however, is not part of the environment but of the architecture; each architectural configuration indicates different transmission powers. That is, an architectural configuration has an indirect effect on the SNR level which is part of the environment. Roughly speaking, the probability that a high SNR level is observed is lower if the current architectural configuration has a high transmission power. Furthermore, the probability that a high SNR level is observed is higher if the current architectural configuration has a low transmission power. This correlation applies as well as for the case of low SNR levels and low/high transmission powers of the corresponding configurations. Therefore, the knowledge of the current configuration (with corresponding transmission power) enables one to determine how the environment (i.e. SNR level) changes. The DeltaIoT example illustrates a scenario where the architecture and environment are interdependent. ■

In the following, we show how the transition function $t_{\mathcal{S}}$ of a self-adaptive system factorises to a product of distributions. More specifically, we prove that $t_{\mathcal{S}}$ provides an enhanced understanding of the stochastic process of a self-adaptive system itself. Additionally, it shows the interdependency of architecture and environment that primarily determines how a self-adaptive system moves through the state space. This is of paramount importance and needs to be considered for analysis at design-time or runtime. Before we mathematically derive the factorisation of $t_{\mathcal{S}}$, we need to introduce one further assumption and property.

We assume that an architectural configuration C' and environmental state E' at a specific time have no direct effect on each other. That is, the configuration C' at time $t + 1$ does not affect the current value of E' and vice versa. C' and E' are exclusively affected by the architectural configuration C and environmental state E at time t:

$$(X_{\mathcal{C}_{t+1}} \perp\!\!\!\perp X_{\mathcal{E}_{t+1}} \mid X_{\mathcal{C}_t}, X_{\mathcal{E}_t}) \tag{4.6}$$

In other words, whenever the states $X_{\mathcal{C}_t}$, $X_{\mathcal{E}_t}$ are known at time t, the states $X_{\mathcal{C}_{t+1}}$, $X_{\mathcal{E}_{t+1}}$ are independent such that the knowledge of $X_{\mathcal{C}_{t+1}}$ (given $X_{\mathcal{C}_t}$, $X_{\mathcal{E}_t}$) does not provide any information about the value of $X_{\mathcal{E}_{t+1}}$ and vice versa. This might seem to contradict what we defined as the interdependency of system and environment; however, what assumption

(4.6) states is that the effect of the mutual interaction of system and environment is not immediate (i.e. at a particular time instance t) but temporal (i.e. between time instances t and $t + 1$). For example, if an adaptation δ is applied to configuration C at time t such that we observe $\phi(C, \delta)$ at time $t + 1$ the corresponding environmental state E at $t + 1$ is not directly affected by $\phi(C, \delta)$ but solely by C at t. However, the environmental state E at $t + 2$ is possibly affected by $\phi(C, \delta)$ observed at $t + 1$.

Recall from section 4.2 the deterministic property of ϕ. Each deterministic function can be written as a probability distribution by using the indicator function:

Property 3. *The deterministic adaptation process in conjunction with property 2 implies the following equalities:*

$$P(X_{C_{t+1}} \mid X_{C_t}, X_{\Delta_t}) = P(X_{\Delta_t} \mid X_{C_{t+1}}, X_{C_t}) = \mathbb{1}_{\phi(C_t,\Delta_t)=C_{t+1}} \tag{4.7}$$

Note that we abuse notations here in which C_t, Δ_t and C_{t+1} are placeholders for concrete variable realizations in the indicator function which evaluates to 1 if $\mathbb{1}_{\phi(C_t,\delta_t)=C_{t+1}}$ holds for a given triple (C_t, δ_t, C_{t+1}) and returns 0 otherwise.

Property 3 is very important for the following lemma:

Lemma 4.3.1. *Given the deterministic property of ϕ, the following equality holds: $P(X_{C_{t+1}} \mid X_{C_t}, X_{\mathcal{E}_t}) = \chi_{\delta^*}(C_t, \mathcal{E}_t, C_{t+1})$ where*

$$\chi_{\delta^*}(C_t, \mathcal{E}_t, C_{t+1}) = \begin{cases} P(X_{\Delta_t} = \delta^* \mid X_{C_t}, X_{\mathcal{E}_t}), & \exists_{=1}\delta^* \in \Delta : \phi(C_t, \delta^*) = C_{t+1} \\ 0, & \textit{otherwise} \end{cases}$$

Proof.

$$\begin{aligned} P(X_{C_{t+1}} \mid X_{C_t}, X_{\mathcal{E}_t}) &= \sum_{\delta \in \Delta} P(X_{C_{t+1}}, X_{\Delta_t} = \delta \mid X_{C_t}, X_{\mathcal{E}_t}) \\ &= \sum_{\delta \in \Delta} P(X_{\Delta_t} = \delta \mid X_{C_t}, X_{\mathcal{E}_t}) \cdot P(X_{C_{t+1}} \mid X_{\Delta_t} = \delta, X_{C_t}, X_{\mathcal{E}_t}) \end{aligned}$$

Whenever X_{Δ_t}, X_{C_t} are known $X_{C_{t+1}}$ can be uniquely derived so that the environmental state $X_{\mathcal{E}_t}$ does not affect the probability $P(X_{C_{t+1}} \mid X_{\Delta_t}, X_{C_t}, X_{\mathcal{E}_t})$ and can be omitted.

$$= \sum_{\delta \in \Delta} P(X_{\Delta_t} = \delta \mid X_{C_t}, X_{\mathcal{E}_t}) \cdot P(X_{C_{t+1}} \mid X_{\Delta_t} = \delta, X_{C_t})$$

According to property 3:

$$= \sum_{\delta \in \Delta} P(X_{\Delta_t} = \delta \mid X_{C_t}, X_{\mathcal{E}_t}) \cdot \mathbb{1}_{\phi(C_t,\Delta_t=\delta)=C_{t+1}}$$

According to property 2:

$$
\begin{aligned}
&= \begin{cases} P(X_{\Delta_t} = \delta^* \mid X_{C_t}, X_{\mathcal{E}_t}), & \exists_{=1}\delta^* \in \Delta : \phi(C_t, \delta^*) = C_{t+1} \\ 0, & \text{otherwise} \end{cases} \\
&= \chi_{\delta^*}(C_t, \mathcal{E}_t, C_{t+1})
\end{aligned}
$$

□

Basically, lemma 4.3.1 proves that distribution $P(X_{C_{t+1}} \mid X_{C_t}, X_{\mathcal{E}_t})$ is equal to $P(X_{\Delta_t} = \delta^* \mid X_{C_t}, X_{\mathcal{E}_t})$ if and only if there is an adaptation δ^* such that $\phi(C, \delta^*) = C'$. Hereby, C and C' correspond to the configurations for which $P(X_{C_{t+1}} \mid X_{C_t}, X_{\mathcal{E}_t})$ is queried. Otherwise, the distribution returns 0.

Based on lemma 4.3.1, the following corollary can be derived:

Corollary 4.3.1. $P(X_{\mathcal{S}_{t+1}} \mid X_{\mathcal{S}_t}) = P(X_{\mathcal{E}_{t+1}} \mid X_{C_t}, X_{\mathcal{E}_t}) \cdot \chi_{\delta^*}(C_t, \mathcal{E}_t, C_{t+1})$

Proof.

Recall that a self-adaptive system state consists of an architectural configuration and environmental state (i.e. $S := (C, E)$). Thus, distribution $P(X_{\mathcal{S}_{t+1}} \mid X_{\mathcal{S}_t})$ can be written as a conditional joined distribution over four random variables:

$$P(X_{\mathcal{S}_{t+1}} \mid X_{\mathcal{S}_t}) = P(X_{C_{t+1}}, X_{\mathcal{E}_{t+1}} \mid X_{C_t}, X_{\mathcal{E}_t})$$

According to the independence assumption of 4.6:

$$= P(X_{\mathcal{E}_{t+1}} \mid X_{C_t}, X_{\mathcal{E}_t}) \cdot P(X_{C_{t+1}} \mid X_{C_t}, X_{\mathcal{E}_t})$$

According to lemma 4.3.1:

$$= P(X_{\mathcal{E}_{t+1}} \mid X_{C_t}, X_{\mathcal{E}_t}) \cdot \chi_{\delta^*}(C_t, \mathcal{E}_t, C_{t+1})$$

□

Finally, w.r.t. theorem 4.3.1, it can be shown that transition function $t_{\mathcal{S}}$ factorises to a product of two distributions:

Theorem 4.3.1. *The transition function $t_{\mathcal{S}}$ of a self-adaptive system factorises to $P(X_{\mathcal{S}_{t+1}} \mid X_{\mathcal{S}_t}, X_{\Delta_t}) = P(X_{\mathcal{E}_{t+1}} \mid X_{C_t}, X_{\mathcal{E}_t}) \cdot \mathbb{1}_{\phi(C_t, \Delta_t) = C_{t+1}}$*

Proof.

$$P(X_{\mathcal{S}_{t+1}} \mid X_{\mathcal{S}_t}, X_{\Delta_t}) = \frac{P(X_{\mathcal{S}_t}, X_{\mathcal{S}_{t+1}}, X_{\Delta_t})}{P(X_{\mathcal{S}_t}, X_{\Delta_t})}$$

$$= \frac{P(X_{\mathcal{S}_t}) \cdot P(X_{\mathcal{S}_{t+1}} \mid X_{\mathcal{S}_t}) \cdot P(X_{\Delta_t} \mid X_{\mathcal{S}_{t+1}}, X_{\mathcal{S}_t})}{P(X_{\mathcal{S}_t}) \cdot P(X_{\Delta_t} \mid X_{\mathcal{S}_t})}$$

According to property 3:

$$= \frac{P(X_{\mathcal{S}_t}) \cdot P(X_{\mathcal{S}_{t+1}} \mid X_{\mathcal{S}_t}) \cdot \mathbb{1}_{\phi(C_t, \Delta_t) = C_{t+1}}}{P(X_{\mathcal{S}_t}) \cdot P(X_{\Delta_t} \mid X_{\mathcal{S}_t})}$$

According to corollary 4.3.1:

$$= \frac{P(X_{\mathcal{S}_t}) \cdot P(X_{\mathcal{E}_{t+1}} \mid X_{C_t}, X_{\mathcal{E}_t}) \cdot \chi_{\delta^*}(C_t, \mathcal{E}_t, C_{t+1}) \cdot \mathbb{1}_{\phi(C_t, \Delta_t) = C_{t+1}}}{P(X_{\mathcal{S}_t}) \cdot P(X_{\Delta_t} \mid X_{\mathcal{S}_t})}$$

$$= \frac{P(X_{\mathcal{E}_{t+1}} \mid X_{C_t}, X_{\mathcal{E}_t}) \cdot \chi_{\delta^*}(C_t, \mathcal{E}_t, C_{t+1}) \cdot \mathbb{1}_{\phi(C_t, \Delta_t) = C_{t+1}}}{P(X_{\Delta_t} \mid X_{\mathcal{S}_t})}$$

Let $S := (C, E)$, $S' := (C', E')$ and δ be the self-adaptive system states and selected adaptation for which $P(X_{\mathcal{S}_{t+1}} \mid X_{\mathcal{S}_t}, X_{\Delta_t})$ is queried, i.e. $Pr(X_{\mathcal{S}_{t+1}} = S' \mid X_{\mathcal{S}_t} = S, X_{\Delta_t} = \delta)$. At this point two cases can occur: $\phi(C, \delta) = C'$ or $\phi(C, \delta) \neq C'$

$\phi(C, \delta) \neq C'$:

$$= \frac{P(X_{\mathcal{E}_{t+1}} \mid X_{C_t}, X_{\mathcal{E}_t}) \cdot \chi_{\delta^*}(C_t, \mathcal{E}_t, C_{t+1}) \cdot 0}{P(X_{\Delta_t} \mid X_{\mathcal{S}_t})} = 0$$

$\phi(C, \delta) = C'$:

$$= \frac{P(X_{\mathcal{E}_{t+1}} \mid X_{C_t}, X_{\mathcal{E}_t}) \cdot P(X_{\Delta_t} = \delta^* \mid X_{\mathcal{S}_t})}{P(X_{\Delta_t} \mid X_{\mathcal{S}_t})}$$

From property 2 follows $\phi(C, \delta) = C' \Rightarrow \delta = \delta^*$. This in turn means that the distribution in the denominator and $P(X_{\Delta_t} = \delta^* \mid X_{\mathcal{S}_t})$ are equal and can be truncated.

$$= P(X_{\mathcal{E}_{t+1}} \mid X_{C_t}, X_{\mathcal{E}_t})$$

Taking both cases into account, we can write $P(X_{\mathcal{S}_{t+1}} \mid X_{\mathcal{S}_t}, X_{\Delta_t})$ equivalently:

$$= P(X_{\mathcal{E}_{t+1}} \mid X_{C_t}, X_{\mathcal{E}_t}) \cdot \mathbb{1}_{\phi(C_t, \Delta_t) = C_{t+1}}$$

$\square$

Theorem 4.3.1 shows that $t_{\mathcal{S}}$ and thus the stochastic dynamics of a self-adaptive system factorises to two terms. The term $\mathbb{1}_{\phi(C_t,\Delta_t)=C_{t+1}}$ refers to the deterministic adaptation process that ensures that the architectural configurations of two states S and S' are linked by an adaptation. Otherwise, there is no chance that the system transitions from S to S', i.e. $\mathbb{1}_{\phi(C_t,\Delta_t)=C_{t+1}}$ and thus $t_{\mathcal{S}}$ evaluates to 0. The second term $P(X_{\mathcal{E}_{t+1}} \mid X_{C_t}, X_{\mathcal{E}_t})$ refers to the environmental dynamics. Intuitively, one would assume that the evolution of the environment is an independent process responsible for changes in the system configuration. However, theorem 4.3.1 showed that it cannot be ruled out whether the system (or configurations of the system) affects the stochastic evolution of the environment. In the case of the DeltaIoT system, we illustrated a scenario in which the environmental dynamics are not evolving independently but are influenced by the system configurations (i.e. the configured transmission power of each configuration). However, there are possibly other scenarios in which the environment evolves completely independently (or where the system has a negligible effect on the environmental dynamics). Ultimately, the theorem embodies what we denote as the interdependency of the architecture and environment.

As discussed at the beginning of this section, an integral part of this thesis is about evaluating self-adaptive systems (or their adaptation strategies) at design-time. Thus, the dynamics of a self-adaptive system must be taken into account to simulate the dynamic behaviour. However, because they strongly depend on the interdependency of the architecture and environment, it is crucial to make appropriate assumptions. More specifically, if the assumptions do not reflect the true dynamics that would be observed at runtime, the evaluation results at design-time are likely to be inaccurate. We will revisit the interdependency of the architecture and environment in chapter 6.

4.4. Problem Statement

After we have mapped the formal semantics of self-adaptive systems onto MDPs (see section 4.3.1) and discussed the dynamics (i.e. the transition function, see section 4.3.2), we are now able to fully describe a self-adaptive system as an MDP. Based on this formalisation, we can also explain how adaptation strategies fit into this picture. More specifically, we consider the development of adaptation strategies as the engineering challenge or problem faced by a software engineer. We formalise the engineering problem based on MDPs which is to be understood as the problem statement this work aims to address. In the following, we start to discuss the state space complexity of a self-adaptive system in section 4.4.1 and introduce afterwards the engineering problem of engineering a self-adaptive system in section 4.4.2.

4.4.1. State Space Complexity

In section 4.3.1, we formally introduced the self-adaptive system state space $\mathcal{S}$ as the Cartesian product of the set of architectural configurations C and environment $\mathcal{E}$, i.e. $\mathcal{S} := C \times \mathcal{E}$. In the context of non-adaptive systems, the architectural configuration space is comparable to the concept of the *Design Space* introduced by the work of [120]. The design space corresponds to the various system configurations induced by *Design Options*. Design options are comparable to adaptations and determine the variation points of an architecture that can be changed (e.g. changing component implementation). In terms of non-adaptive systems, a software architect needs to explore the design space to find the system configuration that satisfies the non-functional requirements of the software system. We consider the complexity of a space or set (such as the architectural configuration space C) to be the cardinality or the number of elements contained in the set (e.g. $|C|$). The complexity of the design space, for example, is the Cartesian product of the design option sets and can become very large with an increasing number of options. Since the design space (spanned by the distinct design options) and the architectural configuration space (spanned by the distinct adaptations) are closely connected concepts, we assume the size of the configuration space to be comparatively large.

Recall from section 4.1 that the environment $\mathcal{E}$ consists of a set of environmental states E. An environmental state is structured by a sequence of atomic and discrete environmental variables $E := (e_1, ..., e_n)$. Let $Val(e_i)$ be the value space of each environmental variable e_i with $|Val(e_i)| \geq 2$. The upper and lower bound of the complexity of the environment is estimated as follows:

$$\begin{aligned} |\mathcal{E}| &= \prod_{i=1}^{n} |Val(e_i)| \\ &\leq \prod_{i=1}^{n} v_{max} = v_{max}^n \text{ with } v_{max} = \max_{e \in E} |Val(e)| \\ &\Rightarrow 2^n \leq |\mathcal{E}| \leq v_{max}^n \end{aligned} \tag{4.8}$$

From formula (4.8) follows that the upper and lower bounds of the complexity of $\mathcal{E}$ are v_{max}^n and 2^n, respectively. Consequently, the complexity of $\mathcal{E}$ is exponential in the number of environmental variables n that form an environmental state E.

Note that the same complexity estimation of E can be applied to the above-mentioned design space and corresponding design options. Considering these findings, we can conclude that the state space of the self-adaptive system $\mathcal{S}$ is spanned by two spaces, each of which has exponential size.

In non-adaptive systems, it is sufficient to explore the design space of possible architectural candidates which is comparable to what we denote as architectural configuration space C. More precisely, it is assumed that there is an architectural configuration that sufficiently satisfies the quality requirements in any environmental state. In the context of self-adaptive systems, however, this is not sufficient because the aspect of time (that causes different architectural configurations) cannot be neglected. According to Esfahani and Malek [57], engineering self-adaptive systems is associated with several uncertainties. One of these uncertainties is known as *Parameter over time*. The uncertainty *Parameter over time* primarily argues that the future behaviour of the system and environment must be considered to select optimal adaptations. For example, consider a chess game in which two players must make their moves. A player who makes moves based on the current state of the chessboard (i.e. the arrangement of pieces on the chessboard) performs worse compared to moves that take into account how the opponent possibly reacts or how the state of the chess game possibly evolves. This analogy perfectly explains the uncertainty *Parameter over time* because an adaptation that seems to fit well in a given situation may have positive effects only in the short run, but may perform poorly in the long run.

Unfortunately, the uncertainty *Parameter over time* has drastic implications on the state space complexity of self-adaptive systems. Since the uncertainty implies that the temporal aspect must be taken into account when selecting an adaptation, it is not sufficient to explore the architectural configuration space and select the configuration (by applying an adaptation) that seems to be the best solution exclusively for the current state. Instead, the adaptation that achieves the best results (i.e. satisfies quality objectives) in the long run must be selected. In terms of MDPs, the adaptation that achieves the best possible accumulated reward over time is the most preferred solution regarding *Parameter over time*. More formally, selecting an adaptation δ_t in a self-adaptive system state S_t at time t has an effect on the future behaviour $S_{t+1}, \ldots, S_{T-1}, S_T$ and thus on the accumulated reward, i.e. $\sum_{i=t}^{T-1} r_{\mathcal{S}}(S_i, \delta_i, S_{i+1})$. Equivalently, an adaptation strategy π has to be engineered considering exactly these effects. Recall from section 4.3.1 that the adaptation strategy is the equivalent concept to a policy in MDPs.

To formally define the state space complexity, we must account for the temporal nature of self-adaptive systems and the uncertainty *Parameter over time*. This means that the state space complexity is defined by the sequences of states a self-adaptive system generates by changing the system configuration in response to environmental changes. We denote such a sequence as *Trajectory*.

Definition 26 (Trajectory)**.** *A trajectory τ is a sequence of self-adaptive system states $\tau := (S_0, S_1, ..., S_T)$ where state S_0 corresponds to the initial state and S_T corresponds to the state where the self-adaptive system terminates with $T \in \mathbb{N}$.*

A trajectory represents a possible path through the state space $\mathcal{S}$ that a self-adaptive system can experience when starting at an initial state S_0 and terminating at S_T at time T. We denote T as *Horizon*. Because we consider the dynamics of a self-adaptive system as a stochastic process (i.e. as MDP), there exist a multitude of possible trajectories that form the *Trajectory Space*.

Definition 27 (Trajectory Space)**.** *The trajectory space $\mathcal{T}$ encompasses all possible trajectories a self-adaptive system can traverse. More formally, as a trajectory $\tau \in \mathcal{T}$ defines a possible path through the self-adaptive system state space $\mathcal{S}$, the trajectory space is spanned by the Cartesian product of all states up to horizon T:*

$$\mathcal{T} := \mathcal{S}_0 \times \mathcal{S}_1 \times \cdots \times \mathcal{S}_T \tag{4.9}$$

The trajectory space $\mathcal{T}$ is of paramount importance because $\mathcal{T}$ encompasses all possible trajectories (of length T) a self-adaptive system can theoretically traverse. Thus, $\mathcal{T}$

constitutes the central object for evaluating adaptation strategies considering the uncertainty *Parameter over time*. Based on the definition of the trajectory space, the state space complexity is the cardinality of the trajectory space $|\mathcal{T}|$:

$$|\mathcal{T}| = |\mathcal{S}_0 \times \mathcal{S}_1 \times \cdots \times \mathcal{S}_T| = \prod_{i \in \{0,\ldots,T\}} |\mathcal{S}_i| = |\mathcal{S}|^T \tag{4.10}$$

Equation (4.10) shows that the state space complexity grows exponentially in horizon T. Moreover, as discussed at the beginning of this section, the state space $\mathcal{S}$ is of exponential complexity. The state space complexity of self-adaptive systems is more complex compared to non-adaptive systems and poses significant challenges to software engineers.

Finally, it is important to note that adaptation strategies to some extent govern how the self-adaptive systems move through the state space. Therefore, certain trajectories τ of the trajectory space are probably never observed at runtime. The subset of trajectories $\mathcal{T}_\pi \subseteq \mathcal{T}$ a self-adaptive system is traversing by following strategy π should be those that generate the highest possible rewards. However, $\mathcal{T}_\pi$ is unknown when engineering strategy π at design-time and has to be analysed. The complexity of $\mathcal{T}$ (see equation (4.10)) is an additional complicating factor that is infeasible to be analysed manually. The state space complexity again highlights the importance of automated analysis at design-time and model-based approaches. Model-based approaches introduce abstraction to deal with the complexity of $\mathcal{T}$ which is crucial to evaluating and designing adaptation strategies at design-time.

4.4.2. The Engineering Problem of Self-Adaptive Systems

In the previous sections, we introduced all the necessary concepts to formulate the engineering problem. Moreover, we catch up with the formal definition of a self-adaptive system as an MDP that we deliberately omitted in section 4.3.1.

In the previous sections, we introduced the environmental dynamics (described by a Markov chain) and the deterministic adaptation process (triggered by an adaptation strategy). In combination, both concepts induce a stochastic process capturing the dynamics of self-adaptive systems. As discussed in section 4.3.1, the stochastic process corresponds to an MDP. We have discussed how MDPs are instantiated in the domain of self-adaptive systems but have not introduced a concrete definition of self-adaptive systems as MDPs. We make up for this, taking into account the theoretical discussion and findings of section 4.3.2.

Definition 28 (Stochastic Dynamics of Self-Adaptive Systems). *The dynamics of a self-adaptive system is a stochastic process* $(X_{\mathcal{S}_t})_{t \in \mathbb{N}}$ *for which the Markov assumption holds. More precisely, the stochastic process is captured by a Markov decision process* $\lambda_{SAS} := (\mathcal{S}, \Delta, t_{\mathcal{S}}, r_{\mathcal{S}})$ *where*

- $\mathcal{S}$ *corresponds to the set of self-adaptive system states.*

- Δ *corresponds to the set of adaptations.*
- $t_{\mathcal{S}} : \mathcal{S} \times \Delta \times \mathcal{S} \rightarrow [0,1]$ *corresponds to the transition function where* $t_{\mathcal{S}} = P(X_{\mathcal{E}_{t+1}} \mid X_{C_t}, X_{\mathcal{E}_t}) \cdot \mathbb{1}_{\phi(C_t,\Delta_t)=C_{t+1}}$ *(according to theorem 4.3.1).*
- $r_{\mathcal{S}} : \mathcal{S} \times \Delta \times \mathcal{S} \rightarrow \mathbb{R}$ *corresponds to the reward function encoding quality objectives.*

The major challenge of MDPs is to develop a policy π that determines what action to execute in a given state. As discussed in section 4.3.1, the policy π corresponds to the adaptation strategy of a self-adaptive system. Therefore, we consider the engineering problem of self-adaptive systems in developing an adaptation strategy taking into account various quality objectives (captured by the reward function) and the exponential state space complexity (i.e. the trajectory space $\mathcal{T}$) discussed in section 4.4.1. When developing adaptation strategies, a software engineer must ensure that the required quality objectives (reflected by $r_{\mathcal{S}}$) are maintained over time by considering the implications of the uncertainty *Parameter over time.* That is, a strategy must be developed such that not only short-term effects but also long-term effects are considered. Consequently, the possible trajectories of the trajectory space of a self-adaptive system must be considered to evaluate a strategy regarding the uncertainty *Parameter over time.* Because adaptation strategies generate subspaces $\mathcal{T}_\pi \subseteq \mathcal{T}$, not all trajectories have to be analysed to determine the effectiveness of π. Nonetheless, it is still challenging to deal with such spaces compared to static software systems.

The engineering problem again emphasises the significant importance of design-time analysis in the development of adaptation strategies. Each strategy involves numerous design decisions, each of which has a different impact on the quality objectives. The specific impact on quality objectives cannot be foreseen at design-time without automated tool support. One may argue that there are several approaches for optimising adaptation strategies at runtime (e.g. [78]), or approaches that use models at runtime for decision-making (e.g. [209]). However, there remains the challenge of first designing a strategy as a starting point. Moreover, optimisation approaches and formal verification can benefit from design-time analyses. For optimisation, an increased convergence behaviour can be expected; in terms of formal verification, scalability is addressed by a constraint search space as a result of the pre-explored trajectory space.

4.5. Assumptions

In section 4.1, we assume a discrete set of environmental states. The assumption of discretising the environment to a set of environmental states is well established in the research community of self-adaptive systems (e.g. [182]). The discretisation of the environment has the advantage of drastically reducing the number of environmental states (compared to continuous spaces) and tackles the state space explosion problem. Moreover, it is a common approach for design-time analysis to abstract and simplify real-world concepts.

In section 4.2, we introduced the deterministic adaptation process ϕ. Without explicitly stating it, the adaptation process implies that adaptations are always applied successfully, i.e. the adaptation process does not fail. One reason for this assumption is that it is not known to which state the system transitions in case of an adaptation failure: Does the system remain in the state before the adaptation? Does the system go into a dedicated error state? Ultimately, it is again a simplifying assumption to address real-world problems and justified as a starting point to analyse self-adaptive systems at design-time. In future work, however, the assumption can be relaxed, and our approach extended both theoretically and practically.

In section 4.3.2, we assumed conditional independence of observing an environmental state E_{t+1} and architectural configuration C_{t+1} given the last state, i.e. $(C_t, \mathcal{E}_t)$ (see formula (4.6)). The assumption states that the current architecture configuration and the environmental state have no immediate influence on each other. This assumption may be too strong if the considered time difference Δt between two states is too large. However, the assumption can be sufficiently approximated if Δt is chosen precisely, e.g., by a domain expert.

Finally, an MDP is associated with several assumptions, namely the Markov assumption and the assumption of a fully observable state. As discussed at the beginning of this chapter, MDPs constitute a widely used framework in the self-adaptive system community. Their successful use strongly suggests that the assumptions associated with MDPs are reasonable. In addition, according to Koller and Friedman [105, p. 201], the Markov assumption can always be sufficiently approximated by considering a reach state description (we revisit this in the next section). Indeed, the assumption of a fully observable state is not always appropriate. In chapter 7, we discuss how this assumption can be relaxed by considering *Partially Observable Markov Decision Processes* which generalise MDPs.

4.6. Summary

In this chapter, we formally described the dynamics of self-adaptive systems. More specifically, we defined a self-adaptive system as a stochastic process that can be described as an MDP. We instantiated the framework of MDPs in the domain of self-adaptive systems. We mapped the abstract concepts of MDPs to the equivalent concepts in the domain of self-adaptive systems which we formally introduced earlier. Afterwards, we have proven a specific probabilistic behaviour of self-adaptive systems which is induced by the interdependency of the architecture and environment. We discussed the exponential complexity of the state space as a result of a self-adaptive system-specific uncertainty known as *Parameter over time*. Based on the definition of the state space complexity and the MDP-based semantics of self-adaptive systems, we formulated the engineering problem or challenge. Finally, we discussed the assumptions we made.

5. Using Bayesian Modelling to Capture the Environmental Dynamics

In this chapter, we present our metamodel for describing the environmental dynamics of self-adaptive systems. The metamodel is discussed exclusively from the perspective of modelling the environmental dynamics (i.e. the operating environment) of self-adaptive systems; it does not discuss the modelling of sensitivity models of AI components. The modelling capabilities of the metamodel go beyond modelling static probabilistic structures, as these form only one aspect of the metamodel required to capture the temporal behaviour of the environmental dynamics. For a better separation and understanding, however, we concentrate the discussion of the metamodel in this chapter solely on the environmental dynamics and discuss the modelling of sensitivity models in the section provided for this purpose (more precisely section 7.1) and then revisit the corresponding part of the metamodel. The contribution, presented in this chapter, is based on the publication [158].

There are three key aspects that our formal modelling language must address. The first aspect results from the requirement of domain-independent applicability and refers to the *Level of Abstraction.* Intuitively, the level of abstraction of a modelling language increases with the requirement of cross-domain applicability. A high level of abstraction purposefully omits domain-specific information or knowledge to allow for domain independence. This can lead to a loss of *Accuracy* in the analysis of adaptation strategies and corresponds to the second key aspect. That is, the aspects of level of abstraction and accuracy compete. However, both aspects need to be balanced in such a way that the level of abstraction is sufficient to achieve domain independence, but with an acceptable loss of accuracy to enable design-time analysis. Finally, the last aspect refers to the *Representation* of the environmental dynamics. As discussed in section 4.1, we consider the environmental dynamics as a DTMC (discrete-time Markov chain). Recall that the state space of a DTMC refers to the environment $\mathcal{E}$ consisting of environmental states $E \in \mathcal{E}$. Representing environmental dynamics appropriately is challenging due to the state space explosion problem. For instance, one may argue that as we consider environmental dynamics as DTMC, we can also represent them by state machine-based models. However, this requires the modelling of each state separately which is impractical as well as infeasible for large state spaces. Additionally, an environmental state E is considered as a tuple consisting of atomic variables, i.e. $E := (e_1, ..., e_n)$. The variables e_i might be stochastically correlated such that $P(X_{e_i}, X_{e_j}) \neq P(X_{e_i}) \cdot P(X_{e_j})$; that is, they form a network of variables with certain dependencies. Such correlations cannot be adequately captured by state machine-based models.

In this chapter, we mainly address the sub-research questions **RQ1.1** and **RQ1.2**, which bring us one step closer to answering the main research question **RQ1**. The previously discussed key aspects are reflected in **RQ1.1** and **RQ1.2**. For the sake of completeness, we restate the research question and its sub-questions (omitting **RQ1.3** and **RQ1.4**):

Research Question 1: How to evaluate adaptation strategies of self-adaptive systems at design-time regarding the ability to meet quality objectives?

Research Question 1.1: How can environmental dynamics be formalised domain-independently at design-time?

Research Question 1.2: What is an appropriate level of abstraction to represent the environmental dynamics domain independently? By appropriateness, we mean that

- adaptation strategies can be analysed at design-time with sufficient accuracy.
- environmental state spaces can be described flexibly and compactly.

In the following of this chapter, we present the environmental dynamics metamodel based on *Dynamic Bayesian Networks* (DBNs). DBNs are compact encodings of DTMCs and therefore fit perfectly into our definition of environmental dynamics (see definition 19 on page 75). Their human-understandable nature makes them easy to use, but yet powerful enough to model complex structured environments. Furthermore, DBNs are part of a broader framework called *Template-based Probabilistic Models*. Template-based models not only generalise DBNs but define a concept of domain-independent type-level descriptions that can be instantiated in domain-specific contexts to instantiate probabilistic structures (such as DBNs). We reuse these semantics of template-based models in the environmental dynamics metamodel to achieve domain independence.

This chapter is organised as follows: First, in section 5.1 we discuss requirements that the environmental dynamics metamodel must satisfy such that self-adaptive systems (or rather the adaptation strategy) are evaluated. Afterwards, we present the environmental dynamics metamodel in section 5.2. In section 5.3 we show how the model instances of the metamodel must be associated with the architecture model to instantiate domain-specific probabilistic structures capturing the environmental dynamics of a given domain. In section 5.4 we give a brief overview of the implementation of the metamodel. Finally, we discuss assumptions and limitations in section 5.5 and conclude the chapter with a summary in section 5.6.

5.1. Requirements

At the beginning of this chapter, we already discussed three key aspects that have to be addressed by our modelling language, namely the level of abstraction, accuracy and representation. In this section, we break down the key aspects and enumerate the requirements our modelling language must satisfy. The requirements can be assigned to one of the key aspects.

Domain independence Domain independence is linked to the level of abstraction aspect and is the most important requirement for cross-domain analysis. One of the central goals of this thesis is to evaluate the effect of architectural safeguards on the reliability of a system by considering non-adaptive and self-adaptive approaches. In both cases, the modelling of environmental factors or variables that impact the predictive uncertainty of an AI component is domain-specific. Therefore, the metamodel for describing such environments must not make any assumptions about structure or characteristics. Instead, they should be flexibly modelled and related to elements captured by a different model, e.g. the architecture model.

Architecture description language agnostic The approach presented in this thesis is based on architecture models to describe the fundamental structure of a software system. Therefore, the architecture model is assumed to be the main source of containing domain-specific elements. The environment model (based on our metamodel) is expected to supplement the architecture model. More specifically, the modelled environment has to be instantiated in the architecture model, i.e. by relating environmental variables to the domain-specific elements captured by the architecture model. However, there are several ADLs (architecture description languages) for describing architecture models, e.g. [149, 87, 73, 21]. Some of them are rather generally applicable (e.g. ADL for component-based software architectures [149]) while others are more domain-specific (e.g. embedded systems [21]). Therefore, the metamodel should be ADL-agnostic such that environment models can be instantiated in any architectural model. Modelling environments independent of the used ADL facilitates domain independence and contributes to the key aspect level of abstraction.

Stochastic dynamics From a theoretical perspective, we consider the environmental dynamics as a stochastic process or DTMC as the environmental state changes over time. More specifically, the goal is to sample sequences or trajectories from the DTMC capturing the stochastic dynamics of the environment. The trajectories are important for analysing the quality of an adaptation strategy. As discussed at the beginning of this chapter, one may argue to use state machine-based models to capture the stochastic dynamics of a DTMC. However, this is impractical in terms of modelling large state spaces. Therefore, the metamodel must represent the stochastic dynamics or DTMC in such a way that trajectories of environmental states can be generated from the model, taking into account a low modelling effort. This requirement refers to the key aspect of representation.

Stochastic correlations An environmental state comprises environmental variables, i.e. $E := (e_1, ..., e_n)$. The variables might correlate with each other. For instance, let us consider a web-based software system in which a state consists of two variables $E := (e_W, e_S)$. The first variable e_W describes the current workload of the system; the second variable e_S describes the state of the server, i.e. available or not available. For the sake of illustration, we assume a single server. The variable e_S is dependent on e_W because the probability of observing a server failure increases with an increased workload of e_W. Therefore, if we want to sample a trajectory of environmental states the direct effect of e_W at time t to e'_S at time $t + 1$ has to be captured (for all t):

$$E_1, E_2, \ldots, E_t := (e_W, e_S)_t, E_{t+1} := (e'_W, e'_S)_{t+1}, \ldots, E_T$$

Such correlations have to be represented by the metamodel and form another requirement that can be assigned to the key aspect representation.

Compactness State machine-based models are impractical to describe the DTMC due to large state spaces. This is true not only for modelling the states and transitions (i.e. the structure of the DTMC) but also for specifying $t_{\mathcal{E}}$. In state machine-based models, $t_{\mathcal{E}}$ is represented by a transition matrix. The number of matrix entries grows quadratically with the number of states. The manual creation of the transition matrix is costly even for medium-sized state spaces. Therefore, the metamodel has to reduce the effort of modelling $t_{\mathcal{E}}$. The requirement of compactness is strongly related to the stochastic dynamics requirement and thus contributes as well to the key aspect of representation.

Discretisation level Finally, the discretisation level constitutes the last requirement. As discussed in section 4.1, an environmental state consists of a tuple of discrete variables such that the state itself is discrete. Discretisation, however, can be either fine-grained or course-grained. For instance, a continuous variable with a range $[0, 100]$ can be discretised by a resolution of 10 (i.e. 10, 20, 30, ..., 100) or 5 or 1 and so on. The finer the resolution, the more adequately reality is captured; at the cost of enlarged state space. The choice of a suitable discretisation level is up to the software developer (or domain expert) and should be flexibly configurable by the metamodel. The discretisation level belongs to the key aspect of accuracy because it directly affects the accuracy of the analysis of adaptation strategies.

5.2. The Environmental Dynamics Metamodel

In this section, we introduce the environmental dynamics metamodel, which we abbreviate as *EnvDyn* in the following. The formal modelling language integrates concepts to describe and instantiate probability spaces domain independently. However, before we introduce the main concepts and formal semantics of our metamodel, we first discuss how to use DBNs to represent the environmental dynamics and how it fits in our formal framework presented in chapter 4.

5.2.1. Representing Environmental Dynamics with Dynamic Bayesian Networks

In section 4.1, we formally described the environmental dynamics as DTMCs. The reason for considering environmental dynamics as DTMCs is to show (i) how they fit into a broader mathematical framework (i.e. considering self-adaptive systems as MDPs) and (ii) for generalisation. DTMCs (and other state machine-based representations), however, are not appropriate to model large and complex structured stochastic environments. Instead, an alternative approach needs to be taken into account which maintains the semantics and assumptions associated with DTMCs.

Such an approach or formal framework provides DBNs. DBNs are probabilistic graphical models (recall from section 2.6) that comprise a family of probabilistic models. Probabilistic graphical models provide a powerful framework for representing complex probability distributions in a manageable way. As introduced in section 2.6.2, DBNs are specialisations of DTMCs. That is, w.l.o.g. we can represent the environmental dynamics as DBNs without violating the formal semantics of chapter 4.

Using DBNs, however, is not only promising to maintain mathematical consistency but also for practical reasons. Formally, a DBN is a tuple $(\mathcal{B}_0, \mathcal{B}_\rightarrow)$ where $\mathcal{B}_0$ is a BN (Bayesian network) describing the initial distribution over the states and $\mathcal{B}_\rightarrow$ is a 2-TBN inductively describing the dynamic evolution of the states. The core of the DBN forms the probabilistic structure encoded by $\mathcal{B}_0$ (which from now on we simply refer to as $\mathcal{B}$). The BN $\mathcal{B}$ constitutes a graph that enables the specification of probabilistic relations. The graph-based representation provides an intuitive and human-understandable structure for modelling complex probability spaces. In addition, the decomposability (see section 2.6.1) property factorises the BN into a set of local CPDs (conditional probability distributions); each CPD is associated with a node in the graph and can be manually estimated by a domain expert. The human understandable nature and the decomposability property of Bayesian models provide a foundation to incorporate domain knowledge. Domain experts can model the probabilistic structure with a very familiar and intuitive graph structure. Even in the absence of domain experts, the structure and parameters (of the local CPDs) of BNs and DBNs can be learned from data, e.g. [105, 130, 129]. Finally, the initially modelled BN $\mathcal{B}$ is complemented by a temporal or dynamic extension captured by 2-TBN $\mathcal{B}_\rightarrow$. Similarly to BNs, 2-TBNs are described in a graph-based structure and local CPDs. Moreover, the Markov and stationary assumption (see section 2.4.1) empower DBNs to compactly represent entire trajectory spaces. This allows DBNs to sample environmental states and perfectly satisfies the requirement of representing the stochastic nature of the environmental dynamics. Capturing the environmental dynamics with DBNs constitutes one of the key aspects of this work to evaluate adaptation strategies of self-adaptive systems.

Therefore, the key idea of this chapter is to present a metamodel for describing the environmental dynamics based on the semantics of DBNs. The generic and probabilistic structure of the environmental variables of an environmental state $E := (e_1, ..., e_n)$ are initially modelled with BNs. The graph $\mathcal{G}$ associated with $\mathcal{B}$ captures the stochastic correlations of the variables and describes the initial distribution. Recall definition 19 on page 75 of the environmental dynamics and the set of initial states $\mathcal{E}_0$. The modelled BN represents the initial distribution over $\mathcal{E}_0$ based on $\mathcal{G}$ and the local CPDs, which we call *Static Environment*. The transition function $t_{\mathcal{E}}$ includes the knowledge about the evolution of the environmental states and is captured by the 2-TBN $\mathcal{B}_{\rightarrow}$. Roughly speaking, a DBN $(\mathcal{B}, \mathcal{B}_{\rightarrow})$ describes the static environment $\mathcal{E}$ by the non-dynamic BN $\mathcal{B}$ and the *Dynamic Environment* or *Environmental Dynamics* by the 2-TBN $\mathcal{B}_{\rightarrow}$, i.e. $(\mathcal{B}, \mathcal{B}_{\rightarrow})$ compactly encodes the DTMC $(\mathcal{E}, \mathcal{E}_0, t_{\mathcal{E}})$ from definition 19.

Finally, note that DTMCs induce a set of discrete states; regarding the environmental dynamics, the set relates to $\mathcal{E}$. BNs and DBNs are structured as a network of random variables according to some graph $\mathcal{G}$ encoding a set of local CPDs. The random variables over which the CPDs are defined are potentially continuous (and thus the CPD itself) and violate the discrete state property. Therefore, we assume all random variables of a DBN describing the environmental dynamics as discrete. Moreover, we assume that all local CPDs are multinomial distributed such that the discrete state property is maintained. We defer the discussion of the assumption to section 5.5. The multinomial assumption drastically reduces the state space. Thus, the state space explosion problem is addressed and facilitates design-time analysis of adaptation strategies.

5.2.2. Overview of the Metamodel

Before we present the individual concepts of the *EnvDyn* metamodel, we first provide an overview. Therefore, consider Figure 5.1 which depicts the metamodel.

The metamodel is divided into three packages, namely `template`, `static` and `dynamic`. Roughly speaking, the `template` package allows the modelling of random variables and their distributions at the type level. The package defines template variables and template factors based on the concepts of template-based probabilistic models (see section 2.6.3).

The `static` package includes the metamodel elements for describing BNs and thus the non-temporal or static environment $\mathcal{E}$, i.e. the probabilistic structure of the environmental variables. Therefore, the defined template variables and factors of the `template` package are instantiated to generate random variables. The BN is generated w.r.t. the dependency structure of the templates. Based on the concepts of template-based models from section 2.6.3, we connect each ground random variable to a particular domain object in the architecture model to complete the instantiation process. It may seem contradictory that we use the term "instantiate" within the same meta-level i as the term is rather used to describe instances at meta-level $i-1$. However, in *Multi-Level Modelling* it is common

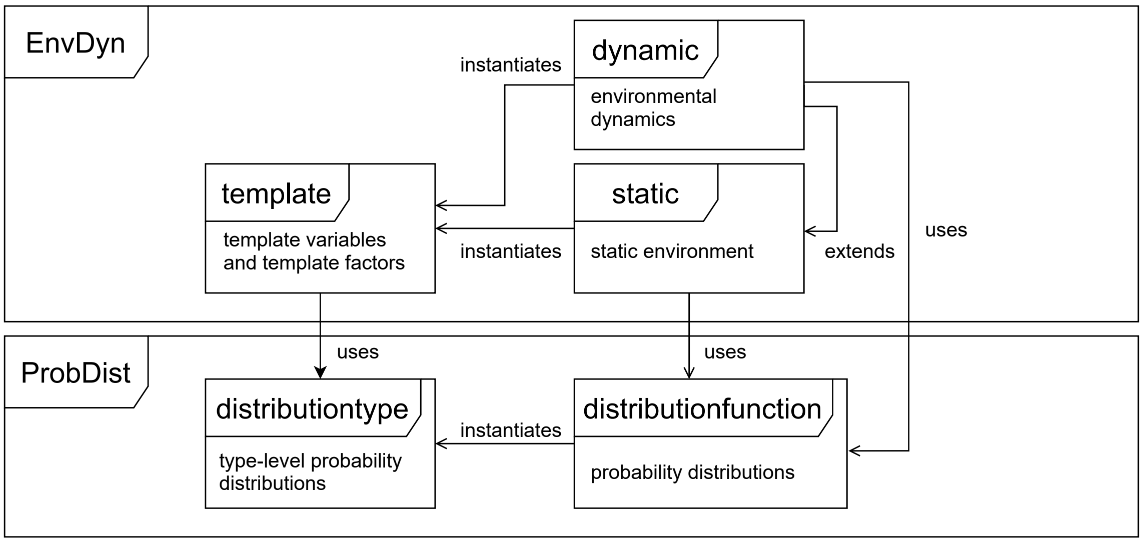


Figure 5.1.: Overview of the *EnvDyn* metamodel packages (including the *ProbDist* metamodel packages).

to describe instances ontologically (i.e. at the same meta-level) because it circumvents constraints of strict meta-modelling [49].

The `dynamics` package complements the metamodel by modelling dynamic and temporal behaviour. As we consider DBNs as environmental dynamics, this part of the metamodel encompasses metamodel elements for specifying the stochastic evolution, i.e. $\mathcal{B}_{\rightarrow}$. The semantics of 2-TBNs complete the metamodel such that the created models describe DBNs of the form $(\mathcal{B}_0, \mathcal{B}_{\rightarrow})$.

Finally, Figure 5.1 depicts a fourth metamodel for modelling probability distributions; we call the metamodel *ProbDist*. The metamodel enables the modelling of probability distributions at type- and instance-level. The packages `template`, `static` and `dynamic` provide the required semantics to represent probabilistic structures (Bayesian models) but do not specify elements for defining probability distributions. This gap is addressed by the *ProbDist* metamodel. In the *EnvDyn* metamodel, only multinomial distributions are considered and represent one of many distribution types that can be modelled with the *ProbDist* metamodel. Therefore, we extracted the metamodel as a standalone modelling tool for specifying arbitrary probability distributions that can be reused in different contexts or metamodels.

5.2.3. Modelling Domain-Independent Template Variables and Template Factors

In this section, we present the `template` package in more detail. We start to explain all the elements and formal semantics of the metamodel. Afterwards, we illustrate the application of the metamodel package by applying it to the DeltaIoT example system from section 1.5.2. The package is depicted on Figure 5.2. For illustrative purposes, we have omitted

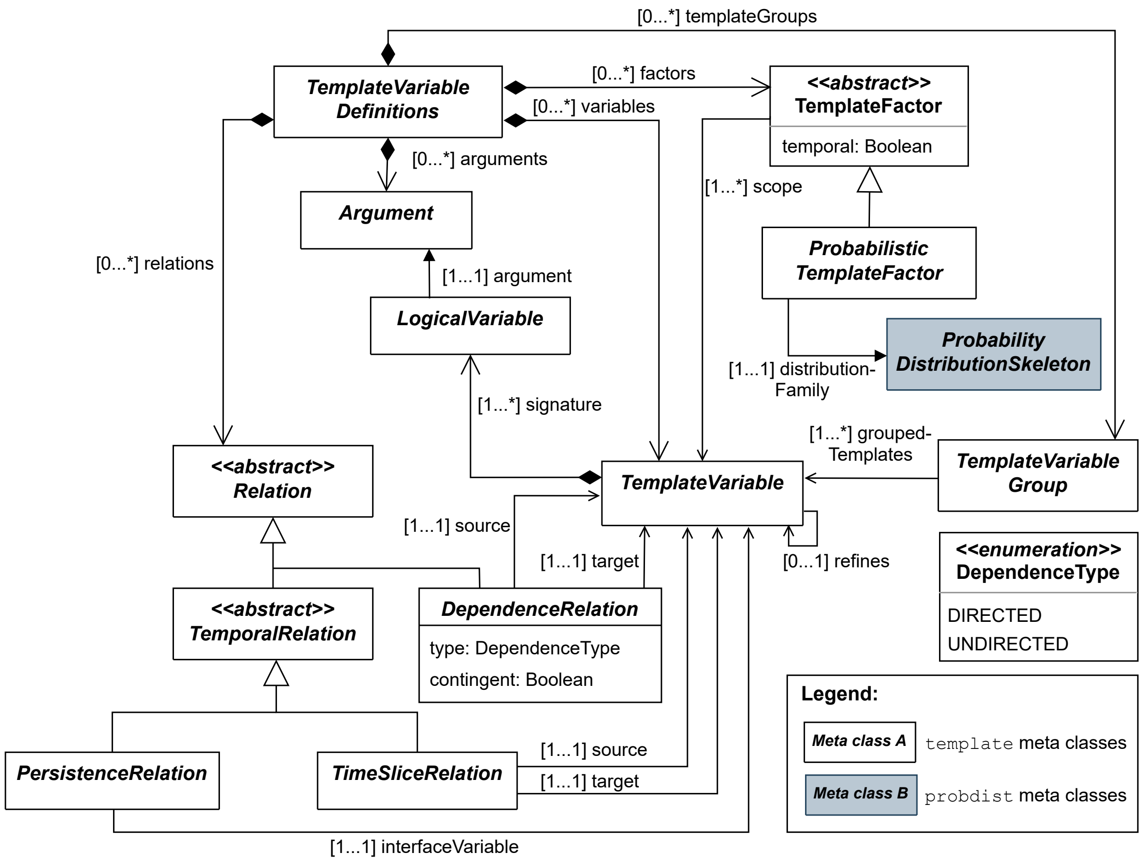


Figure 5.2.: The `template` package of the metamodel for modelling type-level random variables.

descriptive attributes such as name or ID to provide a clear overview of the metaclasses and their relationships.

5.2.3.1. Formal Semantics

As discussed in previous sections, our metamodel reuses concepts of template-based probabilistic models. The two main building blocks in template-based models are template variables and template factors which play a central role in this part of the metamodel.

The root element of the `template` package forms the `TemplateVariableDefinitions`. It defines the starting point for modelling `TemplateVariables`, `TemplateFactors`, `Relations`, `Arguments` and `TemplateVariableGroups`. Starting from `TemplateVariableDefinitions`, `TemplateVariables` and `Arguments` can be created. Recall from section 2.6.3 that template variables describe random variables at the type level. In our metamodel, a template variable is represented by the entity `TemplateVariable`. An `Argument`-entity is indirectly linked to a `TemplateVariable` by a `LogicalVariable`. Thus, a `TemplateVariableDefinitions` instance can include a set of `Argument` instances. Additionally, `TemplateVariables` possess a set of `LogicalVariables` where each `LogicalVariable` is uniquely associated with an `Argument`. The set of `LogicalVariables` of a `TemplateVariable` forms its argument signature.

The entities `TemplateVariables`, `Argument` and `LogicalVariable` and their relationships are semantically equal to the formal definition of template variables. For better readability, we give the definition again:

Definition 13 (Template Variable)**.** *A template variable $\mathcal{V}(U_1, ..., U_n)$ is a function with some range $Val(\mathcal{V})$. Each argument U_i of $\mathcal{V}$ is a typed logical variable where $Q[U_i] \in \boldsymbol{Q}$.*

More specifically, a template variable $\mathcal{V}$ is represented by the entity `TemplateVariables`. The argument signature $\alpha(\mathcal{V}) = (U_1, ..., U_n)$ of a template corresponds to the set of `LogicalVariables` contained in `TemplateVariable`. Because each U_i is associated with a specific class $Q \in \boldsymbol{Q}$ (i.e. $Q[U_i]$), each `LogicalVariable` is typed by a single `Argument` representing a class Q.

The second key concept of template-based models refers to template factors. As before, we start to give the definition again:

Definition 14 (Template Factor)**.** *A template factor $\xi : Val(\mathcal{V}_1) \times \cdots \times Val(\mathcal{V}_l) \rightarrow \mathbb{R}$ is a function defined over template variables $(\mathcal{V}_1, \ldots, \mathcal{V}_l)$. Given a tuple of ground random variables $(X_1, ..., X_l)$, if $\forall i \in \{1, ..., l\} : Val(X_i) = Val(\mathcal{V}_i)$ holds true, then $\xi(X_1, ..., X_l)$ defines the instantiated factor from $X_1, ..., X_l$ to $\mathbb{R}$ w.r.t. $(\mathcal{V}_1, ..., \mathcal{V}_l)$.*

Recall from section 2.6.3 that template factors are (just as template variables) a type-level construct to describe probabilistic properties. A template factor is defined over a set of template variables $\mathcal{V}_1, \ldots, \mathcal{V}_l$ that denotes the scope of factor ξ. In terms of our metamodel, template factors are represented by the entity `TemplateFactor`. To satisfy the formal semantics of definition 14, `TemplateFactor` is referencing a set of `TemplateVariables` which define the scope of the factor. The metaclass `TemplateFactor` is abstract to facilitate the extendability of the metamodel. Template-based formalisms are not only suitable for modelling probability structures based on directed graphs (e.g. BNs and DBNs), but also for structures based on undirected graphs (e.g. Markov random fields). Currently, `TemplateFactor` is only extended with the entity `ProbabilisticTemplateFactor` to model probability distributions but could be extended for other types of factors (e.g. factors of Markov random fields). `ProbabilisticTemplateFactor` are referencing exactly one `ProbabilityDistributionSkeleton` which is part of the `prodist` metamodel. We discuss the metamodel in section 5.2.6 in more detail. At this point, it is sufficient to know that `ProbabilityDistributionSkeleton` describes types or families of probability distributions such as multinomial, exponential or uniform distributions. In summary, a `TemplateFactor`-entity is defined over a set of `TemplateVariables` which forms the scope of the factor. It also specifies a distribution family (e.g. multinomial) so that the distribution parameters must be specified after each instantiation of a factor.

In addition to template variables and factors, the `template` package of the metamodel has a third concept that allows the specification of *Relations* between template variables. The central entity here is `Relation`. Basically, `Relations` define relationships between template variables and thus induce the graph-based structure of BNs and DBNs after instantiation. `Relation` is abstract and extended by the metaclasses `TemporalRelation` and `DependenceRelation`. A `DependenceRelation` defines a relation between two template variables by specifying a `source` and `target`. `DependenceRelations` have a `type` that refers to an enumeration called `DependenceType` with the constants `DIRECTED` and `UNDIRECTED`. Note that (just as for template factors) we facilitate extendability by allowing the specification of undirected graphs, e.g. to model Markov random fields. Throughout this work, however, we focus only on BNs and DBNs and thus consider only directed graph structures. Therefore, an `DependenceRelation` instance of type `DIRECTED` describes a parent-child relationship between two template variables where `source` attribute refers to the parent and the `target` attribute refers to the child. While `DependenceRelation` is primarily used to encode stochastic dependencies for static probability structures (i.e. BNs), `TemporalRelation` specifies dynamic dependencies of DBNs. `TemporalRelation` is also abstract and extended by `PersistenceRelation` and `TimeSliceRelation`. `PersistenceRelation` and `TimeSliceRelation` are DBN-specific relations based on the semantics of persistence edges and inter-time-slice edges known from DBNs. Recall from section 2.6.2 that in DBNs time-slice edges define the dependency between two random variables between two time slices, i.e. $X_t \longrightarrow Y_{t+1}$ from time t to $t+1$. Persistence edges are a special kind of time-slice edges where the two random variables between two time slices are equal, i.e. $X_t \longrightarrow Y_{t+1}$ where $X = Y$. In the `template` package of the metamodel, however, we distinguish between `PersistenceRelation` and `TimeSliceRelation`, although the former is a special case of the latter. The main reason for that is (i) a clear separation of concepts and (ii) a simplified modelling process, as a `PersistenceRelation` is associated with a single `TemplateVariable`. `TimeSliceRelation` defines temporal relations of `TemplateVariables` that are not equal.

Finally, the entity `TemplateVariableGroup` groups a set of templates forming a BN. Such template groups can be instantiated multiple times; this is discussed in more detail in section 5.3.

Based on the semantics of `Relations`, `TemplateVariables` and `TemplateFactors`, we can describe plate models from section 2.6.3.1. The generic framework of plate models are based on template-based probabilistic models and thus share the same semantics in terms of template variables $\mathcal{V}$, template factors ξ, logical variables $U_1, \ldots, U_n$ and object classes $\mathcal{Q}$. Plate models are well-established probabilistic graphical models that are commonly used in practice [105, P. 216]. Although plate models provide much more concepts than discussed in this work, we only consider the formal definition of plate models. In this way, we show that our metamodel is in conformance with the semantics of plate models, as they provide the fundamental concepts for instantiating ground Bayesian networks in arbitrary domains; this is discussed in the section 5.2.4.1. Here again, we repeat the formal definition of plate models from section 2.6.3.1:

Definition 15 (Plate Model). *For a set of template variables $\mathcal{V} \in \aleph$ with argument signature $\alpha(\mathcal{V}) = U_1, ..., U_n$, let $B_i(\mathbf{U_i})$ denote the variables of the argument signature of parent B_i. A plate model M_{Plate} defines for each template:*

- *A set of template parents* $Pa(\mathcal{V}) := \{B_1(\mathbf{U_1}), ..., B_k(\mathbf{U_k})\}$ *in which* $\forall i \in \{1, ..., k\} : B_i(\mathbf{U_i}) \Rightarrow \mathbf{U_i} \subseteq \{U_1, ..., U_n\}$.
- *A template CPD* $P(\mathcal{V} \mid Pa(\mathcal{V}))$.

The set of template variables $\aleph$ is captured by `TemplateVariableDefinitions`. We already discussed how template variables and their argument signatures are captured by the metamodel, i.e. `TemplateVariable`, `LogicalVariable`, `Argument`. Template conditional probability distributions $P(\mathcal{V} \mid Pa(\mathcal{V}))$ are represented by `TemplateFactors`. The parent structure for each template $\mathcal{V}$ is induced by the `Relation` instances. Thus, for each $\mathcal{V}$, the corresponding template parents $Pa(\mathcal{V})$ are determined by browsing the modelled `Relations`. Requirements, such as enumerated in definition 15, i.e. $\forall i \in \{1, ..., k\} : B_i(\mathbf{U_i}) \Rightarrow \mathbf{U_i} \subseteq \{U_1, ..., U_n\}$ and $\forall \mathcal{V} \in \aleph, \exists \xi \in \Xi : \xi = P(\mathcal{V} \mid Pa(\mathcal{V}))$, can be enforced by defining OCL constraints at metamodel-level.

The `template` package adheres to the formal semantics of template variables and template factors which are the two main building blocks in template-based models for defining probability models with reoccurring structures. Furthermore, the package is complemented with the concept of relations to express dependencies for static and dynamic probabilistic models. Based on template variables, template factors and relations, we are now able to describe probabilistic structures at the type level. These probabilistic structures induce BNs that can be instantiated in several domains. The instantiation of template variables and factors is discussed in section 5.2.4.

5.2.3.2. Applying Template Variables and Template Factors

In this section, we give an illustrative example of how the `template` metamodel package can be applied. In doing so, we define a set of template variables for the DeltaIoT example system that we presented in section 1.5.2.

Recall that there are several uncertainties in the DeltaIoT system, namely wireless interference, SNR (signal-to-noise ratio) and fluctuations in traffic load. All these factors have a direct influence on the quality attributes of the system, i.e. packet loss and energy consumption. Because wireless interference, SNR and fluctuations in traffic load fit into our definition of an environmental state, they are considered environmental variables, each of which is represented by an environmental variable, say WI for the wireless interference variable, SNR for the SNR variable and MA for the variation in traffic load w.r.t. mote activation. It is important to note, however, that the environmental state of the DeltaIoT system does not only consist of the three variables. For example, wireless interference occurs whenever there is a wireless link between two communicating motes. Accordingly, there are different SNR values for each wireless link and different traffic fluctuations for different motes. Therefore, the variables WI, SNR and MA represent template variables that have to be instantiated for each wireless link and mote. As a result, the environmental state of the DeltaIoT system encompasses a set of environmental variables that represent ground random variables instantiating WI, SNR and MA. Figure 5.3 shows the template

MA
Mote
WI
SNR
Wireless link

(a) Plate model of mote activation

(b) Plate model of wireless interference and SNR

Figure 5.3.: Plate model representation of the template variables and object classes of the environmental variables of the DeltaIoT system: (a) represents the mote activation variable, (b) shows the wireless interference and SNR variables.

variables and their corresponding object class Q in the plate model notation (see section 2.6.3.1).

It can be seen from Figure 5.3b that the SNR template depends on the WI variable. This results from the fact that the SNR value represents the ratio of the level of a mote's signal and the level of a noise signal that refers to the wireless interference variable WI. Both the WI and SNR template share the same object class *Wireless Link* (abbreviated as Q_{WL}). That is, they can only be instantiated in the architecture model for objects that are considered to be instances of object class Q_{WL}. Similarly, the template MA has object class *Mote* (abbreviated as Q_{Mote}) indicating that MA can only be instantiated for objects that correspond to class Q_{Mote}.

Based on the template variables WI, SNR and MA a `TemplateVariableDefinitions` instance of the `template` metamodel package is created to model the templates. Figure 5.4 depicts an excerpt of the `TemplateVariableDefinitions` instance that specifies the template variables WI, SNR of the DeltaIoT system.

For illustrative purposes, we only show the template variables WI and SNR, their argument signatures (i.e. $\alpha(WI)$, $\alpha(SNR)$), their temporal and non-temporal relationships and their respective template factors; otherwise, the illustration would lack clarity due to the multiple relationships between the model elements. It can be seen that the template variables WI and SNR are modelled by two `TemplateVariable` instances with name `WirelessInterference` and `SNR`. Both share the same argument signature; each of them has a single `LogicalVariable` instance referencing the same `Argument` `WirelessLink`, i.e. object class Q_{WL}.

The dependency structure given by WI and SNR is modelled by a `DependencyRelation` object where the `source` is pointing to the `WirelessInterference` `TemplateVariable` and `target` is associated with the `SNR` `TemplateVariable` as shown in Figure 5.3b. So far, the structural elements of type-level template variables and their relationships have been modelled, which only need to be complemented by the probabilistic specifications, i.e.

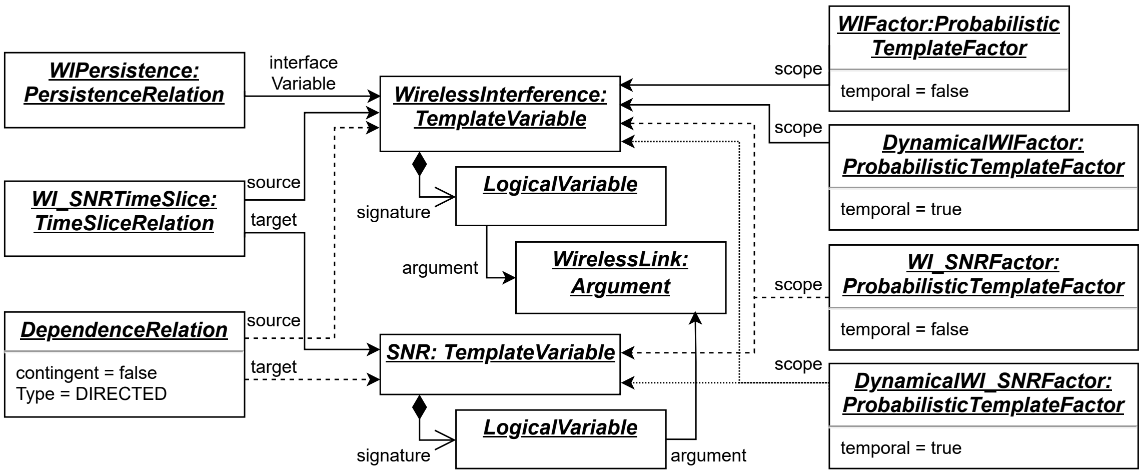


Figure 5.4.: An excerpt of the `TemplateVariableDefinitions`-instance of the `template` metamodel package applied to the DeltaIoT system. Note that the different appearances of the arrows have no semantic meaning but merely serve the purpose to distinguish between the references of the model elements.

the template factors. Regarding WI and SNR, there are two template factors (captured by a respective `ProbabilisticTemplateFactor`): The `WIFactor` describing $\xi_{WI} = P(WI)$ and `WI_SNRFactor` describing $\xi_{SNR} = P(SNR \mid WI)$. Template factors are defined over a set of template variables, i.e. the scope. The scope of `WIFactor` is a single template variable (namely `WirelessInterference`). In contrast, the scope of `WI_SNRFactor` is defined over two template variables (namely `WirelessInterference` and `SNR`) as it described a CPD with `WirelessInterference` as the conditional variable. Note that these type-level constructs refer to the static part (i.e. $\mathcal{B}_0$) of the environmental dynamics $(\mathcal{B}_0, \mathcal{B}_{\rightarrow})$. Thus, the attribute `temporal` is set to `false` accordingly. As discussed in the last section, each `ProbabilisticTemplateFactor` is referencing a single distribution type of the `probdist` metamodel package which is, however, note depicted on Figure 5.4. For the sake of illustration, we omitted the reference. Because we assume multinomial distributions, each `ProbabilisticTemplateFactor` refers to a corresponding `probdist` instance indicating that the distribution belongs to the family of the multinomial distribution.

For the dynamic part $\mathcal{B}_{\rightarrow}$, two more relations and template factors are modelled. There are two temporal relations, namely `PersistenceRelation` `WIPersistence` and `TimeSliceRelation` `WI_SNRTimeSlice`. `WIPersistence` is referencing the template variable `WirelessInterference` and indicates that the template variable has a persistent behaviour over time. The `TimeSliceRelation` `WI_SNRTimeSlice` models the temporal dependency of WI at time t on SNR' at time $t + 1$. Note that the `WI_SNRTimeSlice` is not a persistence relation as it does not exhibit persistent behaviour over time, but is primarily driven by the wireless interference of the environment. Suppose the wireless interference is non-existent, the SNR value of a wireless link would be constant (w.r.t. the transmission power of the respective mote). The corresponding 2-TBN $\mathcal{B}_{\rightarrow}$ of WI and SNR would look as depicted on Figure 5.5.

The `template` instance is completed by the temporal template factors, each of which is modelled by a `ProbabilisticTemplateFactor` instance with attribute `temporal` set to

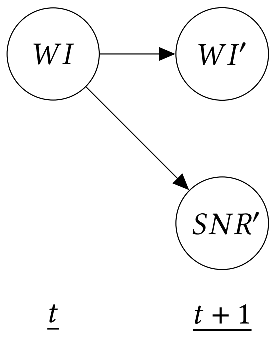


Figure 5.5.: 2-TBN $\mathcal{B}_\rightarrow$ of template variables WI and SNR

`true`. `DynamicalWIFactor` models the template factor $\xi_{WI'} = P(WI' \mid WI)$. Although the $\xi_{WI'}$ is defined over two template variables (i.e. WI, WI') the scope encompasses only the template variable `WirelessInterference`. To simplify modelling, there is no need to reference the same template variable two times as it can be derived by the `temporal` attribute of the `ProbabilisticTemplateFactor` and the `PersistenceRelation`. Finally, the `DynamicalWI_SNRFactor` describes template factor $\xi_{SNR'} = P(SNR' \mid WI)$ where the scope is defined over the template variables `WirelessInterference` and `SNR`.

5.2.4. Modelling the Static Environment

In this section, we discuss the `static` package of the *EnvDyn* metamodel. The `static` package encompasses all concepts relevant to model the static environment with BNs, i.e. the $\mathcal{B}_0$ part of the environmental dynamics $(\mathcal{B}_0, \mathcal{B}_\rightarrow)$. More specifically, it instantiates template variables and factors modelled with the `template` package. Note again that template variables and factors are type-level descriptions of random variables and probability distribution families. Therefore, the `static` package contains the entities for instantiating template variables and factors of the `template` package. The `static` package of the metamodel is depicted in Figure 5.6.

5.2.4.1. Formal Semantics

In this section, we present the formal semantics of the static package of the *EnvDyn* metamodel. The package provides all modelling concepts to instantiate type-level template variables and factors. The modelled `relations` from section 5.2.3 specify parent-child relationships between templates and induce probabilistic structures, i.e. BNs. Recall from section 2.6.3.1 that a plate model M_{Plate} and an object skeleton κ generate a ground Bayesian network. The object skeleton κ specifies for each class $Q \in \mathcal{Q}$ a finite set of objects for which template variables can be instantiated. In section 5.2.3, we formally described how plate models can be generated by the entities of the `template` package. In this section, we present how BNs are instantiated based on an object skeleton κ and modelled template variables and factors that adhere to the formal semantics of plate models. However, we

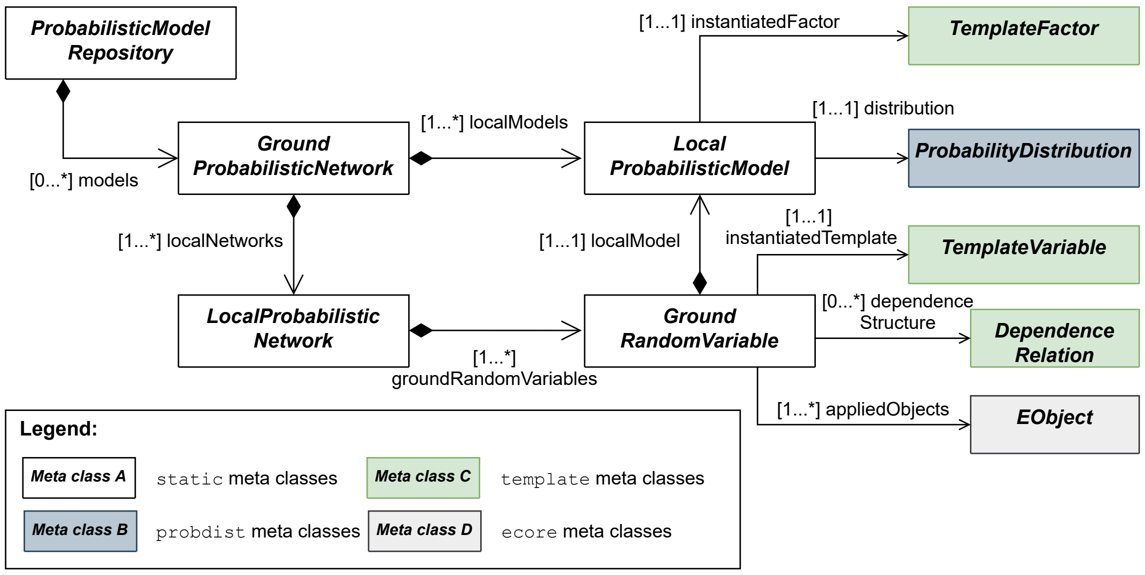


Figure 5.6.: The `static` package of the metamodel for modelling Bayesian networks.

start by introducing the metaclasses of the `static` package and then formally describe how they induce ground Bayesian networks.

The root element of the `static` package corresponds to `ProbabilisticModelRepository`. A `ProbabilisticModelRepository` contains a set of `GroundProbabilisticNetworks` that represent instantiated BNs.

Each `GroundProbabilisticNetwork` references a set of `LocalProbabilisticNetworks`. `Lo-calProbabilisticNetworks` consist of `GroundRandomVariables` and describe local probabilistic structures of the BN. One would expect that `GroundProbabilisticNetwork` refers only to a single `LocalProbabilisticNetwork`. However, this is generally not the case; this is illustrated in section 5.2.4.2. Each `GroundRandomVariable` is associated with exactly one `TemplateVariable` of the `template` package, i.e. to indicate that the `Templat-eVariable` is instantiated by the `GroundRandomVariable`. Recall that each template variable is instantiated for a set of objects included in object skeleton κ, i.e. $\mathcal{V}(\gamma)$ where $\gamma = (U_1 \mapsto u_1, \dots, U_k \mapsto u_k)$. Therefore, a `GroundRandomVariable` is associated with a set of `EObjects`. Technically, we realised the *EnvDyn* metamodel in the *Eclipse Modelling Framework* [177], which allows the design and construction of metamodels. The *Env-Dyn* metamodel is realised with the EMF, where `EObject` corresponds to the root of each model object (equivalent to java.lang.Object in the Java programming language). Thus, a `GroundRandomVariable` can be associated with any kind of EMF-based model object. Since the PCM is also implemented in the context of EMF, a `GroundRandomVariable` references a `TemplateVariable` that is instantiated for a set of model objects (i.e. PCM-specific model objects); we revisit the topic in section 5.3. Finally, each `GroundRandomVariable` might have parents described by `DependenceRelations` of the `template` package. Thus, a `GroundRandomVariable` is referencing a set of `DependenceRelations` indicating the parents or dependence structure of the random variable. Based on the relations and `EObjects` (i.e.

the objects for which the random variable is instantiated), the parent `GroundRandomVariables` are resolved.

Principally, a BN consists of a structural (i.e. $\mathcal{G}$) and a parametric part (i.e. a set of CPDs associated with $\mathcal{G}$). The structural part is captured by the previously introduced entities, namely `GroundProbabilisticNetworks`, `LocalProbabilisticNetworks` and `GroundRandomVariables`. The parametric part is captured by the entity `LocalProbabilisticModel` which (i) instantiates `TemplateFactors` of the `template` package and (ii) describes the concrete probability distribution of a `GroundRandomVariable`. Thus, each `GroundRandomVariable` refers to exactly one `LocalProbabilisticModel` where each model describes the corresponding distribution. Furthermore, each LocalProbabilisticModel refers to a single `TemplateFactor` that models the instantiation of said factor. Recall from definition 13 that a template factor is a function defined over a set of template variables; that is, the scope of `TemplateFactor`. Thus, a `TemplateFactor` (or more specifically a `ProbabilisticTemplateFactor`) is properly instantiated by a `LocalProbabilisticModel` if the scope of the `TemplateFactor` encompasses the instantiated `TemplateVariable` of the associated `GroundRandomVariable` and the `TemplateVariables` of the parents, if any. `LocalProbabilisticModels` are also contained in `GroundProbabilisticNetworks` and correspond to their parametric part. Recall that template factors describe probability distribution families at the type level. Distribution families form a set of distributions of the same type (e.g. normal distribution) but different parameter values (e.g. mean and standard deviation). Regarding `LocalProbabilisticModel`, an instantiated `TemplateFactor` is complemented by a `ProbabilityDistribution`-entity. The `ProbabilityDistribution` is part of the *ProbDist* metamodel (discussed in section 5.2.6) and corresponds to a distribution with fixed parameter values. Roughly speaking, a `LocalProbabilisticModel` instantiates a `TemplateFactor` by connecting the factor with a probability distribution of the same distribution type with fixed parameter values.

Recall formula (2.15) from page 44 that defines $\mathcal{X}_\kappa[\aleph]$ as the set of all ground random variables $\mathcal{V}(\gamma)$ w.r.t. some object skeleton κ. A ground random variable $\mathcal{V}(\gamma)$ is captured in our metamodel by `GroundRandomVariable` where `TemplateVariable` corresponds to the instantiated template $\mathcal{V}$ and the set of referenced `EObjects` correspond to the applied objects γ. By iterating over all `LocalProbabilisticNetworks` that are contained in `GroundProbabilisticNetwork`, we obtain $\mathcal{X}_\kappa[\aleph]$ by considering all `GroundRandomVariables` referenced by `LocalProbabilisticNetworks`. Thus, we adhere to the formal semantics of ground Bayesian networks. For better readability, we repeat the definition 16 of a ground Bayesian network:

Definition 16 (Ground Bayesian Network). *A Ground Bayesian Network $\mathcal{B}_\kappa^{M_{Plate}}$ is generated by a plate model M_{Plate} and object skeleton κ as follows:*

$$\forall \mathcal{V}(U_1, ..., U_n) \in \aleph, \forall \gamma \in \Gamma_\kappa[\mathcal{V}] : \exists_{=1} \mathcal{V}(\gamma) \in \mathcal{X}_\kappa[\aleph] \tag{2.16}$$

where $\gamma := (U_1 \mapsto u_1, \ldots, U_n \mapsto u_n)$ and for all template parents $\mathcal{V}_{Pa} \in Pa(\mathcal{V})$ of ground random variable $\mathcal{V}(\gamma)$ there exist an instantiated CPD: $P(\mathcal{V}(\gamma) \mid \mathcal{V}_{Pa_1}(\gamma), \ldots, \mathcal{V}_{Pa_k}(\gamma))$.

The basic semantics of a plate model are captured by the `template` package. We discuss in section 5.3 how we use the architecture model (i.e. PCM models) to represent the object skeleton κ. Finally, `TemplateFactors` (or in this case `ProbabilisticTemplateFactors`) and `ProbabilityDistributions` describe the instantiated CPDs $P(\mathcal{V}(\gamma) \mid \mathcal{V}_{Pa_1}(\gamma), \ldots, \mathcal{V}_{Pa_k}(\gamma))$. The requirement of complete CPD instantiations and the requirement of formula (2.16) can be enforced by OCL constraints at the meta-level (as discussed in section 5.2.3.1).

5.2.4.2. Applying the Static Environment Model

In this section, we illustrate how to model an instance of the `static` metamodel package to capture the static environment. Again, we use the DeltaIoT example system as a use case for which we have already described how to model template variables and template factors (see section 5.2.3.2). Now, we illustrate how the modelled template variables and template factors are instantiated in a model of the `static` metamodel package. The resulting model represents the static environment, i.e. $\mathcal{B}_0$ of the environmental dynamics $(\mathcal{B}_0, \mathcal{B}_{\rightarrow})$.

Suppose we have an architecture model that describes the DeltaIoT system as depicted on Figure 1.2 of section 1.5.2. The role of the architecture model in conjunction with the instantiation process of template variables and template factors is discussed in section 5.3. For simplicity, we therefore assume that the architecture model of DeltaIoT contains the objects of the respective object classes of the template variables, namely *Mote* and *WirelessLink* (see Figure 5.3).

Recall from section 5.2.3.2 that we defined three template variables that represent the environmental variables of the DeltaIoT system, namely mote activation MA, wireless interference WI and the SNR SNR. Each template variable is instantiated multiple times in the DeltaIoT architecture model. For example, the MA template variable is instantiated 17 times because there are 17 motes. Similarly, WI and SNR templates are instantiated for each wireless link. Consequently, this results in a large ground Bayesian network describing the static environment of the DeltaIoT system. Therefore, consider Figure 5.7 which depicts only an excerpt of the ground Bayesian network.

For the sake of illustration, the ground Bayesian network depicts only the instantiation of the template variable MA of a single mote (mote 10 of the DeltaIoT system which is equipped with a passive infrared sensor, see Figure 1.2) and template variables WI and SNR for a single wireless link (the wireless link which connects mote 10 and mote 5, see Figure 1.2). It can be seen that the model consists of the root element `BasicDistributionRepo ProbabilisticModelRepository` which contains a single ground Bayesian network, namely the `GroundNetwork` instantiating a `GroundProbabilisticNetwork`. The `GroundNetwork`-object encompasses the instantiated template variables (i.e. the ground random variables) and the set of instantiated template factors characterising the distribution of each ground random variable.

The ground random variables are grouped into `LocalProbabilisticNetworks`. For instance, each ground random variable instantiating the template variable MA forms an

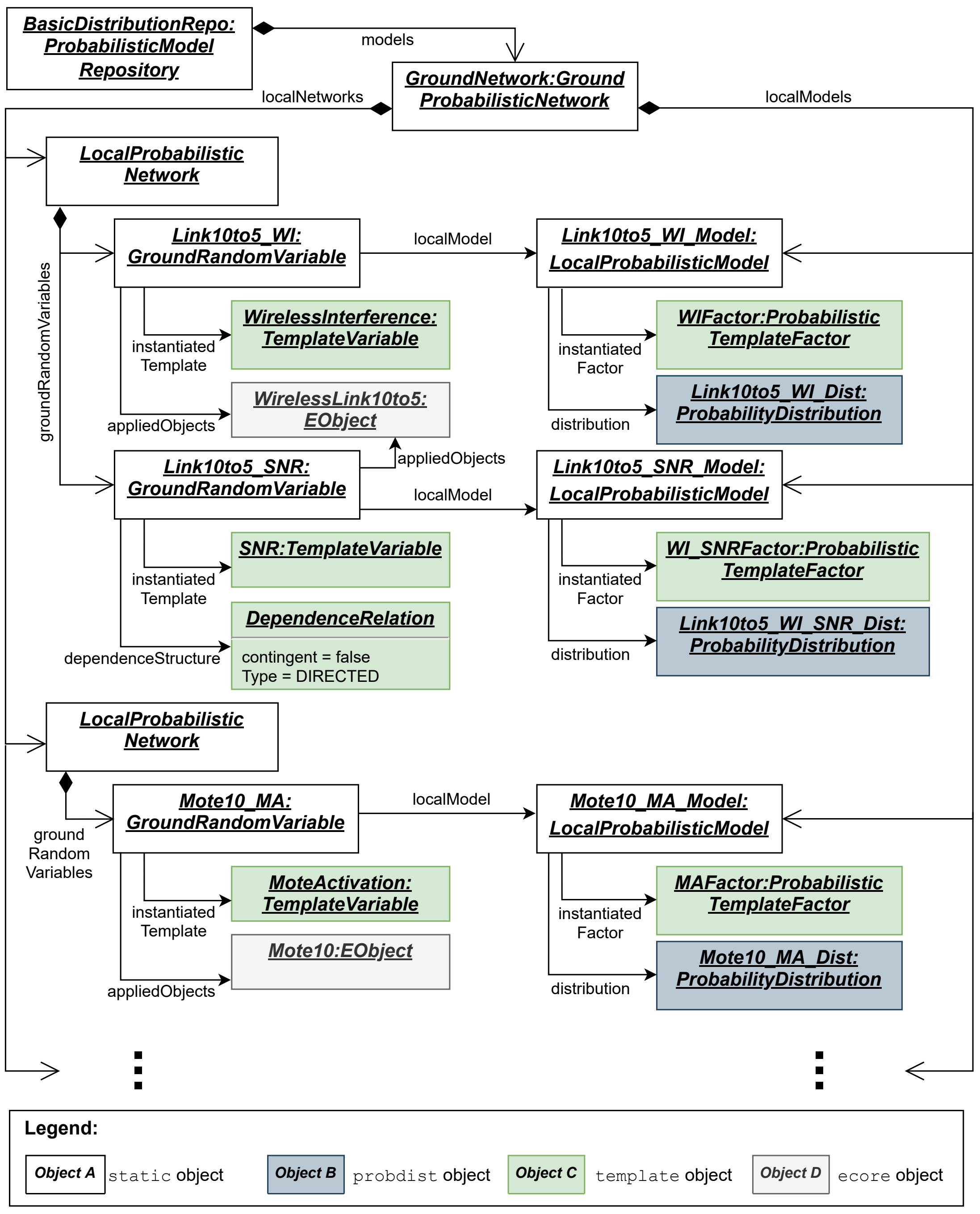

BasicDistributionRepo: ProbabilisticModel Repository
models
GroundNetwork:Ground ProbabilisticNetwork
localNetworks
localModels
LocalProbabilistic Network
groundRandomVariables
Link10to5_WI: GroundRandomVariable
localModel
Link10to5_WI_Model: LocalProbabilisticModel
instantiated Template
WirelessInterference: TemplateVariable
appliedObjects
WirelessLink10to5: EObject
instantiated Factor
WIFactor:Probabilistic TemplateFactor
distribution
Link10to5_WI_Dist: ProbabilityDistribution
appliedObjects
Link10to5_SNR: GroundRandomVariable
localModel
Link10to5_SNR_Model: LocalProbabilisticModel
instantiated Template
SNR:TemplateVariable
dependenceStructure
DependenceRelation
contingent = false
Type = DIRECTED
instantiated Factor
WI_SNRFactor:Probabilistic TemplateFactor
distribution
Link10to5_WI_SNR_Dist: ProbabilityDistribution
LocalProbabilistic Network
ground Random Variables
Mote10_MA: GroundRandomVariable
localModel
Mote10_MA_Model: LocalProbabilisticModel
instantiated Template
MoteActivation: TemplateVariable
appliedObjects
Mote10:EObject
instantiated Factor
MAFactor:Probabilistic TemplateFactor
distribution
Mote10_MA_Dist: ProbabilityDistribution
Legend:
Object A static object
Object B probdist object
Object C template object
Object D ecore object

Figure 5.7.: An excerpt of the ground Bayesian network model of the `static` metamodel package applied to the DeltaIoT system.

individual local network such that for each of them there is a corresponding `LocalProbabilisticNetwork`-object. Similarly, for each instantiation of template variables WI and SNR, there is a corresponding `LocalProbabilisticNetwork`-object. Since WI and SNR are stochastically correlated, they form a local probabilistic network and are grouped for this reason. Each ground random variable refers to the instantiated template variable, the set of applied objects and the dependency structure, if any. For example, the `Mote10_MA`

`GroundRandomVariable` instantiates the template variable MA. It also references the corresponding mote object (i.e. mote 10 of the DeltaIoT system) in the architectural model for which the MA is instantiated. Equivalently, the `Link10to5_WI` `GroundRandomVariable` and `Link10to5_SNR` `GroundRandomVariable`-objects instantiate the template variables WI and SNR, respectively. Both refer to the `WiressLink-EObject` because both template variables are instantiated together for the same object. However, note that the `Link10to5_SNR` `GroundRandomVariable` is also referencing `DependenceRelation`-object that forms its dependency structure as SNR depends on WI.

Finally, each ground random variable references a `LocalProbabilisticModel`-object that instantiates a template factor. Recall from section 5.2.3.2 that regarding the DeltaIoT system we have three template factors, namely ξ_{MA}, ξ_{WI} and ξ_{SNR}, each of which is instantiated multiple times. For example, in Figure 5.7 there are three `LocalProbabilisticModel`-objects instantiating ξ_{MA}, ξ_{WI} and ξ_{SNR}. Furthermore, each `LocalProbabilisticModel`-object refers to a `ProbabilityDistribution`-object that is part of the *ProbDist* metamodel (more precisely, the `distributionfunction` package). Recall that template factors specify a distribution family (e.g. the multinomial distribution family), which must be concretised as soon as they are instantiated by a `LocalProbabilisticModel`. This concretisation is done by the respective `ProbabilityDistribution`-object, i.e. by determining the parameters. The representation of `ProbabilityDistribution`-objects are discussed in section 5.2.6. As mentioned earlier, each `LocalProbabilisticModel` is associated with a ground random variable and models its concrete probability distribution.

5.2.5. Modelling the Dynamic Environment

So far we discussed how to model type-level descriptions of random variables (i.e. template variables and factors) and how they can be instantiated to generate ground Bayesian networks which form the static environment of the system, i.e. the $\mathcal{B}_{\rightarrow}$ part of the environmental dynamics $(\mathcal{B}_0, \mathcal{B}_{\rightarrow})$. However, the primary objective of this thesis is to evaluate adaptation strategies of self-adaptive systems in environments that indicate stochastic and temporal behaviour. This section discusses the part of *EnvDyn* that adds dynamic modelling capabilities to the metamodel. The `dynamic` package contains the corresponding metaclasses and is depicted on Figure 5.8.

5.2.5.1. Formal Semantics

The root of the `dynamic` package forms the `DynamicBehaviourRepository` that consists of a set of `DynamicBehaviourExtensions`. Recall that BNs form joint probability distributions over a set of random variables. Each DynamicBehaviourExtensions is associated with a single `GroundProbabilisticNetwork` to indicate that the (static) ground Bayesian network is extended to a DBN. Consider the definition of a DBN $(\mathcal{B}_0, \mathcal{B}_{\rightarrow})$ that is defined as a tuple consisting of an initial BN $\mathcal{B}_0$ and 2-TBN $\mathcal{B}_{\rightarrow}$. Semantically, the entity `DynamicBe-haviourExtension` refers to a 2-TBN, where the referenced `GroundProbabilisticNetwork`

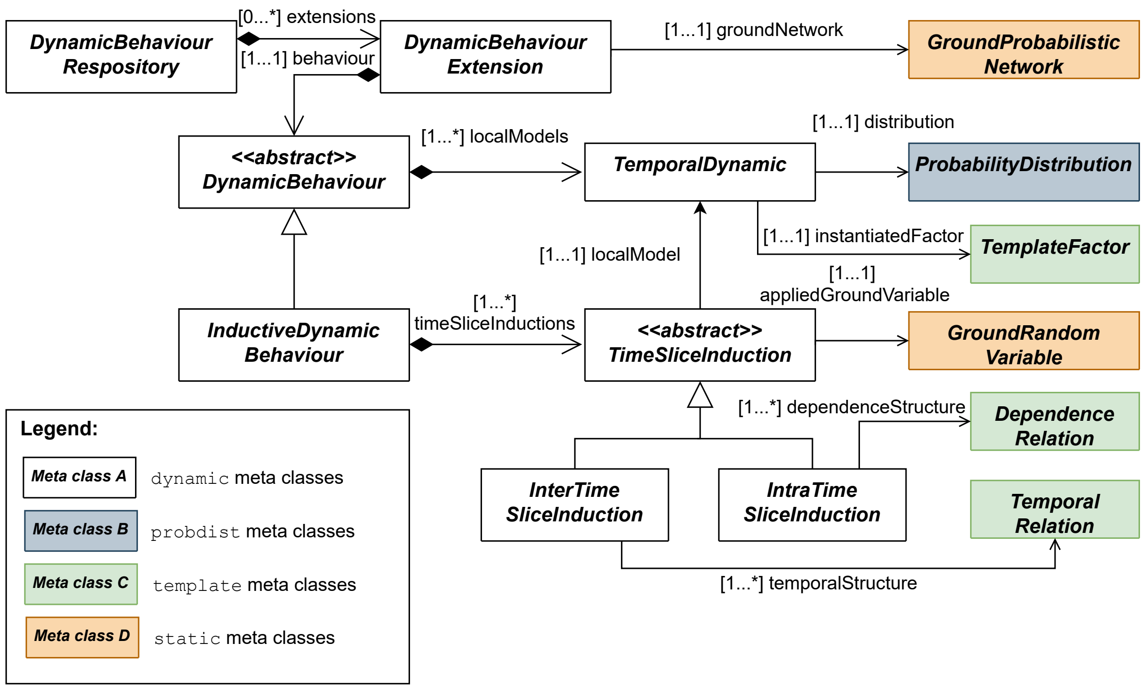


Figure 5.8.: The `dynamic` package of the metamodel for modelling dynamic Bayesian networks.

corresponds to $\mathcal{B}_0$, i.e. the BN to be extended. The specific behaviour description is captured by the abstract metaclass `DynamicBehaviour` which is referenced by `DynamicBe-haviourExtension`.

`DynamicBehaviour` is abstract to keep the metamodel extensible for other types of behaviour. Currently, there is only a single extension: `InductiveDynamicBehaviour`. The `Inductive-DynamicBehaviour`-entity refers to a set of `TimeSliceInductions` which is an abstract metaclass. `TimeSliceInductions` establish a temporal dependency between two random variables (or `GroundRandomVariables`) and describe inductively from time instance t to $t+1$ how they stochastically evolve. There are two types of `TimeSliceInductions`, namely `InterTimeSliceInductions` and `IntraTimeSliceInductions`. The metaclass `TimeSliceIn-duction` references a single `GroundRandomVariable` which is either complemented by `De-pendenceRelations` (if the sub-metaclass corresponds to `IntraTimeSliceInduction`) or `TemporalRelations` (if the sub-metaclass corresponds to `InterTimeSliceInduction`). Re-garding `InterTimeSliceInductions`, the `GroundRandomVariable` (referenced by the super-metaclass) is complemented by a set of `TemporalRelations` (i.e. either `PersistenceRe-lation` or `TimeSliceRelation`) where the target of each relation is referencing the same `TemplateVariable` as instantiated by the `GroundRandomVariable`.

Example 2. Consider Figure 5.9 where a simple 2-TBN is shown. In this case, a `In-terTimeSliceInduction` is associated with a `GroundRandomVariable` (i.e. X') and refers to a `PersistenceRelation` (i.e. $X \rightarrow X'$) and a `TimeSliceRelation` (i.e. $Y \rightarrow X'$). In terms of `IntraTimeSliceInductions`, the `GroundRandomVariable` is complemented by a set of (non-temporal) `DependenceRelations` where the target of each relation references

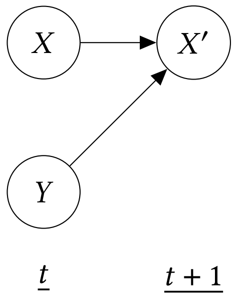


Figure 5.9.: An example structure of a two-time-slice Bayesian network.

the same `TemplateVariable` as instantiated by `GroundRandomVariable` (just as for `InterTimeSliceInductions`). ■

The semantics of `InterTimeSliceInductions` and `IntraTimeSliceInductions` are equivalent to the semantics of inter-time-slice edges and intra-time-slice edges in DBNs. In the `dynamic` package, persistence edges are considered as special cases of inter-stime-slice edges in which the source and target random variables are the same.

We have deliberately reserved the discussion of `TemporalDynamic`-entities as they complement `TimeSliceInductions` with probabilistic models. `TemporalDynamics` are comparable to `LocalProbabilisticModels` from the `static` package as they describe the probability distributions of `TimeSliceInductions` (e.g. $P(X' \mid X, Y)$ from the example 2-TBN shown in Figure 5.9). Therefore, they instantiate a type-level `TemplateFactor` from the `template` package complemented by a `ProbabilityDistribution` instance with fixed parameter values (just as `LocalProbabilisticModels`).

In summary, the formal semantics of the `dynamic` package are based on the notion of 2-TBNs where existing BNs are extended to DBNs. The semantics of inter-time-slice, intra-time-slice and persistence edges are reused to establish the temporal connection between random variables. Equivalently to the `static` package, template factors are instantiated to inductively describe the probability distributions of how random variables change over time.

5.2.5.2. Applying the Dynamic Environment Model

In this section, we illustrate the application of the `dynamic` metamodel package. Again, we use the DeltaIoT system as an example system and add the dynamic environment model to the model instances of sections 5.2.3.2 and 5.2.4.2 presented so far. All three model instances form the environmental dynamics $(\mathcal{B}_0, \mathcal{B}_\rightarrow)$.

Since the environment of the DeltaIoT system is quite complex, we have again only presented an excerpt from the dynamic environment model for the sake of clarity. We also discuss the temporal extensions (described by the dynamic environment model) for the

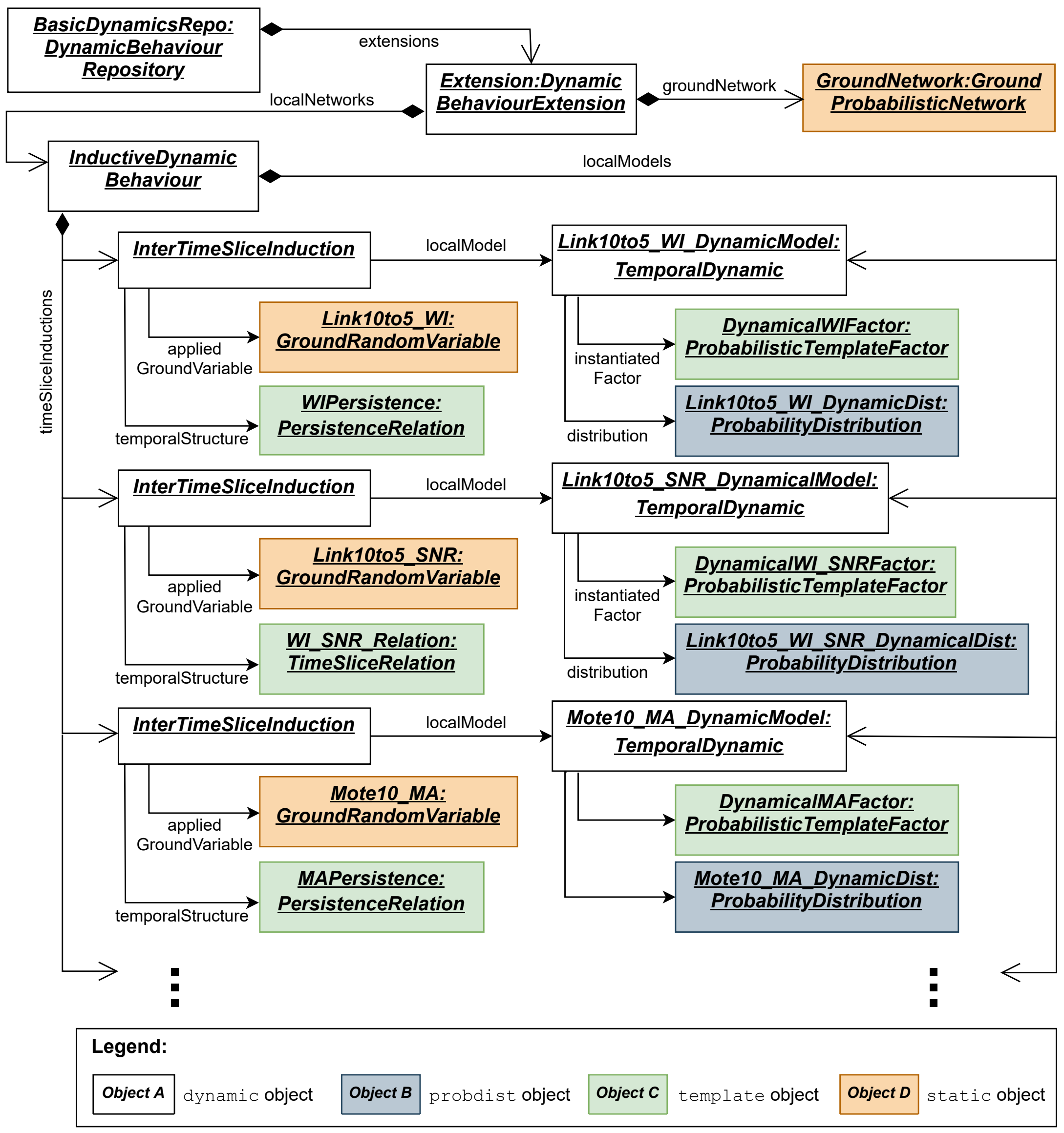

BasicDynamicsRepo: DynamicBehaviour Repository
extensions
Extension:Dynamic BehaviourExtension
groundNetwork
GroundNetwork:Ground ProbabilisticNetwork
localNetworks
InductiveDynamic Behaviour
localModels
timeSliceInductions
InterTimeSliceInduction
localModel
Link10to5_WI_DynamicModel: TemporalDynamic
applied GroundVariable
Link10to5_WI: GroundRandomVariable
temporalStructure
WIPersistence: PersistenceRelation
instantiated Factor
DynamicalWIFactor: ProbabilisticTemplateFactor
distribution
Link10to5_WI_DynamicDist: ProbabilityDistribution
InterTimeSliceInduction
localModel
Link10to5_SNR_DynamicalModel: TemporalDynamic
applied GroundVariable
Link10to5_SNR: GroundRandomVariable
temporalStructure
WI_SNR_Relation: TimeSliceRelation
instantiated Factor
DynamicalWI_SNRFactor: ProbabilisticTemplateFactor
distribution
Link10to5_WI_SNR_DynamicalDist: ProbabilityDistribution
InterTimeSliceInduction
localModel
Mote10_MA_DynamicModel: TemporalDynamic
applied GroundVariable
Mote10_MA: GroundRandomVariable
temporalStructure
MAPersistence: PersistenceRelation
DynamicalMAFactor: ProbabilisticTemplateFactor
Mote10_MA_DynamicDist: ProbabilityDistribution
Legend:
Object A dynamic object
Object B probdist object
Object C template object
Object D static object

Figure 5.10.: An excerpt of the dynamic environment model of the `dynamic` metamodel package applied to the DeltaIoT system.

same ground random variables of the DeltaIoT system presented in section 5.2.4.2. The excerpt of the model is depicted on Figure 5.10.

The model consists of the root object `BasicDynamicsRepo` and defines a set of `extensions` that refer to `DynamicBehaviourExtensions`. In this case, there is only a single `DynamicBe-hvaiourExtension`-object, namely `Extension`. Recall that `DynamicBehvaiourExtensions` are associated with a single ground Bayesian network that is to be extended to include dynamic behaviour (in our case, this is the ground Bayesian network of the DeltaIoT system from section 5.2.4.2).

To relate the ground Bayesian network with its respective dynamic behaviour extension, the `Extension`-object references an `InductiveDynamicBehaviour`-object that models the concrete dynamic behaviour extensions. More specifically, `InductiveDynamicBehaviour` references a set of `TimeSliceInductions` (i.e. attribute `timeSliceInductions`) and a set of `TemporalDynamics` (see attribute `localModels`). The former specifies the ground random variables of the `GroundNetwork`-object indicating dynamic behaviour; the latter models the probability distribution that describes the stochastic evolution, i.e. the probabilistic description of the dynamic behaviour. This structural split is similar to what we have seen for objects of `GroundProbabilisticNetworks` where a set of `GroundRandomVariables` specify the instantiated template variables and a set of `LocalProbabilisticModels` determine their probability distributions.

Each `TimeSliceInduction` refers to a single `GroundRandomVariable` that is to be extended and, depending on the type of `TimeSliceInductions`, the respective `Relation`-object from the `template` metamodel package. In terms of the DeltaIoT system there are no `IntraTimeSliceInductions` but only `InterTimeSliceInductions`. For example, in Figure 5.10 there is an `InterTimeSliceInduction` that is applied to `GroundRandomVariable`-object `Link10to5_WI`. Moreover, the `InterTimeSliceInductions`-object refers to the `PersistenceRelation`-object `WIPersistence` indicating the temporal structure (see attribute `temporalStructure`). This means that the basic random variable `Link10to5_WI` evolves probabilistically and persistently, i.e. there is a stochastic correlation between the ground random variable $X_{Link10to5\_WI}$ at time t and the basic random variable $X'_{Link10to5\_WI}$ at time $t + 1$. As another example, there is an `InterTimeSliceInduction` that is applied to `GroundRandomVariable`-object `Link10to5_SNR`. In this case, the `temporalStructure` is given by the `TimeSliceRelation`-object `WI_SNR_Relation` indicating that the ground random variable `Link10to5_SNR` has no persistent behaviour but is stochastically dependent on ground random variable `Link10to5_WI`; that is, there is a stochastic correlation between the ground random variable $X_{Link10to5\_WI}$ at time t and the basic random variable $X'_{Link10to5\_SNR}$ at time $t + 1$.

Just as `InterTimeSliceInductions` describe stochastic correlations of ground random variables inductively from t to $t+1$, the concrete probability distribution of such correlations is modelled by `TemporalDynamic`-objects. For instance, consider the `TemporalDynamic`-object `Link10to5_WI_DynamicModel` which is related to (see the `localModel` attribute of `InterTimeSliceInduction`-objects) the `InterTimeSliceInduction`-object extending the ground random variable `Link10to5_WI`. The object `Link10to5_WI_DynamicModel` refers to the template factor it instantiates (i.e. `DynamicalWIFactor`) and is complemented by an instance of `ProbabilityDistribution` (i.e. the object `Link10to5_WI_DynamicDist`) of the metamodel package `distributionfunction`, which specifies the concrete probability distribution. Effectively, this is equivalent to the concept of `LocalProbabilisticModels` discussed in section 5.2.4.1 and illustrated in section 5.2.4.2.

5.2.6. Modelling Probability Distributions

This section discusses the *ProbDist* metamodel, which is not explicitly part of the *EnvDyn* metamodel but provides the modelling concepts to describe probability distributions. The

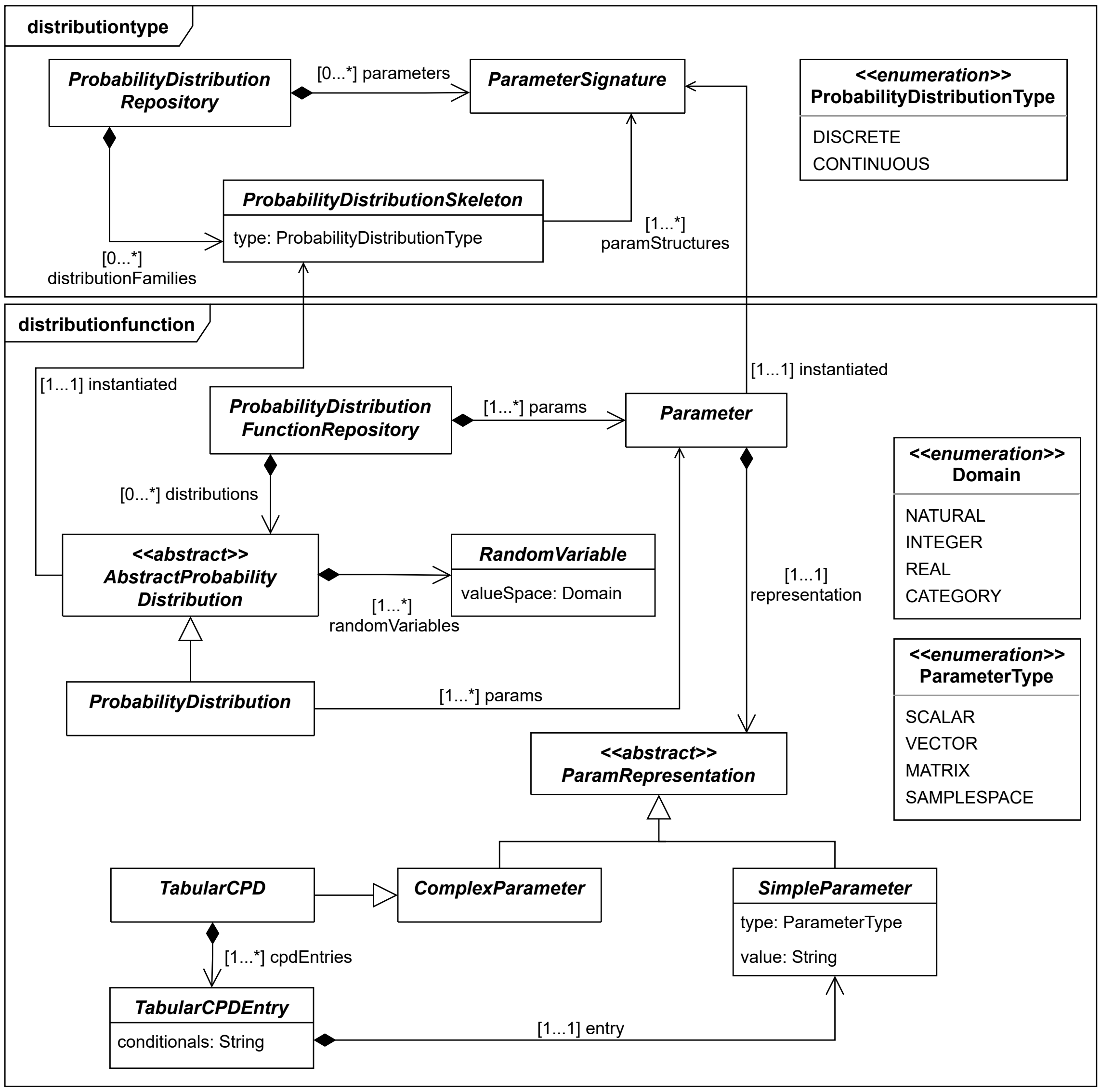


Figure 5.11.: Overview of the *ProbDist* metamodel.

metamodel is divided into two packages, namely `distributiontype` and `distribution-function`. An overview of the entire metamodel is depicted in Figure 5.11.

The `distributiontype` package provides modelling concepts to describe probability distributions at type-level; the `distributionfunction` package instantiates modelling entities of the `distributiontype` package to describe manifestations of probability distributions. Note that, just as in the *EnvDyn* metamodel, we have omitted attributes such as name or ID in Figure 5.11.

5.2.6.1. Type-Level Probability Distributions

The core entities of the distributiontype package form the ProbabilityDistribution-Skeleton and ParameterSignature. Instances of both metaclasses are collected in the ProbabilityDistributionRepository metaclass which corresponds to the root of the package. ProbabilityDistributionSkeletons model distribution families (e.g. exponential, uniform or multinomial distributions) that are distinguished by their names. Additionally, the type attribute refers to an enumeration called ProbabilityDistributionType. ProbabilityDistributionType enumerates two constants, namely DISCRETE and CONTINUOUS. Therefore, a ProbabilityDistributionSkeleton instance is characterised as being either continuous or discrete; that is, the value space of the considered random variables is either discrete or continuous. Each ProbabilityDistributionSkeletons references one or many ParameterSignatures that parameterise the distribution.

Instances of ParameterSignature describe parameters such as the mean, standard deviation or covariance matrix without specifying the actual value. Based on these modelling concepts, type-level distribution can be modelled. For example, a family of normal distributions is modelled by creating an instance of ProbabilityDistributionSkeleton named (say) "normal distribution" and type CONTINUOUS referencing two instances of ParameterSignature. One instance of parametersignature describes the mean and another instance of parametersignature describes the standard deviation (assuming a one-dimensional distribution).

5.2.6.2. Instance-Level Probability Distributions

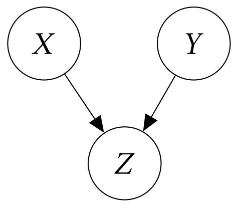
The distributionfunction package provides the modelling concepts to instantiate type-level distributions by adding fixed values to the purely parametric descriptions (i.e. ParameterSignatures).

The root of the package forms the ProbabilityDistributionFunctionRepository that references a set of Parameters and ProbabilityDistributions. Each ProbabilityDistribution references a set of RandomVariables over which the distribution is defined. A RandomVariable is characterised by its value space (see valueSpace attribute in Figure 5.11). A value space has a domain captured by the enumeration Domain with constants NATURAL (i.e. $\mathbb{N}$), INTEGER (i.e. $\mathbb{Z}$), REAL (i.e. $\mathbb{R}$) and CATEGORY. A ProbabilityDistribution entity is associated with a particular ProbabilityDistributionSkeleton that models an instantiation of the skeleton (or a manifestation of a particular distribution w.r.t. a distribution family). To complement the type-level parameters of the instantiated distribution, a set of Parameters are referenced.

Just like ProbabilityDistributions instantiate ProbabilityDistributionSkeletons, Parameters instantiate and supplement ParameterSignatures with fixed values. However, an instance of ParameterSignature can describe parameters with different representations. Such representations range from simple scalar values to matrices or tables (e.g. to

$Pr(X = x_1)$	$Pr(X = x_2)$
0.5	0.5

$Pr(Y = y_1)$	$Pr(Y = y_2)$
0.3	0.7



Con.	$Pr(Z = z_1)$	$Pr(Z = z_2)$
x_1, y_1	0.8	0.2
x_1, y_2	0.3	0.7
x_2, y_1	0.5	0.5
x_2, y_2	0.4	0.6

Figure 5.12.: Example BN with tabular-based parameter representation

describe higher-dimensional multinomial distributions). Therefore, a `Parameter` has a representation captured by `ParamRepresentation`

The abstract `ParamRepresentation` metaclass has two sub-metaclasses, namely `SimplePa-rameter` and `ComplexParameter`. The `SimpleParameter` entity has a `type` and `value` attribute. Former, describes the `ParameterType` by an enumeration with constants `SCALAR`, `VECTOR`, `MATRIX` and `SAMPLESPACE`. The `value` attribute of `SimpleParameter` corresponds to the string-based representation of the value w.r.t. to its specified `type` (for each `type` a certain string pattern is expected to parse the string value). For a value of type `SAM-PLESPACE`, for example, the string pattern refers to a set of pairs where each pair comprises a categorical value and probability. The abstract metaclass `ComplexParameter` describes more sophisticated representations. Currently, there is only one `ComplexParameter` supported: `TabularCPD`. A `TabularCPD`-entity represents table-like parameters (such as for higher-dimensional multinomial distributions). Thus, a `TabularCPD` instance contains a set of `TabularCPDEntries` where each `entry` defines a `conditional` (represented as a string) and refers to exactly one `SimpleParameter`. The referenced `SimpleParameter` of each `en-try` indicates the distribution of some values given the conditional of the entry. Because `TabularCPD` represents tables for higher-dimensional multinomial distributions, the value space of the random variables over which the distribution is defined is discrete. Thus, the `type` attribute of `SimpleParameter` (referenced by each `entry`) must be `SAMPLESPACE`.

Example 3. Consider Figure 5.12 which depicts an example BN with a tabular-based parameter representation.

It shows a BN that describes a multinomial joint distribution over the random variables X, Y and Z. Recall that the joint distribution of a BN factorises w.r.t. some graph $\mathcal{G}$. Thus, given the graph in Figure 5.12, the distribution factorises to $P(X, Y, Z) = P(X) \cdot P(Y) \cdot P(Z \mid X, Y)$. Based on the factorisation the distributions $P(X)$ and $P(Y)$ are described individually, i.e. each by a `SimpleParameter` instance of `type` `SAMPLESPACE` with $(x_1, 0.5)$, $(x_2, 0.5)$ as `value` for $P(X)$ and $(y_1, 0.3), (y_2, 0.7)$ as `value` for $P(Y)$, respectively. The distribution $P(Z \mid X, Y)$ is represented by a `ComplexParameter` (or rather `TabularCPD`) instance with four `TabularCPDEntries` where the `conditionals` refer to (x_1, y_1), (x_1, y_2), (x_2, y_1) and (x_2, y_2). Each `TabularCPDEntry` is associated with a `SimpleParameter` describing the corresponding distribution, e.g. $P(Z \mid X = x_1, Y = y_1)$ with `value` $(z_1, 0.8), (z_2, 0.2)$. ■

5.2.7. Discussion

In section 5.1, we enumerated several requirements that our metamodel must provide. Now, we discuss how the *EnvDyn* metamodel addresses the requirements.

Domain-Independence The rationale for the domain-independence requirement is based on the fact that AI components can be deployed in any domain, so it is necessary to describe the environmental variables or factors that force AI components to make erroneous predictions. The *EnvDyn* metamodel addresses this requirement by integrating modelling concepts from the field of probabilistic template-based models. Template variables and factors enable the specification of type-level random variables that can be instantiated in arbitrary domains. We have discussed in section 5.2.3 how the metamodel *EnvDyn* conforms to the formal semantics of template variables and factors. Thus, the *EnvDyn* metamodel exhibits the same domain-independent modelling capabilities that satisfy the domain independence requirement.

Architecture description language agnostic The *EnvDyn* metamodel is intended to be applicable to any architecture model (w.r.t. a particular ADL), as some architecture models are more suitable for some domains, i.e. enable the modelling of more domain-specific characteristics that are more meaningful in terms of the instantiation of the environmental dynamics. The *EnvDyn* metamodel is ADL-agnostic because each template variable is associated with a set of `EObjects` indicating the set of domain-specific objects for which the template variable is instantiated (see section 5.2.4). An `EObject` in turn forms the root of each model object in the EMF. Roughly speaking, the *EnvDyn* metamodel can be applied to any *Ecore*-based ADL; that is, any ADL which was developed in the context of EMF.

Stochastic dynamics Technically, the environmental dynamics are formally considered as a DTMC to capture the probabilistic evolution of the environmental states. However, modelling DTMCs conventionally (e.g. with state machine-based models) can be quite tedious. In terms of the *EnvDyn* metamodel, we represent the environmental dynamics as DBNs which (if multinomial distributed) form DTMCs. In DBNs, the stationary and Markov assumptions allow an inductive representation of the state changes over time. This requires modelling only the initial distribution of the environmental states (i.e. the distribution over $\mathcal{E}_0$, captured by a BN) and an inductive transition step of the environmental states at time t to $t + 1$ (i.e. the transition function $t_{\mathcal{E}}$, captured by a 2-TBN). In section 5.2.5, we discussed how we integrated the semantics of DBNs in the *EnvDyn* metamodel to model the stochastic dynamics of the environment with DBNs.

Stochastic correlations Recall that an environmental state is a tuple $E := (e_1, ..., e_n)$ consisting of environmental variables e_i. Stochastic correlations refer to stochastic dependencies between the environmental variables e_i that manifest themselves statically and dynamically. Such correlations are taken into account in the metamodel *EnvDyn* through Bayesian modelling. In Bayesian modelling, correlations (or (in-)dependence assumptions) between random variables are expressed by DAGs that form the core of Bayesian models. Such DAGs can also be modelled in the *EnvDyn* metamodel (see section 5.2.3) and allow the representation of correlations between environmental variables.

Compactness The compactness requirement refers to the compact modelling of probabilities or their parameters. In DTMCs, for example, the transition probabilities are captured in a probability transition matrix in which each entry corresponds to the probability of transition to another state given the current. From a modelling perspective, such matrix-based representations are hard to manage as the size of entries grows quadratically in the size of the states. Thus, for large state spaces, such matrices are intractable to model. The *EnvDyn* metamodel tackles the compactness requirements by the decomposability property of Bayesian models, i.e. the decomposition of the probability distribution of Bayesian models (according to some DAG $\mathcal{G}$) into local CPDs. For instance, let $P(X_1, \ldots, X_n)$ be a joint probability distribution over a set of random variables $(X_1, ..., X_n)$. If the distribution is multinomial distributed, the set of parameters to fully describe $P(X_1, \ldots, X_n)$ is $\theta = (\prod_{i \in \{1,...,n\}} |Val(X_i)|) - 1$. If the distribution is represented as BN and factorises to some graph $\mathcal{G}$, the number of parameters to estimate corresponds to the sum of parameters of each local CPD $P(X_i \mid Pa_{\mathcal{G}}(X_i))$. Let CPD_i be the number of required parameters of local CPD $P(X_i \mid Pa_{\mathcal{G}}(X_i))$, then the number of parameters (to fully describe $P(X_1, \ldots, X_n)$) is $\theta = \sum_{i \in \{1,...,n\}} CPD_i$. Depending on the structure of graph $\mathcal{G}$, this can tremendously reduce the number of parameters to describe a given probability distribution; this is illustrated in [105, P.5]. Because the *EnvDyn* metamodel enables the modelling of BNs and DBNs over a set of random variables, the decomposability property applies equally and allows modelling parameters of local CPDs instead of the entire joint probability distribution. For high-dimensional probability distributions, however, where few (in)dependence assumptions can be made, the number of descriptive parameters can still be high and hard to model. Nevertheless, this can be tackled by more course-grained discretisation of the considered random variables.

Discretisation level Finally, the discretisation level requires the flexible modelling of discrete value spaces of random variables. We discussed in section 5.2.6 how the *EnvDyn* metamodel integrates the *ProbDist* metamodel to specify multinomial distributions (recall that we assume only multinomial distributed BNs and DBNs, respectively). The *ProbDist* metamodels enable the modelling of multinomial distributions where the value spaces of the considered random variables are described by tabular-based parameters in which the value spaces of random variables. More precisely, the value ranges are described by a string in which the desired number of discrete states can be specified.

5.3. Instantiating Environmental Dynamics in Domain-Specific Contexts

In the last section, we presented the *EnvDyn* metamodel to model the environmental dynamics. One of the key concepts of the *EnvDyn* metamodel is template variables that describe type-level random variables. Although template variables describe random variables that share common semantics (i.e. the value space and distribution family), it becomes more meaningful after instantiation for a set of objects of a particular domain. After instantiation, the template variable is complemented by the actual distribution of the random variable (described by the instantiation of a corresponding template factor). We briefly discussed the instantiation process in which template variables are instantiated for an object skeleton κ. We also mentioned that κ refers to the architecture model. More specifically, the objects of κ are covered by the objects defined in the architecture model $M_{\mathcal{C}}$, i.e. $\kappa \subseteq M_{\mathcal{C}}$. This again shows the importance of the selection of an appropriate ADL. The architecture model must include all relevant objects from which an object skeleton κ can be derived. Based on κ an *EnvDyn* instance or environment model is generated.

In the following, we discuss in more detail how instances of the *EnvDyn* metamodel are created w.r.t. an architecture model containing all relevant objects to capture κ. We consider an *EnvDyn* instance as $M_{\mathcal{E}}$ (i.e. $M_{\mathcal{E}} = (\mathcal{B}_0, \mathcal{B}_{\rightarrow})$) and an architecture model as $M_{\mathcal{C}}$. An environment and architecture model summarised by a tuple $\mathcal{M} := (M_{\mathcal{C}}, M_{\mathcal{E}})$ indicates that $M_{\mathcal{E}}$ is generated by $M_{\mathcal{C}}$. Therefore, we start to present the general process where template variables and factors are instantiated in the architecture model. Afterwards, we present a semi-automated approach for generating the probabilistic structure of an environment model $M_{\mathcal{E}}$ induced by the instantiated template variables and factors in the architecture model $M_{\mathcal{C}}$. We present an annotation-based approach for instantiating template variables based on so-called *EMF Profiles* [110].

5.3.1. Instantiation of Template-based Structures

The most straightforward way to instantiate template variables is to create the architecture model and template variables (or rather template variable definitions, as discussed in section 5.2.3). When the template variables are defined, they are instantiated in the architecture model (or based on the architecture model). Recall that after the template variables are modelled (see the `template` package of *EnvDyn*), they can be instantiated in the `static` package of the *EnvDyn* metamodel. Note that the term "instantiation" is ambiguous, as template variables are instantiated in the `static` part of *EnvDyn* on the one hand, and instantiated in the architectural model on the other; however, it describes the same concept. The instantiation of a template variable is a two-part process in which a ground random variable is first created to indicate an instantiation of a given template variable in the `static` package of *EnvDyn*. Afterwards, the ground random variable must be complemented by a set of objects that indicate the domain-specific objects for which the template is instantiated. Trivially, this can be achieved by manually adding the corresponding objects of the architecture model to the ground random variable. This is repeated for each template variable that is to be instantiated. Finally, the remaining elements of the `static` and `dynamic` packages are modelled to obtain the BN and DBN, respectively.

The manual instantiation of template variables is straightforward and directly aligned with the modelling process of environment models. Depending on the complexity of the environment model, however, the manual instantiation process might be tedious and error-prone. Therefore, the question arises as to which extent the manual instantiation process can be automated. In the next section, we present a semi-automated approach where the structural part of the environment model is automatically generated.

5.3.2. Semi-Automated Generation of the Structural Environment Model by Annotation-based Instantiation

In the last section, we discussed the manual instantiation process of template variables. Because the process might be time-consuming and error-prone, we now present a semi-automated approach to generate the structural environment model. By structural environment model $M_{\mathcal{E}}^{-} := (\mathcal{B}^{-}, \mathcal{B}_{\rightarrow}^{-})$, we refer exclusively to the pure structure (not the parametric part) of the DBN that captures the environment.

The approach follows an annotation-based instantiation process in which the objects of an architecture model are annotated with template variable-related attributes. More specifically, we implemented a *Stereotype* that allows the annotation of objects of an architecture model in an ADL-agnostic way. The stereotype is realised by using EMF Profiles [110] that implement the concept of UML Profiles [62] in the context of EMF. In a nutshell, an EMF Profile is applied to an EMF model such that stereotypes (defined within the profile) can be applied to objects of the EMF model. Stereotypes extend metaclasses of metamodels with additional attributes. That is, when a stereotype is applied to a model object at the instance level, it enriches the object with more attributes.

In our case, we use stereotypes to annotate model objects of the architectural model. The annotation of an object with the stereotype indicates that one or more templates are instantiated for the object. We denote the stereotype of such an annotation as `InstantiationTag`. The `InstantiationTag` stereotype is shown in Figure 5.13.

The `InstantiationTag` is referencing a `TemplateVariable` and a `TemplateVariableGroup` which specifies template variables that are to be instantiated. Although the stereotype offers the possibility to reference both, it is expected that either a `TemplateVariable` or a `TemplateVariableGroup` are specified; only one of them can be instantiated but not both at the same time. An `InstantiationTag` references exactly one `Argument`. Recall that an

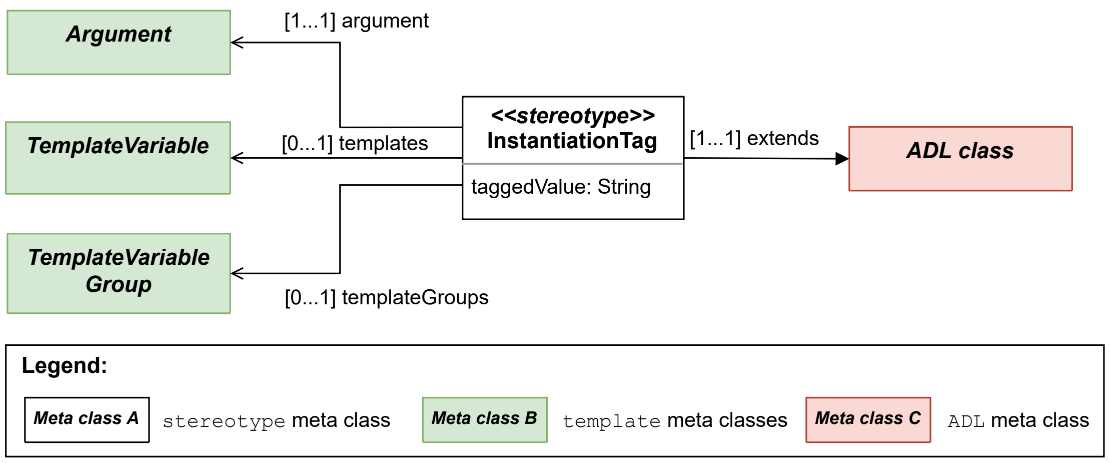


Figure 5.13.: `InstantiationTag` stereotype for annotating architecture models.

`Argument`-object corresponds to a particular object class Q. Thus, an `InstantiationTag` associated with a particular `Argument` instance (i.e. Q) must only be applied to model objects of the architecture model that are considered to be objects of object class Q. Each stereotype can define a set of *Tagged Values* which form the attributes of a stereotype or rather the attributes that enrich the extended metaclass. Regarding `InstantiationTag` there is a single `taggedValue` attribute of type string. Because the same template (or group of templates) can be instantiated multiple times for model objects of the same object class, the attribute `taggedValue` uniquely connects the template with the objects for which it is instantiated. The `taggedValue` attribute is to be understood as an ID for each applied `InstantiationTag`. Finally, a stereotype extends one metaclass of another metamodel as depicted in Figure 5.13. In terms of `InstantiationTag`, a metaclass of an ADL has to be extended to which the stereotype can be applied. Ideally, the ADL-specific metaclass is a superclass that is inherited by most other metaclasses such that `InstantiationTag` can be applied to numerous objects at the instance level.

Example 4. Recall the DeltaIoT example system from section 1.5.2. Suppose an architecture model representing the DeltaIoT system for which we want to instantiate the template variables MA, WI and SNR from section 5.2.3.2 by annotating the corresponding objects (that are instances of object class *Mote* or *WirelessLink*) with an `InstantiationTag`. Each mote object of the architectural model is annotated by an `InstantiationTag` with a unique `taggedValue`, a reference to the template variable MA (not a template group) and an argument referring to Q_{MA}. Similarly, each wireless link object shall be annotated by an `InstantiationTag` with a unique `taggedValue`, a reference to a `TemplateVariable-Group`-object (i.e. WI and SNR) and a corresponding argument of the wireless link. ■

When all model objects (for which templates are to be instantiated) are annotated with corresponding `InstantiationTags`, the structural environment model can be generated. The algorithm for generating the structural environment model is depicted on algorithm 5.1. Note that the algorithm is described from an abstract and formal perspective and reflects only the core concepts; the details of the algorithm should be looked up in the code. The algorithm inputs an architecture model M_C and a set of template variables $M_{\aleph}$ that represent a `TemplateVariableDefinition`-object. The templates defined in $M_{\aleph}$ must be instantiated in the architecture model M_C by following the annotation-based approach (i.e. by applying `InstantiationTags` as described before). Based on M_C and $M_{\aleph}$, the structural environment model $M_{\mathcal{E}}^{-}$ is generated and returned by the algorithm. In principle, the algorithm is divided into five parts which we discuss in the following.

Algorithm 5.1: The structural environment model generation algorithm

```
Input: Architecture model $M_C$, template variable definitions $M_{\aleph}$
Output: Structural environment model $M_{\mathcal{E}}^{-}$
/* Filter all annotated elements from $M_C$ */
$O_{\aleph}^{\kappa} \leftarrow filterAnnotatedElements(M_C)$
/* Create instantiation contexts and equivalence classes */
$IC \leftarrow \emptyset$
foreach $o \in O_{\aleph}^{\kappa}$ do
    $id, Q \leftarrow extractIdAndObjectClass(o)$
    $\mathcal{V}_{IC} \leftarrow \emptyset$
    $IC \leftarrow IC \cup ic := (id, Q, \mathcal{V}_{IC}, o)$
end
$P_{IC} \leftarrow \{[ic]_{\sim} \subseteq IC \mid [ic]_{\sim} := \{ic' := (id', Q', \mathcal{V}'_{IC}, o) \in IC \mid id = id'\}\}$
/* Assign each $o$ to $\mathcal{V}$ for which $\mathcal{V}$ is instantiated */
foreach $[ic]_{\sim} \in P_{IC}$ do
    $\mathcal{V}_{[ic]_{\sim}} \leftarrow extractCommonTemplateStructure([ic]_{\sim})$
    foreach $\mathcal{V} \in \mathcal{V}_{[ic]_{\sim}}$ do
        foreach $ic := (id, Q, \mathcal{V}_{IC}, o) \in [ic]_{\sim}$ do
            if $\exists U_i \in \alpha(\mathcal{V}) : Q[U_i] = Q$ then
                $\mathcal{V}_{IC} \leftarrow \mathcal{V}_{IC} \cup \mathcal{V}$
            end
        end
    end
end
/* Instantiate template variables to obtain $\mathcal{X}_{\kappa}[\aleph]$ */
$\mathcal{X}_{\kappa}[\aleph] \leftarrow \emptyset$
foreach $[ic]_{\sim} \in P_{IC}$ do
    foreach $\mathcal{V} \in \mathcal{V}_{[ic]_{\sim}}$ do
        $O_{\mathcal{V}} \leftarrow \{o \in ic \mid ic := (id, Q, \mathcal{V}_{IC}, o) \in [ic]_{\sim} : \mathcal{V} \in \mathcal{V}_{IC}\}$
        $\gamma \leftarrow asTuple(O_{\mathcal{V}}, \alpha(\mathcal{V}))$ // i.e. $\gamma := (U_1 \mapsto o_1, ..., U_n \mapsto o_n)$
        $\mathcal{X}_{\kappa}[\aleph] \leftarrow \mathcal{X}_{\kappa}[\aleph] \cup \mathcal{V}(\gamma)$
    end
end
return $generateDBN(\mathcal{X}_{\kappa}[\aleph], M_{\aleph})$
```

In terms of efficiency, only the third loop (the assignment of all o to the respective template $\mathcal{V}$) is worth mentioning. Ultimately, in the worst case, the loop iterates over $|M_{\aleph}| \cdot |O^{\kappa}_{\aleph}|$ elements. However, we argue that efficiency is not a serious problem because the generation algorithm inputs human-made models. In other words, it would take a software engineer a considerable amount of time to construct a model, leading to serious efficiency problems.

In the first part of the algorithm all elements (denoted by the set $O^{\kappa}_{\aleph}$) of the architecture model M_C that are annotated by an `InstantiationTag` are filtered, i.e. $O^{\kappa}_{\aleph} \subseteq M_C$. Theoretically, this is achieved by checking whether for an architectural object $o \in M_C$ there exists a template variable $\mathcal{V} \in M_{\aleph}$ such that object o is a possible instantiation object of $\mathcal{V}$:

$$\exists U_i \in \alpha(\mathcal{V}) : o \in O^{\kappa}[Q[U_i]] \tag{5.1}$$

Practically, all elements $o \in M_C$ are checked whether they are annotated with an `InstantiationTag`. All instantiated template variables associated with the corresponding `InstantiationTag` satisfy the requirement of formula (5.1).

The second part of algorithm 5.1 (starting at line 2) takes the filtered objects $O^{\kappa}_{\aleph}$ and creates so-called *Instantiation Contexts*. Formally, an instantiation context $ic := (id, Q, \mathcal{V}_{IC}, o)$ is a tuple where id and object class Q refer to the `taggedValue` and `Argument` attributes of `InstantiationTag` applied to object o. The id and object class Q are directly extracted from the applied `InstantiationTag`. The $\mathcal{V}_{IC}$ of an instantiation context ic refers to a set of templates and is initially empty. For each object o (with respective `InstantiationTag`) an instantiation context is created so that from set $O^{\kappa}_{\aleph}$ a set of instantiation contexts IC is generated. The instantiation contexts IC are partitioned into equivalence classes $IC/\sim$ where the equivalence relation $\sim$ is defined by the id identity of instantiation contexts, i.e. $ic := (id, Q, \mathcal{V}_{IC}, o) \sim ic' := (id', Q', \mathcal{V}'_{IC}, o') \Leftrightarrow id = id'$. The partitioned sets of IC are denoted as P_{IC}, i.e. $P_{IC} = IC/\sim$.

After the creation of instantiation contexts and their partitioning into equivalence classes, the next part of the algorithm (starting at line 9) complements each instantiation context ic of a partition $[ic]_{\sim} \in P_{IC}$ with the concrete template variables. Recall that an $ic := (id, Q, \mathcal{V}_{IC}, o)$ contains a set of template variables $\mathcal{V}_{IC}$ which have been initially declared to be empty, i.e. $\mathcal{V}_{IC} = \emptyset$. It should also be noted that all `InstantiationTags` applied to the objects $o \in ic$ of an instantiation context $ic \in [ic]_{\sim}$ within a partition $[ic]_{\sim} \in P_{IC}$ have the same template structure, i.e. the set of template variables referenced by `InstantiationTag` and applied to o. Thus, for each $[ic]_{\sim} \in P_{IC}$ the common template structure (denoted as $\mathcal{V}_{[ic]_{\sim}}$) is extracted that refers precisely to the template variables contained in all `InstantiationTags` applied to the object o of each instantiation context ic of partition $[ic]_{\sim} \in P_{IC}$. Each template variable $\mathcal{V} \in \mathcal{V}_{[ic]_{\sim}}$ is now added to the set $\mathcal{V}_{IC}$ of each instantiation context if there exists a logical variable $U_i \in \alpha(\mathcal{V})$ such that the object class Q of an instantiation context matches object class $Q[U_i]$ of logical variable U_i. Roughly speaking, the template variable $\mathcal{V}$ is added to all instantiation contexts for whose object the template variable is instantiable. The objects for which a template variable is instantiated are thus resolved in a reverse manner.

Based on the complemented instantiation contexts, the template variables are instantiated and become ground random variables in the penultimate part of the algorithm (starting at line 19). This step iterates over all partitions $[ic]_\sim \in P_{IC}$. Moreover, for each template $\mathcal{V} \in \mathcal{V}_{[ic]_\sim}$ in the respective template structure $\mathcal{V}_{[ic]_\sim}$ a ground random variable is created. Before instantiating template variable $\mathcal{V}$, however, the objects for which $\mathcal{V}$ is instantiated have to be resolved. From the previous part of the algorithm, it is known that all instantiation contexts are complemented by the template variables that are instantiated for the objects contained in the instantiation contexts. Therefore, for a template $\mathcal{V}$ all objects $O_{\mathcal{V}}$ of each instantiation context $ic \in [ic]_\sim$ are filtered where $\mathcal{V}$ is included, i.e. $O_{\mathcal{V}} := \{o \in ic \mid ic := (id, Q, \mathcal{V}_{IC}, o) \in [ic]_\sim : \mathcal{V} \in \mathcal{V}_{IC}\}$. The set $O_{\mathcal{V}}$ has to be transformed to a tuple γ w.r.t. the argument signature of $\mathcal{V}$, i.e. $\gamma := (U_1 \mapsto o_1, ..., U_n \mapsto o_n)$ w.r.t. $\alpha(\mathcal{V})$. Finally, the template is instantiated $\mathcal{V}(\gamma)$ and added to the set of ground random variables $\mathcal{X}_\kappa[\aleph]$.

In the last part of the algorithm (see line 27), the structural environment model $(\mathcal{B}^-, \mathcal{B}^-_\rightarrow)$ is generated (w.r.t. $\mathcal{X}_\kappa[\aleph]$ and the template variable definitions $M_\aleph$) and returned. The function $generateDBN(\mathcal{X}_\kappa[\aleph], M_\aleph)$ abstracts away the details of the remaining generation algorithm. We deliberately do not go into the details as they involve the same steps we discussed in section 5.2.

The result of the algorithm is the structural environment model, which still has to be complemented with the parametric information of the probability distributions. The parametric part of the environment model cannot be derived from the architecture model and the template variable definitions. Instead, only the structural representation is generated automatically and must be complemented manually with probability distributions in a final step.

5.4. Implementation

In the previous sections, we presented the *EnvDyn* and *ProbDist* metamodels as well as an approach for a semi-automated generation of the structural environment model. However, the pure model instances of the *EnvDyn* and *ProbDist* metamodel do not provide any functionality such as sampling or evaluating the probability of certain events. Therefore, we implemented additional components that provide such basic functionality. In this section, we give a brief overview of the implemented components. The dependency graph of the components is depicted on Figure 5.14. We have made the code available at [155].

The *EnvDyn* and *ProbDist* metamodels are implemented based on the Eclipse Modeling Framework (EMF). Thus, the metamodel and the generated metamodel code are located in the components `EnvDyn.Model` and `ProbDist.Model`. Note that EMF-based metamodels allow the generation of the respective metamodel code to represent the metaclasses and make them accessible at the code level.

The `ProbDist.API` component provides the basic functionality of probability distributions (i.e. evaluating probabilities and sampling) and depends on the `ProbDist.Model` component. For instance, it takes an *ProbDist* model instance and calculates the probabilities of certain events w.r.t. the specified probability distribution of the model.

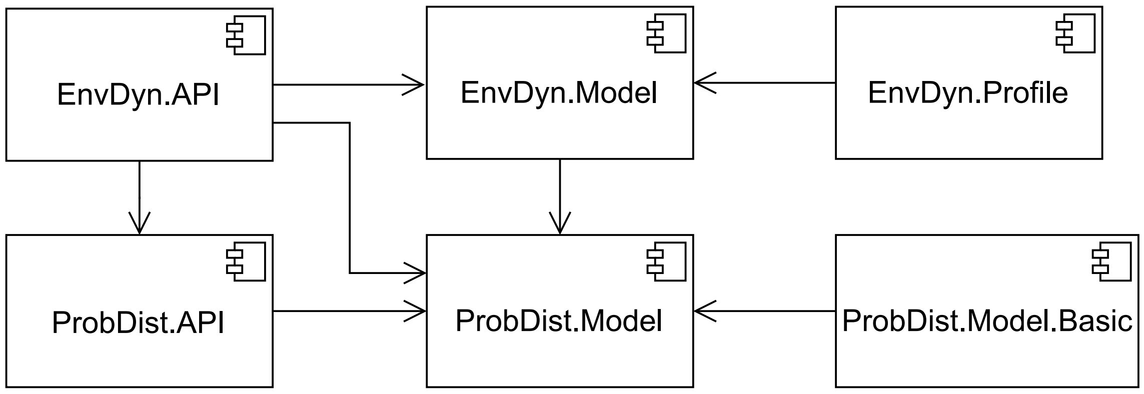


Figure 5.14.: Dependency graph of the implemented *EnvDyn* components.

Recall from section 5.2.6.1 that probability distributions modelled with *ProbDist* must contain type-level descriptions of the corresponding distribution, e.g. distribution type (such as normal distribution) and parameter types (such as mean and variance). Because such type-level distribution descriptions are reused in any probability distribution, the `ProbDist.Model.Basic` provides a model of the `distributiontype` package of *ProbDist* that already models some basic distribution types and parameters. Thus, the component can be reused in multiple contexts and reduces the modelling effort.

The `EnvDyn.API` component provides the basic functionality of Bayesian models, i.e. it implements the core logic of BNs and DBNs. Just as the `ProbDist.API`, it takes an *EnvDyn* model instance and returns either a BN or DBN implementation. Because BNs and DBNs are complex structured probability distributions, they provide the core functionalities of evaluating the probability of certain events and sampling. In addition, the code for the semi-automated generation of the structural environment model is included in the component.

Finally, the `EnvDyn.Profile` component contains the stereotype implementation (i.e. the `InstantiationTag` stereotype introduced in section 5.3.2). Thus, the component can be loaded and applied to any EMF-based architecture model.

5.5. Assumptions and Limitations

In this chapter, we discuss the made assumptions and limitations of the environmental dynamics metamodel.

Assumptions To model the environmental dynamics of a self-adaptive system in a compact form, we considered DBNs. However, the compact representation of DBNs results from two assumptions, namely the Markov assumption and the stationary assumption (see sections 2.4.1 and 2.6.2), which we need to discuss. As discussed in chapter 4 several widely accepted approaches employ the formal framework of DTMCs or MDPs to predict the future behaviour of self-adaptive systems or to learn adaptation strategies based on reinforcement learning. Because DBNs are specialisations of DTMCs, the Markov and stationary assumptions also apply to DTMCs which have been successfully used in connection with self-adaptive systems. Therefore, we argue that the two assumptions are not too restrictive. Moreover, the Markov assumption can be sufficiently approximated in terms of DBNs by adding more random variables to the state description of a self-adaptive system that makes the Markov assumption more reasonable [105, P.202].

Recall that a DBN is a network of random variables where each random variable is associated with a probability distribution. If the random variables are continuous, the resulting state space of the DBN is arguably difficult to analyse. Therefore, we assumed the random variables to be discrete and multinomial distributed to tackle the state space explosion problem. One may argue that the multinomial distribution assumption or rather the discretisation of states is associated with information loss such that the resulting DBN model might not accurately capture the environmental dynamics. However, as discussed in chapter 4, in the self-adaptive system community it is widely accepted to consider the environment as DTMC consisting of discrete states. Moreover, as the *EnvDyn* metamodel satisfies the requirement *Discretisation Level* (see section 5.2.7), one can adjust the resolution of the state description to be more fine-grained (at the cost of state space explosion). Finally, in the context of this work, the *EnvDyn* metamodel is used for design-time analysis. At design-time, abstraction is exploited to eliminate details that are not required for analysis. This simplifies the analysis process and allows predictions of particular system attributes, e.g. performance or reliability. Even if the result of the prediction does not perfectly reflect reality, it provides information on whether the initially designed system meets the quality requirements. In summary, using abstraction is a widely used and accepted method to predict system attributes using models that sufficiently reflect the runtime behaviour.

Limitations Although the *EnvDyn* metamodel addresses the state space explosion problem by discretising the states, the state space still grows exponentially in the number of environmental variables (see section 4.4.1). As a result, there might still be domains in which the modelled environment indicates a large state space. In such domains, *Monte-Carlo-Simulation* [148] or *Importance Sampling* [105, P.494] can be applied to explore the state space for the most probable states. This does not examine the entire state space, but at least the most probable environmental states, which should be representative enough for design-time analysis.

Finally, one last limitation of the *EnvDyn* refers to the modelling process that can be still cumbersome and error-prone for large and complex structured environment models. For instance, consider an environmental variable e_1 which depends on, say, three other environmental variables e_2, e_3, e_4. In this case, the corresponding random variable X_{e_1} of e_1 forms a CPD with three conditional random variables. In terms of the *EnvDyn* (or rather *ProbDist*) metamodel, the CPD is represented by a table comprising $|Val(e_2) \times Val(e_3) \times Val(e_4)|$ rows. Even for small numbers of $Val(e_i)$ with $i \in \{2, 3, 4\}$, the modelling effort is not negligible. However, there are still methods (e.g. [105, P.157]) which tackle this significant disadvantage by considering different CPD representations. Currently, such methods are not considered in the *EnvDyn* metamodel but are the subject of future work.

5.6. Summary

In this chapter, we have introduced the *EnvDyn* metamodel that enables the modelling of the stochastic environment in which self-adaptive systems operate.

Therefore, we initially discussed in section 5.1 the requirements that the metamodel must satisfy, namely domain independence, ADL-agnostic, stochastic dynamics, stochastic correlations, compactness and discretisation level.

Before presenting the metamodel in section 5.2, we discussed why DBNs are perfectly suited to represent the environmental dynamics of self-adaptive systems. We then presented the *EnvDyn* metamodel that is divided into three packages, namely the `template`, `static` and `dynamic` package. We have indicated how each of the packages conforms to the formal semantics of probabilistic template-based models (see section 2.6.3), which plays an important role in enabling domain-independent application. The section was concluded with a discussion on the fulfilment of the aforementioned requirements for the metamodel.

In section 5.3, we outlined how an initially modelled *EnvDyn* instance is instantiated into an architecture model which completes the environment model. Hereby, we described a manual instantiation (e.g. performed by a software engineer) and an annotation-based instantiation process. The latter allows the annotation of specific objects in the architecture model for which an *EnvDyn* model should be instantiated. Based on the annotated architectural model, we have presented a semi-automated approach to generate the structural environment model, which only needs to be complemented by a probabilistic description of the instantiated random variables of the environment model.

Finally, we briefly presented the implementation of the *EnvDyn* metamodel in section 5.4 and discussed the made assumptions in section 5.5.

6. Evaluating Self-Adaptive Systems by Simulating Experience: The SimExp Method

In this chapter, we present a model-based method for evaluating self-adaptive systems (or rather adaptation strategies that self-adaptive systems pursue). The contribution, presented in this chapter, is based on the publication [158].

So far, in chapter 4, we have discussed the underlying mathematical framework, i.e. MDPs. Subsequently, in chapter 5 we have discussed how a particular concept of this mathematical framework, namely environmental dynamics, is captured in a model-based way, i.e. by reusing concepts from the field of probabilistic graphical models. In section 4.3.1 we already discussed how the concepts of self-adaptive systems are mapped to concepts that constitute an MDP. Now, we discuss how the concepts of self-adaptive systems are mapped into the domain of *Model-based Quality Analysis* (MBQA), i.e. a branch of MDSD (see section 2.2) that makes use of models to analyse system attributes (e.g. performance or reliability). Based on MBQA, we simulate the dynamics and evaluate adaptation strategies of self-adaptive systems based on the formal framework of MDPs. More specifically, we use *Dynamic Programming* (DP) (see section 2.5) which provides a collection of methods for evaluating, improving or (in general) optimising policies in MDPs (recall that an adaptation strategy reflects a policy π). Because our primary goal is to evaluate adaptation strategies, we use a method of DP referred to as *Policy Evaluation* (see section 2.5.1) to determine the quality of a strategy.

The result of this chapter is a method that we denote as *SimExp* method. *SimExp* stands for simulated experience and originates from the field of reinforcement learning where simulated experience is produced to learn policies ([180, P.131]). Note that we deliberately use the term method instead of approach because *SimExp* rather resembles a framework than a ready-to-use tool. To be more specific, one main question that this work is concerned with is generalisability, i.e. the evaluation of adaptation strategies domain-independently and without a specific focus (e.g. safeguarding AI black-box components). Technically, the generic nature of MDPs allows such an analysis at design-time, provided that we can adequately model the relevant elements of a self-adaptive system. We will see, however, that the instantiation of the *SimExp* method always requires domain-specific extensions. Therefore, we present in this chapter a framework that implements the main building blocks of the *SimExp* method that are extended or complemented by domain-specific concepts for evaluating adaptation strategies.

This chapter addresses research question **RQ1**, which is:

Research Question 1: How to evaluate adaptation strategies of self-adaptive systems at design-time regarding the ability to meet quality objectives?

As presented in section 1.3, research question **RQ1** breaks down into sub-research questions that, when answered individually, allow the main research question **RQ1** to be answered. The sub-research questions **RQ1.1** and **RQ1.2** have already been dealt with in previous chapters. This chapter relates to the sub-research question **RQ1.3**, which is:

Research Question 1.3: What is an appropriate analytical model to enable design-time analyses of self-adaptive systems?

As an analytical model for analysing self-adaptive systems, we consider MDPs and employ methods of DP to evaluate policies π, i.e. adaptation strategies.

The chapter is structured as follows: In section 6.1, we generally describe the process of using model-based techniques to evaluate adaptation strategies. In section 6.2, we formally describe how we make use of DP and Monte Carlo methods to evaluate adaptation strategies. Afterwards, in section 6.3 we present the *SimExp* method. In section 6.4 we briefly present implementation details of the *SimExp* framework. Finally, in section 6.5 we discuss limitations and assumptions and summarise the chapter in section 6.6.

6.1. Evaluating Adaptation Strategies at Design-time

In this section, we give a brief overview of the formal components of *SimExp* and their relation to each other. Furthermore, we have only associated an adaptation strategy with the concept of a policy in MDPs, but have not provided details on what we consider an adaptation strategy. We make up for this in this section and start by defining an adaptation strategy for MAPE-K-based self-adaptive systems.

In literature, the term adaptation strategy is often associated with the plan-phase of MAPE-K-based self-adaptive systems (e.g. [38, 48]). In some approaches, adaptation mechanisms are hierarchically structured into *Strategies*, *Tactics* and *Actions* (S/T/A). Moreover, (S/T/A) strategies consist of several tactics where each tactic composes a set of actions (e.g. [47, 88]). However, we treat the term adaptation strategy somewhat more broadly. In section 4.3.1 we introduced an adaptation strategy abstractly as a deterministic function π that implements the decision procedure for choosing a particular adaptation δ in a given state without making any assumptions about the internals of the strategy. However, we view any activity as part of an adaptation strategy that influences the decision procedure in selecting an adaptation. In terms of MAPE-K-based implementations, this includes the activities along the MAPE phases. We argue that design decisions in all MAPE phases (and not only in the plan-phase) might potentially determine whether an adaptation is selected or not. Consequently, they must be considered as part of the strategy that adheres to the semantics of policies π in MDPs. The following example illustrates our argument:

Example 5. Let us suppose two MAPE-K-based adaptation strategies π, π'. Let us assume also that π' is identically implemented to π, i.e. $\forall S \in \mathcal{S} : \pi(S) = \pi'(S)$. For simplicity, we consider π and π' as adaptation strategies that monitor a single property, denoted as φ, in the monitor-phase and determine in the analyse-phase whether an adaptation is planned by evaluating the condition $\varphi \geq \varepsilon$ where ε is some threshold. If we change π' slightly by adjusting the threshold ε to ε', we might run into a situation where π and π' do not behave equally, i.e. $\exists S \in \mathcal{S} : \pi(S) \neq \pi'(S)$. This means that the mere change of threshold ε' in the monitor-phase of π' alters its behaviour at the same time. ■

Example 5 illustrates how activities of MAPE phases may influence the decision of executing an adaptation. More generally, decisions made by one phase can influence the actions of other phases. For example, the specific decision on which monitors to use to perceive the current state in the monitor-phase directly affects the analyse-phase, as the monitored properties are evaluated to determine whether an adaptation is planned or not. Furthermore, the plan-phase can use the monitored properties to plan an adaptation. Taking into account the previous discussion, we define an adaptation strategy as follows:

Definition 29 (Adaptation Strategy). *An adaptation strategy of a MAPE-K-based self-adaptive system is a function $\pi : \mathcal{S} \rightarrow \Delta$ that selects an adaptation $\delta \in \Delta$ in a given state $S \in \mathcal{S}$. Moreover, it encompasses all activities along the MAPE phases that affect the decision of selecting an adaptation.*

After we defined an adaptation strategy, we now embed the evaluation of such strategies in the context of MBQA by following the formal semantics of MDPs and DP. Therefore, recall definition 28 that introduces the stochastic dynamics of a self-adaptive system, which was:

Definition 28 (Stochastic Dynamics of Self-Adaptive Systems). *The dynamics of a self-adaptive system is a stochastic process $(X_{\mathcal{S}_t})_{t \in \mathbb{N}}$ for which the Markov assumption holds. More precisely, the stochastic process is captured by a Markov decision process $\lambda_{SAS} := (\mathcal{S}, \Delta, t_{\mathcal{S}}, r_{\mathcal{S}})$ where*

- *$\mathcal{S}$ corresponds to the set of self-adaptive system states.*
- *Δ corresponds to the set of adaptations.*
- *$t_{\mathcal{S}} : \mathcal{S} \times \Delta \times \mathcal{S} \rightarrow [0, 1]$ corresponds to the transition function where $t_{\mathcal{S}} = P(X_{\mathcal{E}_{t+1}} \mid X_{C_t}, X_{\mathcal{E}_t}) \cdot \mathbb{1}_{\phi(C_t, \Delta_t) = C_{t+1}}$ (according to theorem 4.3.1).*
- *$r_{\mathcal{S}} : \mathcal{S} \times \Delta \times \mathcal{S} \rightarrow \mathbb{R}$ corresponds to the reward function encoding quality objectives.*

Also, recall the engineering problem of section 4.4.2, which was, in summary, to develop an adaptation strategy that maintains quality objectives over time encoded by rewards. The idea of our approach is to evaluate adaptation strategies using models to address the engineering problem at design-time. This requires, however, that the concepts of definition 28 must be represented in a model-based way, which are: The environmental dynamics, the architecture of the managed system, adaptations and the adaptation process ϕ itself, the adaptation strategy and the reward function.

In the last chapter, we already introduced the *EnvDyn* metamodel which captures the environmental dynamics (recall that model instances of *EnvDyn* are denoted as environment model $M_{\mathcal{E}}$). Recall also that the tuple $\mathcal{M} := (M_{\mathcal{C}}, M_{\mathcal{E}})$ indicates that the environment model $M_{\mathcal{E}}$ is generated by the architecture model $M_{\mathcal{C}}$. The architecture model $M_{\mathcal{C}}$ is modelled by an ADL (architecture description language) representing the managed system. Because the *EnvDyn* metamodel is ADL-agnostic, no further assumptions need to be made about the architecture model (except that they must be Ecore-based). Thus, we assume that the architecture model is given in terms of an Ecore-based ADL (e.g. PCM). Adaptations and the adaptation process are represented by using model transformations (see section 2.2.2). Each adaptation is represented by an individual model transformation. The adaptation process is simulated by applying a transformation to the architectural model $M_{\mathcal{C}}$, as one obtains a new and adapted model $M'_{\mathcal{C}}$.

These three ingredients, namely the architecture model, environment model and model transformations, enable the implementation of a framework that simulates the environment (i.e. by sampling trajectories from the environment model) and executing adaptations (i.e. applying model transformations) to adapt the managed system (i.e. the architecture model), if required. Such a framework only needs to be complemented by an adaptation strategy and a reward function to evaluate the decisions made by the strategy. This is realised by providing interfaces that need to be implemented to plug in an adaptation strategy and reward function.

The engineering problem states that adaptation strategies must maintain quality objectives over time. These quality objectives are encoded by rewards and serve as a basis for evaluating how well an adaptation strategy satisfies these objectives. However, quality attributes (such as performance or reliability) are traditionally observed at runtime. Therefore, we employ simulation and analysis of models (i.e. MBQA) to predict quality attributes at design-time. For example, the Palladio framework provides a repertoire of analysis and simulation tools that can be applied to PCM instances to predict quality attributes. The PCM-based tools allow predicting only a particular set of quality attributes; however, there are potentially more prediction tools one can take into consideration for design-time analysis (e.g. Prism [109]). Consequently, the framework must provide extension points to provide an entire repertoire of tools for implementing the reward function accordingly. The outlined framework reflects the core idea of *SimExp* to evaluate adaptation strategies. An overview of the method is depicted on Figure 6.1.

As mentioned in the beginning, simulating experience originates from the field of reinforcement learning. Hereby, simulated experience is considered as a tuple consisting of four elements, namely a state and selected action at time t and the next state and observed

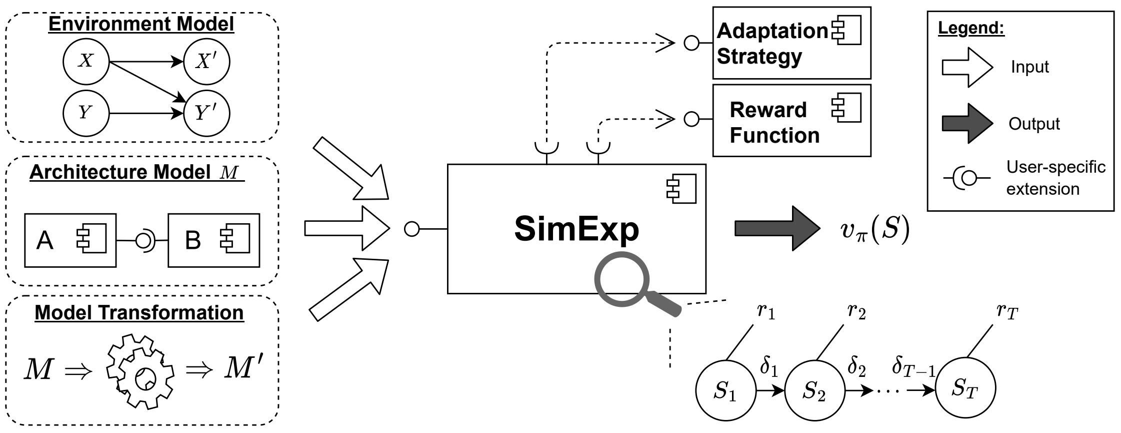


Figure 6.1.: Overview of the *SimExp* framework.

reward at time $t + 1$. In the context of MDPs, such tuples are produced by simulating the environment. Recall that the dynamics of the environment are captured by the probability distribution $p(s', r|s, a)$ that describes the probability of observing state s' and reward r at time $t + 1$ given the current state s and selected action a at time t (see section 2.4.2). In principle, two kinds of models are considered to represent $p(s', r|s, a)$, namely distribution models (i.e. probability distributions that hold all possible outcomes and their probabilities) and sample models (i.e. models that produce individual samples from $p(s', r|s, a)$). Both models are used to generate sequences or trajectories of states by repeatedly sampling transitions $p(s', r|s, a)$ and applying π in response to the new sampled state s'. The simulated experience produced by this procedure enables the evaluation and optimisation of policies π.

In this work, we reuse the idea of simulated experience to evaluate adaptation strategies π. Moreover, $p(s', r|s, a)$ represents the stochastic dynamics of self-adaptive systems (see definition 28) and is captured by a sample model to produce simulated experience. We employ Monte Carlo methods to estimate the quality of an adaptation strategy π w.r.t. the observed reward. Rewards are determined by applying simulation and analysis techniques from MBQA. As our primary concern is to evaluate adaptation strategies in terms of maintaining quality objectives, MBQA is used to predict quality attributes for a given state. That is, we integrate MBQA techniques in the reward function implementation and use the quality attribute predictions to evaluate decisions made by an adaptation strategy.

6.2. A Formal Framework for Evaluating Adaptation Strategies

In this section, we present the formal framework that underlies the *SimExp* method. In the previous chapters, we discussed that we consider the stochastic dynamics of self-adaptive systems as MDPs and explained how the concepts of self-adaptive systems are mapped to the corresponding concepts of MDPs. The advantage of this mapping is that methods applicable to MDPs are similarly applicable to self-adaptive systems. In fact, DP (see section 2.5) provides methods to evaluate or improve policies of MDPs which we reuse in the context of this work. More accurately, we apply the policy evaluation method of DP (see section 2.5.1) to evaluate adaptation strategies of self-adaptive systems. In the following, we discuss how to apply policy evaluation in the context of self-adaptive systems more formally.

6.2.1. Using Dynamic Programming to Evaluate Adaptation Strategies

In section 2.5.1, we presented policy evaluation which is an approach of DP to evaluate a policy π. More specifically, the value function v_π of a policy π is computed. In the following, we discuss how we apply policy evaluation to evaluate adaptation strategies.

6.2.1.1. Computing the Value Function of Adaptation Strategies

Algorithm 2.1 of section 2.5.1 illustrates the algorithm of policy evaluation. The algorithm iteratively updates the value of a state w.r.t. the Bellman equation. For the sake of clarity, we show the Bellman equation once again:

$$\forall s \in S : v_\pi(s) = \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a)\Big(r + \gamma v_\pi(s')\Big) \tag{2.9}$$

Also, recall that $p(s', r|s, a)$ is a shorthand notation of the conditional probability of observing state s' and reward r at time $t + 1$ given state s and action a at time t, i.e. $p(s', r|s, a) := Pr(X_{S_{t+1}} = s', X_{R_{t+1}} = r \mid X_{S_t} = s, X_{A_t} = a)$.

The Bellman equation forms the core of policy evaluation and must be transferred into the domain of self-adaptive systems before using it to evaluate adaptation strategies. First, the set of states $s \in S$ and set of actions $a \in A$ must be replaced by the self-adaptive system state space $S \in \mathcal{S}$ and adaptation space $\delta \in \Delta$ (according to section 4.3.1). Moreover, we defined an adaptation strategy π as deterministic function; that is, the first sum of the equation (2.9) (i.e. $\sum_a \pi(a \mid s)$) is therefore omitted and the selected adaptation of π is directly transferred to the conditional part of $p(s', r|s, a)$:

$$\forall S \in \mathcal{S} : v_\pi(S) = \sum_{S',r} p(S', r \mid S, \pi(S))\Big(r + \gamma v_\pi(S')\Big) \tag{6.1}$$

In MDPs, the dynamics of the environment are captured by $p(s', r|s, a)$. Applied to self-adaptive systems, the function reflects the dynamics of self-adaptive systems and factorises as follows:

$$\begin{aligned} p(S', r|S, \delta) :=& Pr(X_{\mathcal{S}_{t+1}} = S', X_{R_{t+1}} = r \mid X_{\mathcal{S}_t} = S, X_{\Delta_t} = \delta) \\ =& Pr(X_{R_{t+1}} = r \mid X_{\mathcal{S}_{t+1}} = S', X_{\mathcal{S}_t} = S, X_{\Delta_t} = \delta) \\ & \cdot Pr(X_{\mathcal{S}_{t+1}} = S' \mid X_{\mathcal{S}_t} = S, X_{\Delta_t} = \delta) \\ =& \mathbb{1}_{r_\mathcal{S}(S,\delta,S')=r} \cdot t_\mathcal{S}(S, \delta, S') \end{aligned} \tag{6.2}$$

Formula (6.2) shows that $p(S', r|S, \delta)$ factorises into two products. The first product (i.e. $Pr(X_{R_{t+1}} = r \mid X_{\mathcal{S}_{t+1}} = S', X_{\mathcal{S}_t} = S, X_{\Delta_t} = \delta))$ evaluates the probability of observing a particular reward r given states S, S' and adaptation δ. However, definition 28 states that the reward function is deterministic; that is, one can deterministically compute the reward value with reward function $r_{\mathcal{S}}(S, \delta, S')$. This deterministic property allows us to rewrite the probability of the first product by using the indicator function. The second product (i.e. $Pr(X_{\mathcal{S}_{t+1}} = S' \mid X_{\mathcal{S}_t} = S, X_{\Delta_t} = \delta))$ refers to the transition function of definition 28.

If we substitute equation (6.2) in equation (6.1), we obtain a new equation which looks as follows:

$$\begin{aligned}\forall S \in \mathcal{S} : v_\pi(S) &= \sum_{S',r} \mathbb{1}_{r_{\mathcal{S}}(S,\pi(S),S')=r} \cdot t_{\mathcal{S}}(S, \pi(S), S')\Big(r + \gamma v_\pi(S')\Big) \\ &= \sum_{S'} t_{\mathcal{S}}(S, \pi(S), S')\Big(r_{\mathcal{S}}(S, \pi(S), S') + \gamma v_\pi(S')\Big)\end{aligned} \tag{6.3}$$

$\mathbb{1}_{r_{\mathcal{S}}(S,\pi(S),S')=r}$ cancels out all summands with $r_{\mathcal{S}}(S, \pi(S), S') \neq r$. Thus, the equation simplifies to a single sum while the reward function is applied directly to compute the reward for S, S' and δ.

Moreover, recall from section 2.4.2 that parameter γ refers to the discount rate which determines the extent to which future rewards are taken into account. Also, recall that we aim to address the uncertainty *Parameter over time* which is concerned with future effects caused by adaptations. Consequently, for the rest of this thesis, we define the discount rate $\gamma = 1$. Thus, we maximally account for future rewards and address the uncertainty *Parameter over time* as well as possible. Note that since we define $T < \infty$, the existence and uniqueness of value function $v_\pi(S)$ of policy π is guaranteed (see section 2.5.1).

In summary, equation (6.3) corresponds to an adapted version of the Bellman equation by considering formal concepts of self-adaptive systems and their stochastic dynamics. Now one might use DP (or more precisely policy evaluation) to compute the value function of a particular adaptation strategy by recursively evaluating the transition function $t_{\mathcal{S}}$ and reward function $r_{\mathcal{S}}$. However, in section 6.2.2, we will see that applying DP is practically not feasible such that we make use of Monte Carlo methods.

6.2.1.2. Partial Ordering of Adaptation Strategies

In section 2.5, we presented formula (2.8) which induces a partial ordering over the policy space of MDPs. For better readability, we present the formula again, but apply it to the domain of self-adaptive systems, i.e. we take into account the concepts of the self-adaptive system state space and adaptation strategies.

$$\pi \geq \pi' \Leftrightarrow \forall S \in \mathcal{S} : v_\pi(S) \geq v_{\pi'}(S) \tag{6.4}$$

However, formula (6.4) has a significant disadvantage. Suppose two distinct adaptation strategies π, π' where there exist a subset $\mathcal{S}' \subset \mathcal{S}$ such that $\forall S \in \mathcal{S}' : v_\pi(S) \leq v_{\pi'}(S)$ and

$\forall S \in \mathcal{S} \setminus \mathcal{S}' : v_\pi(S) \geq v_{\pi'}(S)$. In other words, π' produces a better expected reward for a given set of states (i.e. $\mathcal{S}'$) while π performs better for the remaining states (i.e. $\mathcal{S} \setminus \mathcal{S}'$). Naturally, the questions arise of which strategy is better. That is, formula (6.4) provides no way to compare the two strategies. For optimisation, the formula is sufficient as it holds for optimal strategies. However, the primary concern of this work is the evaluation; thus, we must expand formula (6.4) such that any pairs of strategies π, π' can be compared.

Recall that the value function for a strategy π is the expected reward when starting in state S and following the respective strategy, i.e. $v_\pi(S) := \mathbb{E}_\pi[X_{G_t} \mid X_{\mathcal{S}_t} = S]$. The fact that value function $v_\pi(S)$ is conditioned on S complicates the comparison of any strategies π, π' and forms the main drawback of formula (6.4). To overcome this drawback, one can get rid of this condition by considering the overall expected reward of a strategy, i.e. $\mathbb{E}_\pi[X_{G_t}]$:

$$\mathbb{E}_\pi[X_{G_t}] := \sum_{S \in \mathcal{S}} Pr(X_{\mathcal{S}_t} = S) \cdot \mathbb{E}_\pi[X_{G_t} \mid X_{\mathcal{S}_t} = S] = \sum_{S \in \mathcal{S}} Pr(X_{\mathcal{S}_t} = S) \cdot v_\pi(S) \tag{6.5}$$

In simple terms, the expected reward of a strategy π is the sum of all value functions $v_\pi(S)$ weighted by the probability of observing S at time t. By considering the overall expected reward, one can assign a value to each strategy and makes it, in fact, possible to compare any strategies π, π'. To clarify this idea, we expand formula (6.4) by considering $\mathbb{E}_\pi[X_{G_t}]$:

$$\begin{aligned}(6.4) \Rightarrow\ & \mathbb{E}_\pi[X_{G_t}] \geq \mathbb{E}_{\pi'}[X_{G_t}] \\ =\ & \sum_{S \in \mathcal{S}} Pr(X_{\mathcal{S}_t} = S) \cdot v_\pi(S) \geq \sum_{S \in \mathcal{S}} Pr(X_{\mathcal{S}_t} = S) \cdot v_{\pi'}(S)\end{aligned} \tag{6.6}$$

Again, let us assume two strategies π and π' where $\pi \geq \pi'$ according to formula (6.4). If this is true, then $\mathbb{E}_\pi[X_{G_t}] \geq \mathbb{E}_{\pi'}[X_{G_t}]$ must also hold. If we expand $\mathbb{E}_\pi[X_{G_t}]$ (and $\mathbb{E}_{\pi'}[X_{G_t}]$, respectively) according to formula (6.5), then we obtain an inequality with two sums adding the weighted value functions of all states S. As a result, to compare two strategies π, π', one merely needs to add the weighted value functions of each strategy. This is shown in formula (6.6).

Note that although (6.6) preserves the optimality criterion (i.e. for an optimal strategy: (6.6) $\Rightarrow$ (6.4)), this is not generally true for any pair of strategies (i.e. for all π, π': (6.6) $\not\Rightarrow$ (6.4)). However, we argue that $\mathbb{E}_\pi[X_{G_t}]$ is more suitable in terms of evaluating and comparing strategies as we associate the value of each state with the probability of observing said state. Thus, state values $v_\pi(S)$ with low probability $Pr(X_{\mathcal{S}_t} = S)$ are less affecting the overall result and vice versa.

As a last remark, note that we could have simply used the total accumulated reward generated by a strategy π. However, when considering the total reward, convergence is not guaranteed (unlike (6.6) where the value function converges). This is especially an issue when comparing strategies because it can not be ruled out that the total rewards for both strategies still change if we sample more trajectories. Nonetheless, we would like to stress that the total reward can still be used as a quality measure.

6.2.2. Using Monte-Carlo-Methods to Generate Simulated Experience

In the last section, we discussed how to make use of DP (or policy evaluation) to compute the value function v_π. Based on the value function, one may assign a real value to a strategy π, inducing a partial ordering over the strategy space that forms the basis for comparison and evaluation. The problem of DP is that it requires complete knowledge of the probability distribution capturing the dynamics of self-adaptive systems, i.e. $p(S', r|S, \delta)$ (recall from equation (6.2)). Generally, this is not the case for self-adaptive systems; this is discussed in more detail in section 6.3.2.

Therefore, we advocate the use of Monte Carlo methods. More specifically, we apply Monte Carlo prediction (recall from section 2.5.2) to estimate the value function v_π. Monte Carlo prediction is an alternative approach to policy evaluation of DP in terms of handling the prediction problem. The great advantage of using Monte Carlo prediction is that it does not require complete knowledge of $p(S', r|S, \delta)$. The key idea is to generate trajectories by probabilistic sampling and estimating v_π based on the samples; such samples are denoted as *Simulated Experience.*

More accurately, Monte Carlo prediction generates trajectories by repeatedly drawing samples from $p(S', r|S, \delta)$ (represented by a sample model) and following strategy π until termination:

$$S_0, \delta_0, r_1, S_1, \delta_1, r_2, S_2, \ldots, r_T, S_T \tag{6.7}$$

For the trajectory depicted on (6.7), for example, each δ_i is determined by applying strategy π; each self-adaptive systems state S_i is sampled from $p(S', r|S, \delta)$. For each generated trajectory, the value of a state S (i.e. $v_\pi(S)$) can be estimated by averaging the expected reward observed after the first visit of S. The more trajectories are sampled, the more estimates are averaged such that $v_\pi(S)$ converges towards its true value.

6.3. Simulating Experience by Model-based Quality Analysis

After we discussed the formal framework of *SimExp* in the last section, we now discuss how we implement the formal concepts by using MBQA. Therefore, we start by discussing how self-adaptive systems and their related concepts are represented by employing models. Afterwards, we outline how these models are simulated to produce experience by applying Monte Carlo prediction.

6.3.1. Modelling Self-Adaptive Systems

In this section, we discuss how the most relevant concepts of self-adaptive systems are represented by models, namely the environmental dynamics, the managed system, the adaptations (and associated adaptation process), the reward function and the adaptation strategy. Each of these is captured by a dedicated concept in the *SimExp* framework which is discussed in the following.

Modelling the Environmental Dynamics The model-based representation of the environmental dynamics has been exhaustively discussed in chapter 5. Therefore, we do not go into detail here. For the sake of completeness, however, we include this section to show that the *EnvDyn* metamodel is an integral part of the *SimExp* framework and corresponds to the model-based representation of the environmental dynamics.

Modelling the Software Architecture of the Managed System One of the main concepts of self-adaptive systems is the managed system, i.e. the system that is supposed to be adapted at runtime. At design-time, we represent the managed system by using an ADL. More specifically, we use the PCM (Palladio Component Model) [149] language as ADL. The reason for choosing PCM is that it is (i) mature and (ii) provides an expressive ADL to describe component-based software architectures. Although we have implemented the *SimExp* framework by first considering PCM instances, we would like to emphasise that *SimExp* is generally not limited to PCM, but is ADL-agnostic (similar to the *EnvDyn* metamodel). We discussed the PCM (or the Palladio approach in general) in section 2.3.

Representation of Adaptations by Model Transformations We abstract an adaptation δ by using model transformations (see section 2.2.2). Recall that our approach depends on EMF and thus on Ecore-based ADLs. Therefore, we must consider MTLs (model transformation languages) that apply to Ecore-based models. Fortunately, EMF provides a set of MTLs that can be used to define model transformations. More specifically, we consider in-place model transformations, i.e. transformations that transform a model M to model M' where the metamodel of M and M' is the same. Thus, the result of a model transformation, when applied to a PCM instance M_C, is a new PCM instance M_C' representing the managed system after executing an adaptation.

Representation of the Adaptation Strategy The adaptation strategy implements the adaptation logic. Thus, it contains a sequence of control structures to plan a concrete adaptation. Representing such adaptation logic with a dedicated DSL is challenging, as one has to find a balance between expressiveness and domain specificity. However, the development of such a language is not in the scope of this work; therefore, we use a general-purpose language (e.g. Java) to represent the adaptation strategy.

More specifically, the *SimExp* framework provides an entry point where software engineers can plug in their strategy implementations. The excerpt of the interface is depicted on listing 6.1. We do not go into the technical details of the listing here, but rather into the interaction of the individual methods. To implement the strategy, a software engineer must implement a set of methods that refer to the monitor-, analyse-, and plan-phase and a method which returns an empty adaptation $\delta_\emptyset$ (according to the semantics of empty adaptations defined in property 1). One might miss the method that represents the execution phase of a MAPE cycle; however, we do not consider the execution of an adaptation as part of the adaptation strategy, which, in fact, corresponds to the application of model transformations and is located in a different part of the code. The main control flow is already implemented in the *select*-method and is complemented by the concrete adaptation logic contained in the remaining methods. Note that the excerpt from the listing 6.1 is a snapshot that may change over time (due to code refactorings).

```
    public abstract class AdaptationStrategy<T> implements Policy<T> {

    ...

    @Override
    public T select(State source, Set<T> options) {
        monitor(source, knowledge);
        if (analyse(source, knowledge)) {
            return plan(source, options, knowledge);
        }
        return emptyReconfiguration();
    }

    protected abstract void monitor(State source, SharedKnowledge knowledge);

    protected abstract boolean analyse(State source, SharedKnowledge knowledge);

    protected abstract T plan(State source, Set<T> options, SharedKnowledge
        knowledge);

    protected abstract T emptyAdaptation();
```

Listing 6.1: Adaptation strategy to be implemented.

Representation of the Reward Function Just like the adaptation strategy, the reward function is represented at code level. In order to plug in a reward function implementation into the *SimExp* framework, one must implement a dedicated interface (see listing 6.2) consisting of a single method.

```
public interface RewardEvaluator {

    public Reward<?> evaluate(StateQuantity quantifiedState);
}
```

Listing 6.2: Reward function to be implemented.

The *evaluate*-method includes a single argument containing the state quantities, i.e. the predicted quality attributes of the managed system provided by Palladio (or rather its simulation and analysis tools) or provided by external tools, e.g. Prism. Based on the predictions, a software engineer determines the resulting reward, e.g. by checking quality objective violations. Also, quality objective preferences can be encoded within the reward function implementation. Similar to the adaptation strategy, note that the excerpt from the listing 6.2 is a snapshot that possibly changes over time.

6.3.2. Evaluating Adaptation Strategies by Generating Simulated Experience

In the last section, we outlined the (predominantly model-based) representation of the self-adaptive system concepts that are relevant for evaluating adaptation strategies. In this section, we discuss in more detail how the individual concepts are mutually interacting to generate trajectories or simulated experience.

Therefore, consider algorithm 6.1 which shows the main procedure of the *SimExp* framework, i.e. the procedure for probabilistically generating trajectories. *SimExp* inputs the initial architecture model M_{C_0}, environment model $M_{\mathcal{E}}$, a set of model transformations denoted as M_{Δ}, the implemented reward function $r_{\mathcal{S}}$ and adaptation strategy π. Moreover, some configuration-specific parameters are passed as well: The number of trajectories to sample (i.e. n) and the horizon (i.e. the final step T of a trajectory). First, a list (denoted as $\mathcal{T}_{\pi}^{*}$) is initialised. The list maintains all sampled trajectories where each trajectory is denoted as τ. Note that we reuse notations of section 4.4.1, i.e. $\mathcal{T}_{\pi}$ that describes the subspace of the trajectory space $\mathcal{T}$ induced by a policy π and a trajectory τ. In terms of algorithm 6.1, $\mathcal{T}_{\pi}^{*} \subseteq \mathcal{T}_{\pi}$; depending on the size of n not all trajectories might be sampled, but a representative subset captured by $\mathcal{T}_{\pi}^{*}$. Also, note that we originally defined a trajectory to be a sequence of states; technically, however, we do not merely consider the states but also the corresponding adaptations (applied in a given state) and the achieved reward for taking said adaptation.

The algorithm contains two main loops, an outer and an inner loop. The outer loop is controlled by parameter n and executes the inner loop until all trajectories are sampled (specified by n). The inner loop is controlled by horizon T, i.e. the number of states to sample or the length of each trajectory. The main logic of the inner loop, however, is to generate simulated experience, i.e. states, adaptations, state transitions and rewards.

Each trajectory starts with sampling the initial state. The initial state sampling is captured by the code block following the *if*-statement when condition $i = 0$ evaluates to true. Because a state consists of an architectural configuration and environmental state, both variables must be generated (i.e. an initial architectural configuration C_0 and an initial environmental state E_0). The initial architectural configuration refers to the initial architecture model M_{C_0} passed to the algorithm. The initial environmental state E_0 is generated by sampling from the initial distribution captured by the BN (Bayesian network) $\mathcal{B}_0$ which is part of the environmental model $M_{\mathcal{E}}$. Both variables form the initial state and are appended to τ (which has been initialised before).

In all other cases (i.e. $i > 0$) the *else*-block of the inner loop is executed. Hereby, the last state S_{i-1} appended to τ is retrieved as it serves as a basis to determine the adaptation, reward and next state. Thus, the adaptation δ_{i-1} is selected by triggering adaptation strategy π w.r.t. S_{i-1}, i.e. by invoking the code which implements the interface shown in listing 6.1. Afterwards, the selected adaptation δ_{i-1} is applied by executing adaptation process $\phi(C_{i-1}, \delta_{i-1})$; that is, the model transformation representing δ_{i-1} is applied to architectural model $M_{C_{i-1}}$. As a result, we obtain the next architectural configuration C_i.

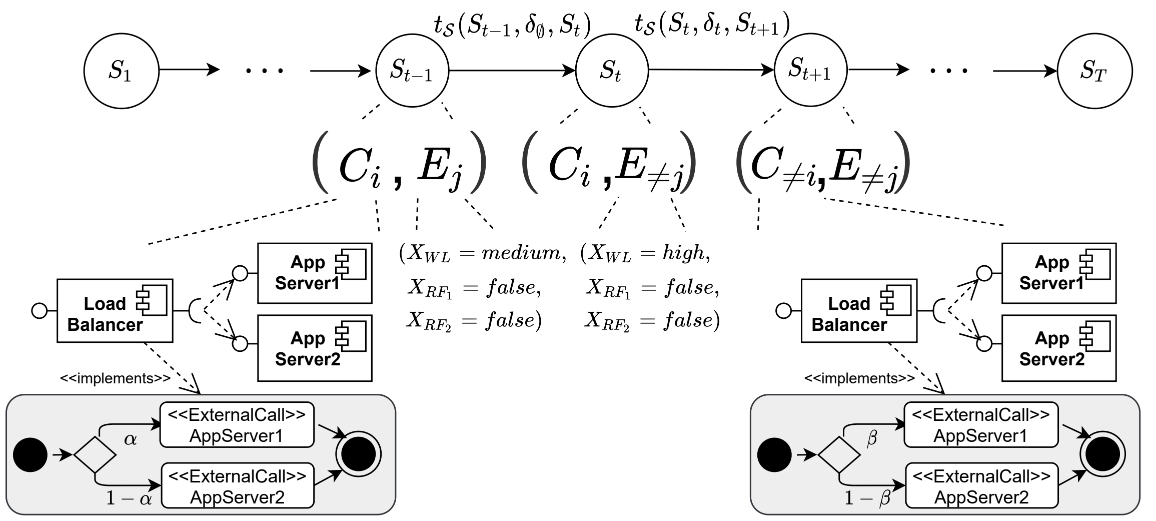
Algorithm 6.1: Core process of *SimExp*: probabilistic trajectory sampling

```
Input: The policy to be evaluated $\pi$,
reward function $r_{\mathcal{S}}$,
initial architecture model $M_{C_0}$,
environment model $M_{\mathcal{E}}$,
model transformations $M_{\Delta}$,
number of trajectories to sample $n$,
horizon $T$
Output: Estimated quality of strategy $\pi$, i.e. $\mathbb{E}_{\pi}[X_{G_t}]$
$\mathcal{T}_{\pi}^{*} \leftarrow emptyList()$ // initialise with empty list
while number of trajectories n is not reached do
    $\tau \leftarrow emptySequence()$ // initialise with empty sequence
    forall $i < T$ do
        if $i = 0$ then
            $E_0 \leftarrow x \sim \mathcal{B}_0 \in M_{\mathcal{E}}$ // sampling of initial environmental state
              from initial distribution $\mathcal{B}_0$
            $C_0 \leftarrow M_{C_0}$

            $append(\tau, (E_i, C_i))$
        end
        else
            $E_{i-1}, C_{i-1} \leftarrow lastState(\tau)$ // i.e. $S_{i-1}$
            $\delta_{i-1} \leftarrow \pi(S_{i-1})$ // w.r.t. $M_{\Delta}$
            $C_i \leftarrow \phi(C_{i-1}, \delta_{i-1})$
            $E_i \leftarrow sampleNext(S_{i-1}, M_{\mathcal{E}})$
            $r_i \leftarrow r_S(S_{i-1}, \delta_{i-1}, S_i)$ // $S_i := (E_i, C_i)$

            $append(\tau, \delta_{i-1})$
            $append(\tau, r_i)$
            $append(\tau, S_i)$
        end
    end
    $append(\mathcal{T}_{\pi}^{*}, \tau)$ // appends sampled trajectory $\tau$
end
return $estimateValueFunction(\mathcal{T}_{\pi})$
```

The next environmental state E_i is sampled w.r.t. S_{i-1}. For now, we abstract the details by the *sampleNext*-method and defer the discussion to section 6.3.2.1 because it relates to the interdependency of architecture and environment that we discussed in section 4.3.2. After determining the environmental state E_i (which completes state S_i), one can compute the reward r_i by applying reward function $r_{\mathcal{S}}$. Adaptation δ_{i-1}, reward r_i and state S_i are appended to τ which is appended to $\mathcal{T}_{\pi}^{*}$ after T runs of the inner loop. Finally, $\mathcal{T}_{\pi}^{*}$ is further



$t_{\mathcal{S}}(S_{t-1}, \delta_{\emptyset}, S_t)$
$t_{\mathcal{S}}(S_t, \delta_t, S_{t+1})$
S_1
S_{t-1}
S_t
S_{t+1}
S_T
(C_i, E_j)
$(C_i, E_{\neq j})$
$(C_{\neq i}, E_{\neq j})$
$(X_{WL} = medium, X_{RF_1} = false, X_{RF_2} = false)$
$(X_{WL} = high, X_{RF_1} = false, X_{RF_2} = false)$
Load Balancer
App Server1
App Server2
<<implements>>
<<ExternalCall>> AppServer1
<<ExternalCall>> AppServer2
α
$1-\alpha$
β
$1-\beta$

Figure 6.2.: A probabilistically sampled trajectory of the load balancer example system adopted from [158].

analysed to estimate the expected reward of strategy π. Again, we abstract the details by the *estimateValueFunction*-method and defer the discussion to section 6.3.2.3.

Note that lines 12-16 of the algorithm reflects the logic of sampling rewards and states from distribution $p(S', r|S, \delta)$. Formula (6.2) indicates how $p(S', r|S, \delta)$ factorises into two products where it can be seen that only the next state (w.r.t. $t_{\mathcal{S}}$) must be sampled as the reward function is deterministic. More generally, the entire logic of the inner loop reflects the sampling process of Monte Carlo prediction (as discussed in section 6.2.2 and illustrated by (6.7)). The trajectories are sampled w.r.t. the decisions made by strategy π and as dictated by the Bellman equation (6.3) (or their adjusted version for self-adaptive systems).

Example 6. Consider the load balancer example system from section 1.5.1. In the load balancer example system, the adaptation problem is about adjusting the distribution factor (which determines how the incoming load is distributed onto two application servers) to keep the response time and resource utilisation of the system as low as possible in the presence of uncertainties such as varying workloads and resource failures. For simplicity, we discretise the workload into three levels, namely *low*, *medium*, *high*. The resource failure of a server is described as a binary random variable.

Suppose a predefined adaptation strategy π that has been developed by a software engineer and which is supposed to be evaluated by the *SimExp* framework. In addition, let us assume that all required models are specified and all required interfaces implemented. Now, consider Figure 6.2 which depicts an example trajectory of the load balancer system following strategy π sampled by the *SimExp* framework.

It shows a possible trajectory through the load balancer system state space and illustrates two possible state transitions, i.e. from S_{t-1} to S_t and from S_t to S_{t+1}. More precisely, S_{t-1}

to S_t illustrates a transition where only the environmental state is changing; that is, the system has not been adapted by the adaptation strategy ($\pi(S_{t-1}) = \delta_\emptyset$) at time $t-1$, but the environmental state (or more specifically, the workload) has changed at time t. The environmental state change is determined by the *sampleNext*-method which we discuss in section 6.3.2.1. From S_t to S_{t+1}, one can observe an adaptation of the architectural configuration. The environmental state transition from time $t-1$ to t could have resulted in a self-adaptive system state that violates the quality objectives. As the workload variable increases to a higher level, the response time of the system increases equally such that the strategy π triggers an adaptation in response to the environmental change, i.e. by adapting the distribution factor. As the *SimExp* framework uses the prediction tools of the *Palladio* framework, the increased workload is also noticeable in the predicted response time of the system. When the adaptation strategy is invoked again, the increased response time is detected and a corresponding adaptation, i.e. a model transformation, is planned. The model transformation is carried out in such a way that a new architecture configuration (represented by the transformed architecture or the PCM model) is created that indicates an improvement in the predicted response times. ■

So far, we deliberately neglected detailed discussions about how the *sampleNext*-method, *estimateValueFunction*-method of algorithm 6.1 are implemented and how we make use of software quality prediction approaches to compute rewards. This is done in the subsequent sections.

6.3.2.1. Encoding Interdependency Assumptions of Software Architecture and Environment

In the last section, we presented the core process of the *SimExp* framework (see algorithm 6.1). Hereby, we abstracted the details of the procedure for sampling environmental states by the *sampleNext*-method. In this section, we discuss the internals of said method and how it relates to the interdependency of software architecture and the environment from section 4.3.2.

Therefore, recall theorem 4.3.1 on page 83 which states that the transition function of a self-adaptive system factorises to:

$$t_{\mathcal{S}} = P(X_{\mathcal{S}_{t+1}} \mid X_{\mathcal{S}_t}, X_{\Delta_t}) = P(X_{\mathcal{E}_{t+1}} \mid X_{C_t}, X_{\mathcal{E}_t}) \cdot \mathbb{1}_{\phi(C_t,\Delta_t)=C_{t+1}}$$

Moreover, in the last section, we discussed how the core process of *SimExp* follows the Bellman equations (with the difference that we sample from $t_{\mathcal{S}}$ instead of summing over all possible subsequent states). If we substituted the transition function $t_{\mathcal{S}}$ of the adjusted Bellman equations (6.3) from page 135, we obtain:

$$\begin{aligned} \forall S := (C,E) \in \mathcal{S} : v_\pi(S) = \sum_{S':=(C',E')} & Pr(X_{\mathcal{E}_{t+1}} = E' \mid X_{C_t} = C, X_{\mathcal{E}_t} = E) \\ & \cdot \mathbb{1}_{\phi(C,\pi(C))=C'} \Big(r_{\mathcal{S}}(S,\delta,S') + \gamma v_\pi(S') \Big) \end{aligned} \tag{6.8}$$

As mentioned earlier, the core process of *SimExp* samples trajectories according to the dynamics described by the Bellman equations or their adjusted version (6.8). That is, it is not summed over all possible subsequent states S' but sampled from $t_{\mathcal{S}}$ which factorises into two products according to theorem 4.3.1. Hereby, the *sampleNext*-method implements the procedure of sampling from $P(X_{\mathcal{E}_{t+1}} \mid X_{C_t}, X_{\mathcal{E}_t})$.

However, the dynamics of the distribution depend on the domain in which the self-adaptive system operates. Furthermore, at design-time it may be unclear what the exact distribution of $P(X_{\mathcal{E}_{t+1}} \mid X_{C_t}, X_{\mathcal{E}_t})$ looks like. Therefore, assumptions must be made. By default, it is assumed that the environment is conditionally independent of the software architecture (or architectural configuration) if the last environmental state is given:

$$(X_{\mathcal{E}_{t+1}} \perp\!\!\!\perp X_{C_t} \mid X_{\mathcal{E}_t}) \tag{6.9}$$

Based on that assumption the distribution $P(X_{\mathcal{E}_{t+1}} \mid X_{C_t}, X_{\mathcal{E}_t})$ simplifies to $P(X_{\mathcal{E}_{t+1}} \mid X_{\mathcal{E}_t})$. For the load balancer example system, this might be a reasonable assumption because regardless of how the distribution factor is adapted, it has probably no effect on how the workload and resource failure variables evolve. In section 4.3.2, however, we discussed that assumption (6.9) does not hold for the DeltaIoT system. Therefore, domain-specific assumptions about the distribution must be encoded in some way to account for them during the sampling process of the *SimExp* framework.

Currently, in the *SimExp* framework, domain-specific assumptions (i.e. domains where (6.9) does not hold) are implemented by using a dedicated interface. An alternative approach, however, would be to encode the interdependency assumption directly in the environment model, i.e. $M_{\mathcal{E}}$. That is to say, dedicated architectural template variables must be considered.

Recall from chapter 5 that the core of the *EnvDyn* metamodel are template variables which encode the probabilistic structure of domains at the type level. At this point, the template variables are extended by the architecture-specific templates, which are related to the environment-specific template variables on which they have an influence. Afterwards, the template variables are instantiated in the architecture model as usual and the respective probability distributions are specified accordingly. Thus, changes in the architecture model (triggered by model transformations) are considered by the expanded environment model $M_{\mathcal{E}}$, if the architectural change influences the environment. Finally, $P(X_{\mathcal{E}_{t+1}} \mid X_{C_t}, X_{\mathcal{E}_t})$ is evaluated by sampling from the modelled distributions described by $M_{\mathcal{E}}$.

Example 7. In the DeltaIoT system, an architecture-specific template might refer to the transmission power as it directly affects the SNR (signal-to-noise ratio) of a communication link. That is, a template variable associated with the transmission power must be created and added to the existing (and environmental-specific) template variables. Moreover, the transmission power template must be related to the SNR template variable to indicate that the transmission power influences the SNR. Subsequently, the templates are instantiated in the architecture model as usual, i.e. the transmission power template refers to the architectural elements that represent the transmission power. Thus, adaptations (or model transformations) that adapt the transmission power are considered by the expanded environment model such that the next environmental state is determined w.r.t. the modelled probability distributions (described by the *ProbDist* metamodel). ■

6.3.2.2. Using Software Quality Prediction to Determine Rewards

The quality of an adaptation strategy is determined by evaluating how good quality objectives are maintained. For this purpose, at each time t the reward function r_S evaluates the decisions made by the strategy. Afterwards, the generated rewards are evaluated to estimate the expected reward of a strategy (details follow in section 6.3.2.3). In this section, however, we discuss how particular rewards are generated by using quality prediction tools (such as those provided by the Palladio framework).

Based on predicted quality attributes, a reward signal or value is determined. The calculation of a reward is context specific because it depends on (i) the quality objectives specific to that context and (ii) their prioritisation. For instance, in the load balancer example system, the quality objectives that must be dealt with are response time and resource utilisation. The Palladio framework provides prediction tools to predict the response time and resource utilisation of a modelled PCM instance. That is, for each state, quality predictions are made which serve as a basis to generate a reward signal. For example, a trivial reward generation procedure could simply return a positive reward (e.g. +1) when all quality objectives are satisfied or a negative reward signal (e.g. -1) whenever one of the predicted quality attributes is violated. In some application contexts, however, a software engineer prefers some quality attributes more than others. For example, in terms of the load balancer system, one may prefer to keep the system rather more responsive than economising resource utilisation. In this case, the reward function can be adapted in the sense that +1 is returned when the response time and resource utilisation attributes are satisfied, 0 if at least the response time is satisfied and -1 otherwise. As a result, adaptation strategies that keep the system responsive are likely to generate a larger expected reward signal.

In this thesis, we use PCM as ADL to model software systems. Again, we want to stress that our approach is not limited to PCM as ADL but is rather ADL-agnostic. PCM (or the Palladio approach in general) provides a set of prediction tools that can be used to predict certain quality attributes. However, the quality predictions are dependent on the current environmental state, e.g. when the environmental variables of the load balancer example indicate a high workload, the predicted response time is high as well (assuming that the current architectural configuration is balancing the load not ideally). As the workload variable in PCM is associated with the usage model, the current environmental state (or rather the variable that is associated with the workload) must be synchronised with the current state of the usage model, i.e. the model element that describes the workload. That is, whenever the environmental variable of the workload takes a different value, the usage model of the PCM must be synchronised with said variable (similarly, this applies to resource failures). Therefore, there must be a procedure which synchronises the current environmental state with the current PCM instance. Thus, at each time step, the

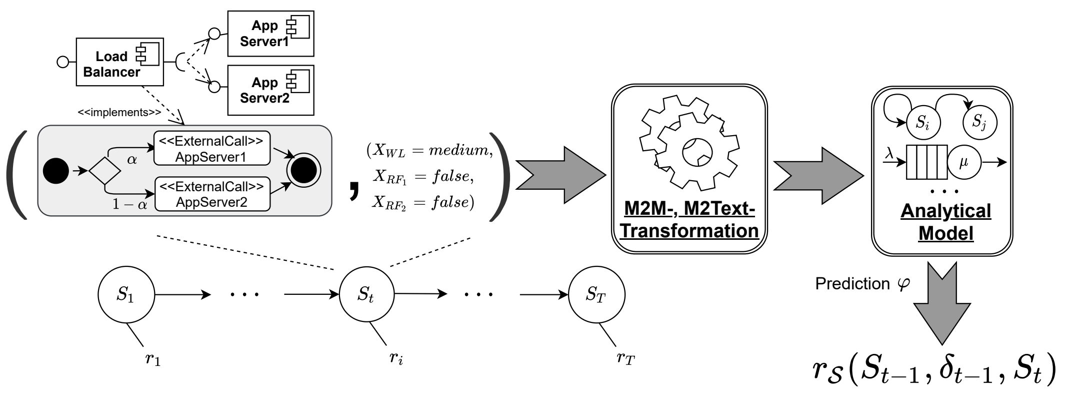

Load Balancer
App Server1
App Server2
<<implements>>
α
1 − α
<<ExternalCall>> AppServer1
<<ExternalCall>> AppServer2
$(X_{WL} = medium, X_{RF_1} = false, X_{RF_2} = false)$
M2M-, M2Text-Transformation
S_i
S_j
λ
μ
Analytical Model
Prediction φ
S_1
S_t
S_T
r_1
r_i
r_T
$r_{\mathcal{S}}(S_{t-1}, \delta_{t-1}, S_t)$

Figure 6.3.: Transformation of the environment- and architecture model to an analytical model for quality prediction and reward evaluation.

predicted quality attributes take into account the current state of the environment. This synchronisation procedure can be generalised as depicted on Figure 6.3.

Instead of thinking of a synchronisation procedure, one can consider a model transformation procedure which takes as input the current environmental state and architectural configuration (e.g. the PCM instance) and transforms both models into an analytical model. Generally, speaking for each supported ADL there might be a transformation procedure that transforms the architecture and environment model. The transformed model forms the analytical model (e.g. queuing networks, Petri nets or Prism specifications) which is used by other tools (e.g. such as Prism) to predict quality attributes. For example, in the DeltaIoT system, the considered quality attributes encompass the consumed energy of the system and packet loss. Both quality attributes cannot be predicted by the tools provided by Palladio. However, the PCM instance and environment model can be transformed (by using an M2Text transformation) into Prism specifications which are passed to the Prism tool to predict the energy consumption and packet loss of the current state. Finally, the prediction of the quality attributes is passed to the reward function (as discussed before) and forms the primary source to evaluate decisions made by an adaptation strategy.

6.3.2.3. Analysing Generated Trajectories to Evaluate Adaptation strategies

In this section, we explain how the *estimateValueFunction*-method is implemented. The sampled trajectories $\mathcal{T}_\pi^*$ serve as a basis to estimate the value function v_π and must be further evaluated. For this estimation, we make use of the first-visit Monte Carlo method that we introduced in section 2.5.2. The first-visit Monte Carlo method has been shown in algorithm 2.2. The difference between algorithm 6.1 and algorithm 2.2 is that the trajectory sampling is done beforehand, i.e. $\mathcal{T}_\pi^*$ contains all samples that can be evaluated as usual by the first-visit method to estimate v_π.

Moreover, for the final evaluation of an adaptation strategy, we use formula (6.6) which we derived in section 6.2.1.2 for the partial ordering of strategies, i.e. the expected reward of a strategy π: $\mathbb{E}_\pi[X_{G_t}]$. More specifically, we are interested in the expected reward of strategies starting at time $t = 0$, i.e. $\mathbb{E}_\pi[X_{G_0}]$. As a result, the expected reward is calculated by considering only the initial states $\mathcal{S}_0 \subset \mathcal{S}$ which forms a subset of the entire state space of self-adaptive systems (as there is only one initial architectural configuration). That is, the expected reward of a strategy is calculated as follows:

$$\mathbb{E}_\pi[X_{G_0}] = \sum_{S \in \mathcal{S}_0} Pr(X_{\mathcal{S}_0} = S) \cdot v_\pi(S) \tag{6.10}$$

Thus, only trajectories with probabilities greater than zero are considered. This is reasonable because the expected reward of trajectories that start in states $S \notin \mathcal{S}_0$ is zero (as they can never occur). It could be argued that this is insufficient because only a subset of states is considered in the value function and does not reflect the overall value of the strategy; however, the recursive nature of the Bellman equations shows that other states are also visited such that their values are implicitly considered. Moreover, it is the quality of a strategy to move to states that generate a higher expected reward (in terms of the value function).

Finally, note that although the *estimateValueFunction*-method specifically estimates the value function v_π of strategy π w.r.t. its generated trajectories $\mathcal{T}_\pi^*$, we can also generalise the method. More specifically, based on the trajectory space, one can apply other quality measures to evaluate π such as the total accumulated reward generated by π. Nonetheless, we use $\mathbb{E}_\pi[X_{G_0}]$ as quality measure but would like to emphasise that the *SimExp* framework is not restricted to $\mathbb{E}_\pi[X_{G_0}]$.

6.4. Implementation

In this section, we briefly provide an overview of the implementation details of the *SimExp* framework. Therefore, consider Figure 6.4 which shows an excerpt of the dependency graph of the implemented components of *SimExp*. For clarity, however, note that Figure 6.4 provides only an excerpt with a simplified view of the component structure; in fact, more components are involved. Therefore, we aggregated related components into a single component (e.g. `SimExp.PCM.*`).

Basically, the `SimExp.Markovian` component implements all relevant concepts of MDPs which are reused in the `SimExp.Core` component. The `SimExp.Core` component provides the basic and ADL-agnostic functionality (which encompasses the quality attribute-based reward function, Monte Carlo prediction, representation of actions as model transformations, etc.) of the *SimExp* framework. Therefore, the `SimExp.PCM.*` components provide the corresponding implementations to use *SimExp* in conjunction with PCM as ADL. Finally, the `SimExp.Workflow` component encapsulates the logic to structure the execution of *SimExp* based on several job implementations.

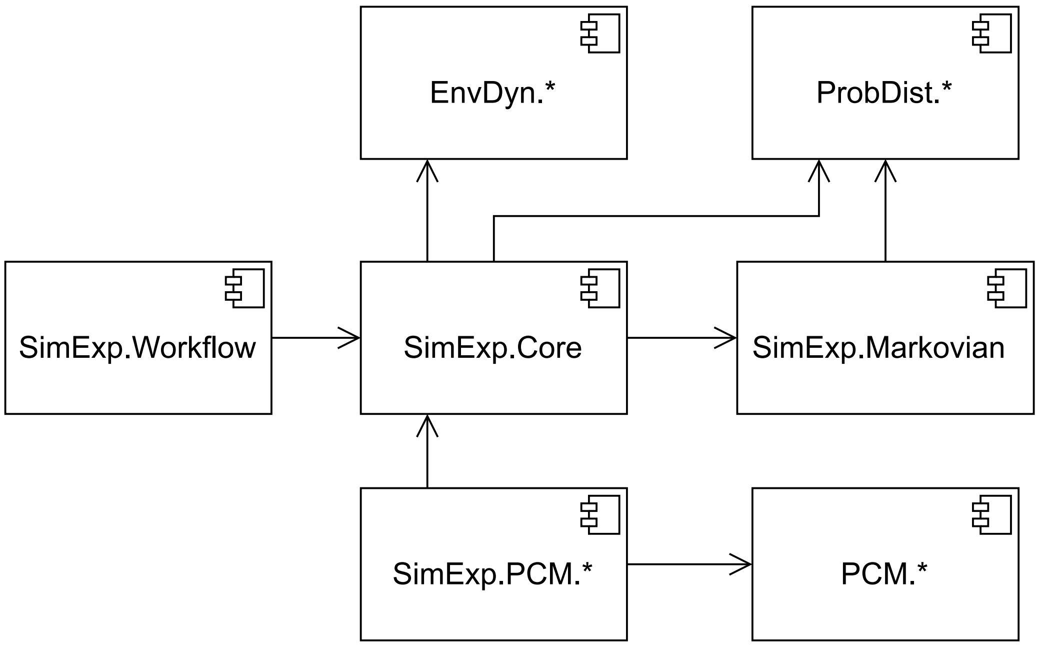


Figure 6.4.: Dependency graph of the implemented *SimExp* components.

6.5. Assumptions and Limitations

In the following, we discuss the assumptions and limitations of the *SimExp* framework.

Assumptions In DP, there is the assumption of a perfect environment model (i.e. the MDP). Although this is a strong assumption, the underlying concern of DP is optimisation. The primary concern of our *SimExp* method, however, is evaluation; that is, we are merely interested in deriving a quality value for a strategy to give them an order of magnitude and to make them comparable. As we evaluate at design-time, where inaccuracies are accepted anyway due to abstraction, it is acceptable to assume imperfect models. In addition, reinforcement learning methods greatly build upon DP without assuming perfect models and yet are successfully used in various contexts.

In section 6.3.2.2, we discussed the general transformation procedure which takes the current architecture model and environment model and transforms them into an analytical model for predicting quality attributes. Hereby, we assumed that the information provided by both models is sufficient to derive a respective analytical model (e.g. Prism specifications). However, this is a weak assumption because the *SimExp* method is conceptually ADL-agnostic. That is, if one may use a particular quality prediction tool (say one that cannot be derived by using PCM as ADL), a different ADL can be used which contains the required information for the transformation procedure.

Limitations Currently, the reward function, adaptation strategy and interdependency assumptions regarding the architecture and environment are not represented by models but merely considered by implementing dedicated interfaces. That is, a software engineer must be familiarised to some extent with the implementation details. However, in future work, we plan to reuse existing models to represent also adaptation strategies, reward functions and interdependency assumptions with models. For example, as our approach is quality-driven, in the work of Becker [15, P.87] a metamodel for describing service level objectives (e.g. tolerance ranges, violation ranges, etc.) is of particular interest and can be potentially reused to model reward functions. Regarding the interdependency assumptions, we already discussed in section 6.3.2.1 how the environment model can be expanded to encode such assumptions. Regarding the model-based description of adaptation strategies, we have already envisioned a formal domain-specific language for describing MAPE-K-based adaptation strategies [147], which is the subject of future work.

Just as the environmental dynamics, the *SimExp* method suffers the state space explosion problem as the architectural configuration space and the state space of the environmental dynamics grow exponentially. We address the state space explosion problem by using (i) abstraction and (ii) Monte Carlo methods. The former is addressed by using architecture models to focus only on the necessary details (i.e. the architectural elements which affect the quality of the system) which reduces the configuration space. In addition, by discretising an environmental state to an arbitrary level, the number of environmental states is reduced as well (recall the discretisation level property of the *EnvDyn* metamodel from section 5). Finally, we employ Monte Carlo methods to sample representative trajectories of the trajectory space to estimate the overall quality of an adaptation strategy. By considering the previously given arguments, we argue that for most of the domains the state space explosion is sufficiently addressed. However, as there are no guarantees there might be systems and domains that indicate such a high degree of complexity which makes it impossible to deal with them at design-time. We discuss how to deal with such domains in chapter 8.

In the work of Stier [178], so-called *Transient Effects* of self-adaptive systems are described which relate to execution times and consumed resources of applied adaptations. In other words, transient effects describe costs associated with adaptations. Moreover, Stier presents an approach to how such transient effects can be analysed at design-time. Our *SimExp* method, however, does not yet consider transient effects because they are not in the scope of this thesis but are subject to future work.

Finally, it is worth mentioning that the presented *SimExp* framework is supposed to be considered as a method. That is, if the approach is instantiated in a specific domain (e.g. IoT), some domain-specific extensions must be done. For the DeltaIoT system, for instance, we had to extend the *SimExp* framework to analyse the packet loss and energy consumption (as we will see in chapter 9). However, as our primary concern is to evaluate adaptation strategies that safeguard AI black-box components no more extensions as the ones presented in this work are required.

6.6. Summary

In this chapter, we presented the *SimExp* method. First, we discussed in section 6.1 how we generally evaluate adaptation strategies at design-time. In section 6.2, we presented the formal semantics of the *SimExp* method. Afterwards, we discussed in section 6.3 how the formal semantics are implemented by using methods from MBQA. Hereby, we indicated how all relevant concepts are represented by models (or at the code level) and how they are interacting to sample trajectories for estimating the quality of an adaptation strategy. In section 6.4, we discussed implementation details of the *SimExp* framework. Finally, we discussed assumptions and limitations in section 6.5.

Part IV.

Safeguarding Uncertain AI Black-Box Components

7. Reliability Prediction of Architectural Safeguards for AI-enabled Systems

In this chapter, we discuss our reliability prediction approach to evaluate architectural safeguards for AI-enabled systems at design-time. Within a software system, an AI component has specific responsibilities (e.g. object recognition) on which other components depend (and thus on the correctness of the prediction results). Therefore, the consequences of an incorrect prediction manifest themselves in other parts of the system and have global effects. Furthermore, Dreossi et al. [54] pointed out that incorrect predictions do not necessarily force the system to fail: In automatic braking systems, for example, a sufficiently distant car that has not been correctly detected by an AI-based object recognition component has no safety-critical impact at that moment. Therefore, we focus on reliability attributes of the system as a whole. For this purpose, we start to explain how the reliability approach is applied to static software systems safeguarded by non-adaptive approaches (e.g. architectural patterns) and generalise the approach to self-adaptive systems afterwards. The contribution, presented in this chapter, is based on the publication [160, 159]. In addition, section 7.1 is based on the Master's thesis of Dennis Marvin Bäuml [14] which was supervised by the author of this thesis.

In the previous chapters, we discussed how to represent the environmental dynamics and analysed self-adaptive systems by using model-based techniques. Now, we focus on software systems with AI components where self-adaptive systems are specifically considered as safeguards to deal with potential erroneous behaviour to which AI components are susceptible. Thus, the concepts presented in this chapter build strongly upon the concepts of the previous chapters, i.e. *SimExp* framework and *EnvDyn* metamodel. More specifically, we use the *EnvDyn* metamodel not only to represent the environmental dynamics of the system but also to model the uncertainties or environmental variables (which are, in fact, the same thing) that affect the predictive uncertainty (i.e. the failure probability of the prediction) of the AI component. Moreover, we instantiate the *SimExp* method from chapter 6 to evaluate adaptation strategies that are supposed to safeguard AI black-box components. For this purpose, however, we need to take a step back and develop additional concepts for evaluating the reliability of AI-enabled static software systems that complement the *SimExp* framework in the next step; this refers to research question **RQ2**:

Research Question 2: How can software systems that contain AI black-box components be evaluated in terms of meeting reliability attributes at design-time?

One of the major problems in connection with safeguarding AI black-box components is that we are not able to determine whether a prediction is correct or not as we cannot observe the true state; we call this the *Hidden State Problem*. However, the incorrect prediction is propagated to the other components of the system which rely on the prediction. In the HRI example system from section 1.5.3, for instance, the trajectory planning component highly depends on the AI-based object detection component to avoid collisions while computing the trajectory. This means that before analysing adaptation strategies, one must first consider how to deal with the fact that the state of the AI components is not observable. This directly refers to research question **RQ2.1**:

Research Question 2.1: How to deal with the hidden state problem of AI black-box components?

The black-box property of AI components makes it difficult not only to safeguard them at runtime but also to analyse them at design-time. For example, how can one analyse the reliability attributes of an AI-enabled system when the true state of the AI component is hidden? How are AI black-box components included in reliability prediction when the input data for which they produce faulty behaviour is unknown? Moreover, Seshia et al. [165] enumerated several challenges that make modelling AI-enabled systems challenging. One of these challenges relates to high-dimensional input spaces that one encounters when dealing with AI (or deep learning). The input space is crucial for how reliably an AI component works and must therefore be taken into account to some extent in the analysis. However, high dimensional input spaces, such as the pixel space in perception tasks, are way too large to be analysed. Dealing with the aforementioned problems relates to research question **RQ2.2**:

Research Question 2.2: How to systematically consider the influence of predictive uncertainty and causally related environmental variables in the reliability prediction?

After dealing with the problems of **RQ2**, we can generalise the concepts to evaluate adaptation strategies of self-adaptive systems that are safeguarding uncertain AI components and relates to research question **RQ3**:

Research Question 3: How can adaptation strategies of self-adaptive systems that safeguard uncertain AI black-box components be evaluated in terms of reliability at design-time?

When we speak of AI-induced uncertainty, we refer to predictive uncertainty [89], i.e. potentially erroneous predictions of an AI component that could remain undetected and propagate to the rest of the system. Moreover, we consider predictive uncertainty as first-order uncertainty. Generally, uncertainty of AI is classified into *Epistemic* and *Aleatoric* uncertainty [139, 89, 167, 163]. Epistemic uncertainty refers to the “lack of knowledge” of an AI model which can be reduced by more training data; aleatoric uncertainty relates to irreducible random phenomena in the input data of an AI model, e.g. such as distinct weather conditions or noise. Therefore, we also consider uncertainties that can be either epistemic or aleatoric, but which directly affect predictive uncertainty, and refer to these as second-order uncertainties. Such uncertainties possibly refer to factors observed in the environment or derived from the input data which allow conclusions to be drawn about the state of an AI component. To avoid confusion, we note that we use the terms uncertainties and properties interchangeably. In section 7.1, we rather use the term uncertainties and in section 7.2 we use the term properties. The reason for this is that we begin to discuss architectural means for dealing with AI-related uncertainties of second order in static software systems. We then discuss how the concepts can be generalised to self-adaptive systems (section 7.2). In this case, we observe properties to draw conclusions about the state of an AI component and plan appropriate adaptations based on the observations. However, both concepts describe the same thing but may differ in the way we view them.

The chapter is organised as follows: In section 7.1, we present an approach to predict reliability attributes of AI-enabled systems. Hereby, we apply an upstream sensitivity analysis to capture the predictive uncertainty of an AI component by the resulting sensitivity model. Additionally, we use ATs (architectural templates) to model non-adaptive architectural safeguards (i.e. architectural patterns) that one can consider dealing with AI-related uncertainties. The effect of an AT on the reliability attributes of the system is analysed by our provided prediction approach. In section 7.2, we generalise the concepts to self-adaptive systems. We reused the concepts of the *SimExp* method from chapter 6 and extended it to evaluate adaptation strategies safeguarding uncertain AI components. In section 7.3, we briefly present the implementation details of the approach. Finally, section 7.4 discusses the assumptions and limitations of the approach, while section 7.5 summarises the chapter.

7.1. Engineering Reliable AI-Enabled Systems in the Presence of Uncertainty

In this section, we discuss a model-based approach to predict reliability attributes of static AI-enabled software systems. To model static architectural safeguards, we reuse a template method to describe reusable architectural patterns or styles for reoccurring problems, e.g. safeguarding an AI component. For this purpose, we reuse the formal language of ATs from Lehrig [113] (see section 2.3.1.2). ATs are completely compatible with the Palladio framework, i.e. they can be applied to PCM instances. That is, they are perfectly suited in the context of this work to capture architectural patterns dealing with AI-specific uncertainties. The overall approach is depicted on Figure 7.1. In section 7.2, we generalise the presented concepts for self-adaptive systems.

The approach is essentially divided into three parts:

- **Architectural knowledge representation:** A collection of architectural patterns described by a formal template language for specifying architectural knowledge.

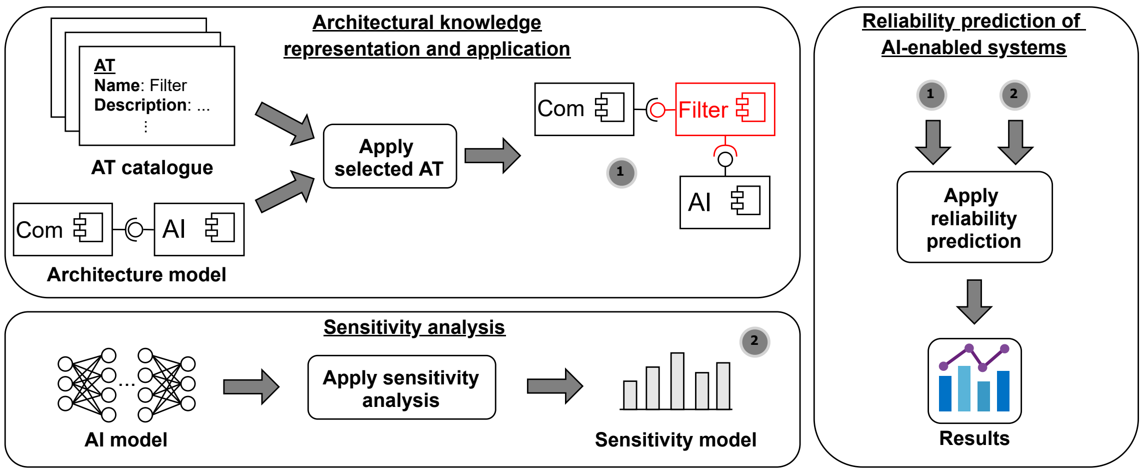

Architectural knowledge representation and application
AT
Name: Filter
Description: ...
AT catalogue
Apply selected AT
Com
Filter
AI
Architecture model
Reliability prediction of AI-enabled systems
1
2
Apply reliability prediction
Results
Sensitivity analysis
AI model
Apply sensitivity analysis
Sensitivity model

Figure 7.1.: Overview of the reliability prediction approach for AI-enabled systems. The reliability prediction inputs (i) an architecture model (PCM model) enriched by an AT and (ii) the sensitivity model of an AI component. Based on the inputs, the success and failure probabilities of the system are predicted.

- **Sensitivity analysis:** An upstream sensitivity analysis to determine a sensitivity model capturing the predictive uncertainty of an AI component.
- **Uncertainty-based reliability prediction:** An approach for predicting reliability attributes of AI-enabled software systems by taking into account AI-induced uncertainties.

As mentioned before, for the first part, we employ ATs to represent architectural knowledge, e.g. n-version programming pattern for deep neural networks [211]. Based on a collection of architectural patterns (i.e. a catalogue of ATs), one can select an appropriate AT for the given application context and apply it to the modelled architecture model, i.e. the PCM model. In parallel, a sensitivity analysis of the AI model is performed. The sensitivity analysis forms the second part of the approach and addresses the problem of representing AI components in the reliability prediction process. The result of the analysis is a sensitivity model represented by a probability distribution that describes the likelihood of observing correct or incorrect predictions (i.e. predictive uncertainty) of the AI model in the presence of uncertainties such as increased illumination, bad weather conditions or blurring in perceptual learning tasks. Based on the AT-enriched PCM model and the sensitivity model, reliability attributes are predicted by an uncertainty-based reliability prediction approach which forms the third part of the approach. Here we extend the reliability prediction tool of the Palladio framework, PCM-Rel (see section 2.3.2.2). Roughly speaking, we extend PCM-Rel by considering the predictive uncertainty of an AI component as a type of software-induced failure that can be assigned a failure probability. The failure probability is derived from the sensitivity model and a given set of uncertainty values that have an impact on the predictive uncertainty.

Finally, it should be noted that we are reusing an existing approach for architectural knowledge representation and also not developing any novel sensitivity analyses (but referencing existing approaches), so neither should be understood as contributions. Instead, the first contribution is a holistic approach that unifies the sub-approaches (i.e. architectural knowledge representation/application, sensitivity analysis and reliability prediction of AI-enabled systems) to a tool that supports software engineers in the design and analysis of AI-enabled systems. The second contribution relates to the approach for predicting reliability attributes of AI-enabled systems themselves. Therefore, we discuss in the following the three parts of the holistic approach, focusing on the reliability prediction part as it forms the main contribution.

7.1.1. Represention of Architectural Safeguards with Architectural Templates

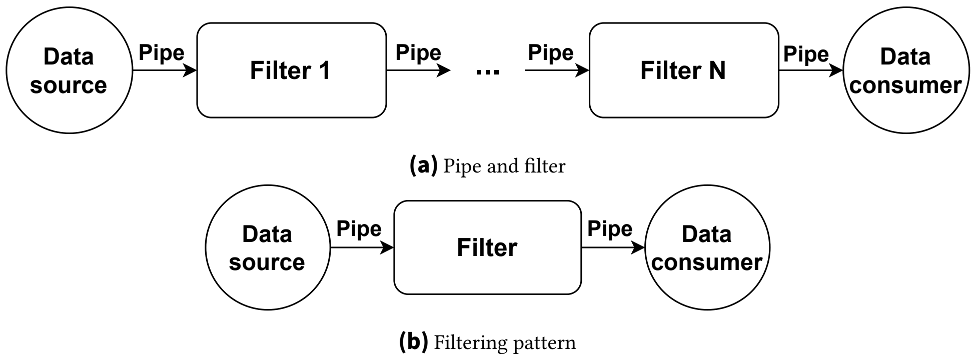
In this section, we discuss how to represent (AI-specific) architectural knowledge by using models. The notion of architectural knowledge is broad and refers to reusable structures such as reference architectures, architectural styles or architectural patterns [184]. We focus on the latter in this section. We consider architectural patterns as architectural safeguards or *Architectural Countermeasures* that either contain/mitigate AI-specific uncertainties (of second order) or reduce predictive uncertainty. The notion of architectural countermeasures is discussed in more detail in section 7.1.3.2.

In literature, several architectural patterns are discussed to enhance the reliability of AI-enabled systems (we discussed some of them in section 3.1.2.1). In this section, however, we focus on two patterns (namely *Filtering* and *N-Version Programming*) and model them as ATs. Note that we do not aim to develop novel architectural patterns for AI systems, nor do we provide a comprehensive literature review of patterns; this is not in the scope of this thesis.

7.1.1.1. Architectural Patterns for Dealing with AI-induced Uncertainties

In literature, several works propose or reuse architectural patterns to maintain quality attributes of AI-enabled systems, e.g. [211, 167, 45, 212, 23]. However, in this work we focus on two architectural patterns: filtering [145] and n-version programming [44]. We have chosen these patterns because (i) they are well researched and known in the software engineering community and (ii) their structures and application to software architectures are well documented. In the following, we briefly present the patterns.

Filtering: The filtering pattern (more commonly referred to as *Pipe and Filter* [145]) is an architectural pattern consisting of a series of pipes and filters. A filter describes a set of components which transform input to output data. A pipe connects one filter with another, a data source with a filter or a filter with a data consumer. The pipe and filter pattern is a widely used approach in software architectures that implement any type of data processing step that requires input data to be transformed into specific output data.



(a) Pipe and filter

(b) Filtering pattern

Figure 7.2.: Overview of (a) the pipe and filter architectural pattern and (b) filtering pattern.

In the context of this work, however, we consider a simplified version of pipe and filter which we refer to as the filtering pattern. Hereby, we consider merely a single filter component and two pipes which connect the filter two a given data source (e.g. a sensor) and to a data consumer (e.g. the AI component) which inputs the result of the filtering process. We simplify the pipe-and-filter approach to a single filtering process, as it is better suited to AI use cases such as [205], where filtering components are used to preprocess incoming data before passing the data to the AI component. The general structure of the filtering pattern is depicted on Figure 7.2. The main component of the filtering pattern is the filter component which is responsible for the preprocessing. The pipes abstract away the concrete communication technology (e.g. event-based) used by the data source, filter and data consumer.

N-Version Programming: The next architectural pattern we are focusing on refers to n-version programming. In the context of deep neural networks various approaches are indicating the benefits of using n-version programming in terms of improving the prediction result, e.g. [77, 211, 119].

Originally, n-version programming was developed to improve the fault-tolerance of software systems [44]. Basically, the idea is to develop *N* independent versions or components that are functionally equal, i.e. following the same specification. Since each component implements the same specification, the input and output data are equal. Thus, new input data is passed to each version which performs the computation resulting in *N* outputs. The second important concept of n-version programming refers to the voter or decision component which selects or merges the *N* output results. The implementation of the voter or decision procedure is manifold, e.g. one may implement a majority vote based on the confidence assigned to each version or by simply computing the average. The basic structure of the n-version programming pattern is depicted on Figure 7.3.

The generic nature of n-version programming makes it applicable in many application contexts. In AI, for example, *N* versions of distinct model types (e.g. DNN) are developed and trained independently. At runtime, the input data is passed to each version simultaneously, and the individual predictions are merged in the voter component to produce a more accurate prediction.

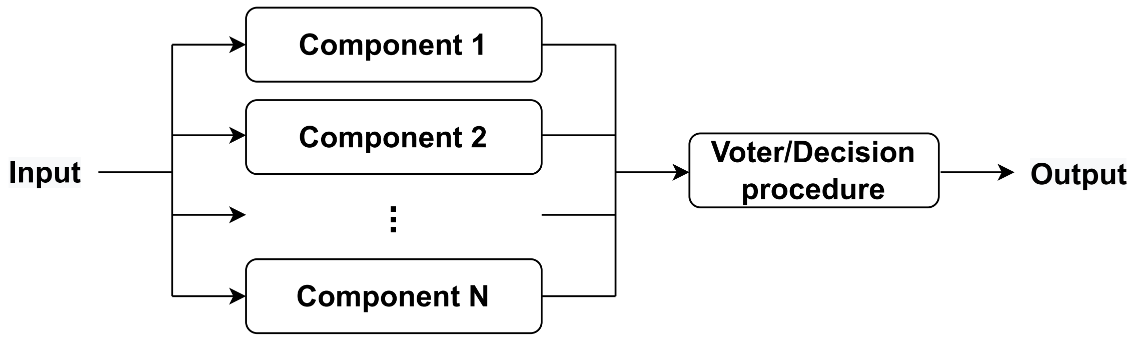


Figure 7.3.: The n-version programming pattern (based on [52]).

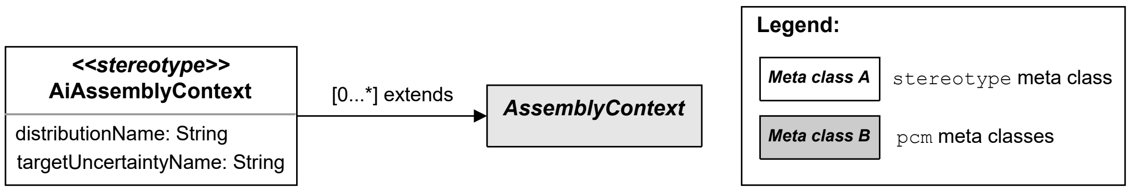


Figure 7.4.: The EMF profile of the filtering pattern.

7.1.1.2. Modelling of the Filtering Pattern as Architectural Template

In this section, we describe how we realised the AT for the filtering pattern. Recall from section 2.3.1.2 that the main tasks in modelling ATs comprise the creation of an EMF profile (which should be considered as annotations to the architectural model, e.g. PCM model) to indicate where the AT is woven in, and a completion, i.e. the model transformation that implements the weaving-in process or application of the AT.

EMF-Profil The EMF profile for the filtering AT is depicted on Figure 7.4. There is only a single stereotype which extends an `AssemblyContext` (or which can be applied to an `AssemblyContext`), namely `AiAssemblyContext`. The stereotype is to be understood as an annotation applied to the `AssemblyContext` representing the AI component (or the instantiated component in the system model). Consequently, the filter is woven in the system, i.e. the filter component is placed before the `AssemblyContext` of the AI component; reflecting the main purpose of the filter pattern, i.e. a preprocessing step to reduce possible uncertainties in the input data. The filter stereotype includes two attributes, namely `distributionName` and `targetUncertaintyName`. The attribute `distributionName` refers to a probability distribution that determines the probabilistic effect on the uncertainty that the filter reduces or contains. As we discuss the concept of architectural countermeasures and the metamodel to describe them and their influence on uncertainties in section 7.1.3.2, we do not go into further details here. It is yet sufficient to know that the attribute `distributionName` allows the resolution of the corresponding distribution that models the influence on a given target uncertainty. However, the target uncertainty is referenced by the attribute `targetUncertaintyName`. This uniquely determines which uncertainty the filter component acts on.

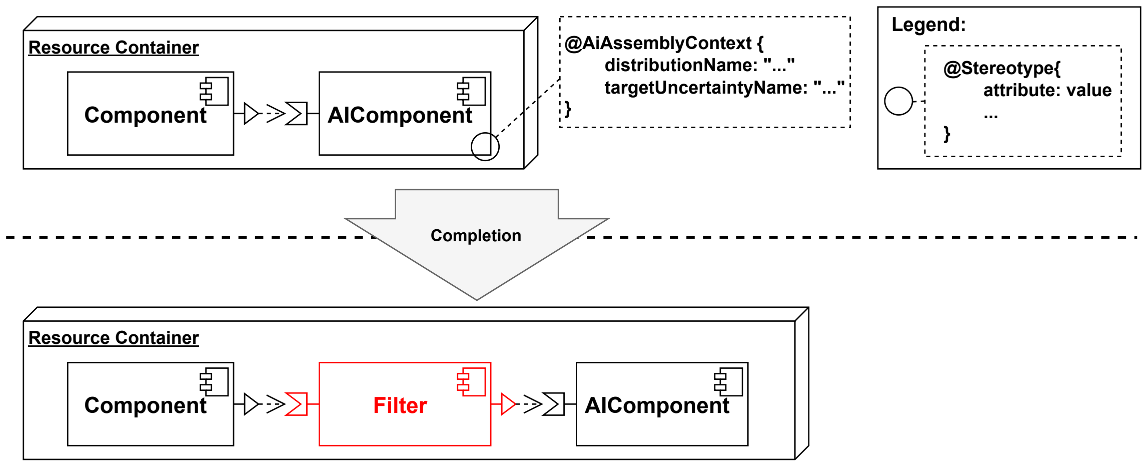


Figure 7.5.: The action of the filtering pattern completion on an annotated PCM model.

Completion The completion of the filtering pattern refers to a model transformation that weaves in the architectural pattern. Figure 7.5 schematically shows the PCM model before and after the completion. We do not go into the technical details of the model transformation but refer to [156] where the model transformation can be looked up. However, it can be seen that the filter is inserted before the AI component. Hereby, a respective `BasicComponent` instance is created, added to the repository model and instantiated in the system model. Since the filter is inserted before the AI component, it emits events (in the case of event-based communication) to the event group from which the AI component receives events. Moreover, the filter component receives events emitted by the component that was previously received by the AI component. The filter component is deployed on the same resource container where the AI component is allocated.

What the illustration does not show is the model completion of the uncertainty model that describes (among others) the architectural countermeasures and their effect on uncertainties. However, we defer the discussion to section 7.1.3 where the reliability prediction approach is explained and the interplay of all concepts is discussed.

7.1.1.3. Modelling of the N-Version Programming Pattern as Architectural Template

In this section, we describe how we realised the AT for the n-version programming pattern. Just as before, we first present the EMF profile and the completion (or model transformation) afterwards.

EMF-Profil The EMF profile for the n-version programming pattern is depicted on Figure 7.6. Just as seen in the filtering profile, the n-version programming profile defines a stereotype (namely `AiNVAssemblyContext`) that is applied to an `AssemblyContext` related to the AI component. Moreover, the stereotype holds an attribute called `improvedModelName` which corresponds to the name of a `GroundProbabilisticNetwork`. This again refers to the metamodel, which we have not yet introduced (not until section 7.1.3.2).

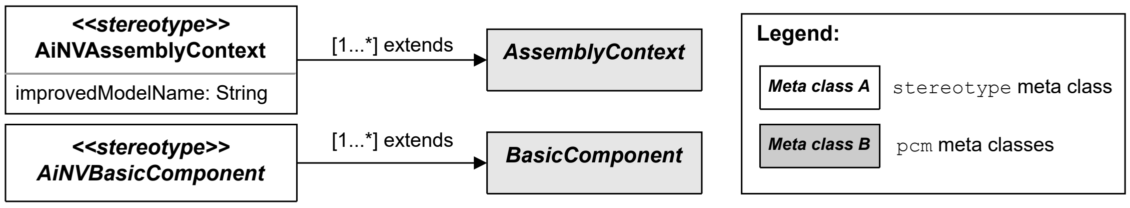


Figure 7.6.: The EMF-profile of the n-version programming pattern.

We defer the discussion again to section 7.1.3 where all concepts will be unified into a holistic approach. For now, it is sufficient to consider the referenced network name as the sensitivity model associated with the n-version programming pattern as a whole and which is more robust.

In addition to the stereotype applied to a `AssemblyContext`, there is another stereotype applied to a `BasicComponent`, namely `AiNVBasicComponent`. The stereotype is used to annotate the components in the repository of a PCM model corresponding to the different AI components to be included in the n-version programming pattern, i.e. each annotated component represents a single version.

Completion Finally, the completion or model transformation for the n-version programming pattern is depicted on Figure 7.7. Just as for the filtering pattern, we do not discuss the details of the transformation but rather refer to [156]. However, as shown in Figure 7.7, the completion substitutes the original AI component with $N + 1$, i.e. the N versions and an additional component which acts as a voter or other decision procedure. The component-internal behaviour of the decision procedure initially dispatches newly arriving input data among the N versions and receives a prediction from each version, which is eventually evaluated. In the system model, each version instantiates the AI component which has been annotated by a corresponding EMF stereotype. The components are allocated on the same resource container on which the replaced AI component was previously allocated. Again, the illustration does not represent the entire model completion (as the corresponding metamodel is presented in section 7.1.3.2). However, this will be made up for in section 7.1.3; at this point, it is more important to understand the principle operation of the completion on a PCM instance.

7.1.2. Sensitivity Analysis of AI Components

At the beginning of this chapter, we discussed the problem of not being able to observe the true state of an AI black-box component. For predicting reliability attributes of a system with AI components, however, one must also account for the reliability or confidence of the predictions made by the AI component. To approach this problem, we apply a *Sensitivity Analysis.*

Essentially, a sensitivity analysis is about varying the input of an AI component and observing how this change affects the output [76, 30] which is captured in a corresponding

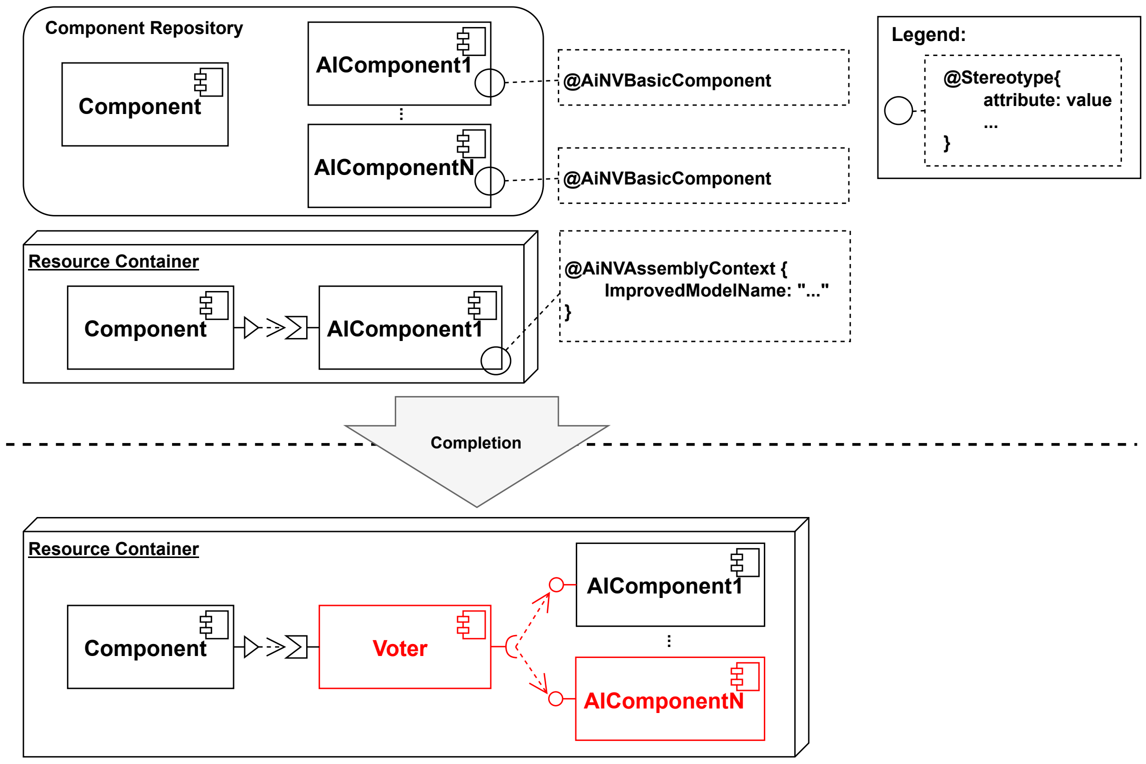


Figure 7.7.: The action of the n-version programming pattern completion on an annotated PCM model.

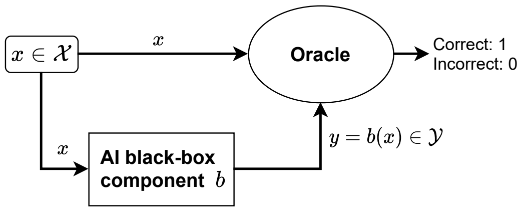
$$\begin{tikzcd}
X_{\varphi_1} \arrow[dr] & \cdots & X_{\varphi_N} \arrow[dl] \\
& X_b &
\end{tikzcd}$$

Figure 7.8.: Probabilistic structure of the sensitivity model (assuming independence between all uncertainty pairs).

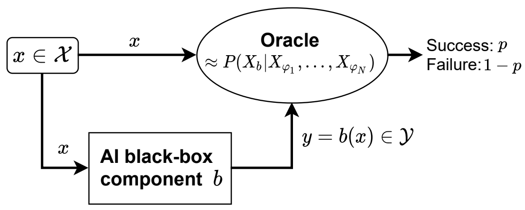
sensitivity model. Before describing the sensitivity model in more detail, we must introduce the notion of an AI black-box component more formally. Similarly to Guidotti et al. [76], we consider an AI black-box component as a function b which maps an input x from an input space $\mathcal{X}$ to an output y from output space $\mathcal{Y}$.

Definition 30 (AI Black-Box Component based on Guidotti et al. [76]). *An AI Black-Box component is a function $b : \mathcal{X} \rightarrow \mathcal{Y}$ that maps an input $x \in \mathcal{X}$ to an output $y = b(x) \in \mathcal{Y}$. We denote b as black-box to indicate that the internals are neither understandable nor interpretable by humans.*

In the context of this work, the result of the sensitivity analysis (i.e. the sensitivity model) is considered as a probability distribution or CPD (conditional probability distribution) with the following structure: $P(X_b \mid X_{\varphi_1}, \ldots, X_{\varphi_N})$.



(a) Oracle for determining the state of AI black-box b


$x \in \mathcal{X}$
x
Oracle
$\approx P(X_b | X_{\varphi_1}, \dots, X_{\varphi_N})$
Success: p
Failure: $1 - p$
x
AI black-box component b
$y = b(x) \in \mathcal{Y}$

(b) Oracle approximation by a sensitivity model w.r.t. some properties $\varphi_1, \dots, \varphi_N$

Figure 7.9.: Approximation of an oracle for AI black-box component by a sensitivity model.

The sensitivity model (captured by the CPD) is defined over the random variables X_b and $X_{\varphi_1}, \dots, X_{\varphi_N}$. The predictive uncertainty of an AI black-box component b is captured by the binary random variable X_b with value space $Val(X_b) := \{Success, Fail\}$ describing the possible events of an AI component to successfully or unsuccessfully make a certain prediction. We define the model to be a discrete BN (Bayesian network), i.e. the value spaces of all random variables are discrete. The generic structure of a sensitivity model is depicted on Figure 7.8 which assumes stochastic independence between all uncertainty pairs, i.e. $\forall i \neq j : X_{\varphi_i} \perp\!\!\!\perp X_{\varphi_j}$.

Recall that one of the main problems this work is concerned with is the inability to observe the true state of an AI black-box component b. That is, there is no oracle that tells whether a prediction made by b is correct or not. However, one can try to approximate such an oracle by estimating the probability of correct or incorrect behaviour of b based on a set of observable properties (see Figure 7.9). These properties may relate to environmental variables or other domain-specific uncertainties that allow conclusions to be drawn about the true state of b. In terms of the sensitivity model, these environmental variables or uncertainties refer to the $\varphi_1, \dots, \varphi_N$. Compared to an oracle that returns 1 (correct prediction) or 0 (wrong prediction) for any input-output pair $(x, b(x))$, the sensitivity model (as an approximation of the oracle) is queried for a given set of uncertainties $\varphi_1, \dots \varphi_N$ and returns a success probability $success \in [0, 1]$ and failure probability $1 - success$. Because the hidden state problem permits us to reason about the true state of b, we cannot estimate the predictive uncertainty (i.e. $P(X_b)$) directly; however, we can approximate $P(X_b)$ by conditioning the predictive uncertainty on the set $\varphi_1, \dots \varphi_N$, i.e. $P(X_b \mid X_{\varphi_1}, \dots, X_{\varphi_N})$.

Example 8. Recall the HRI example system from section 1.5.3 which requires human interaction such that a specific level of safety must be maintained, e.g. to prevent injuries like squeezing or collision with the robot. However, this is dependent on the AI-based object detection component because an undetected body part is not taken into account by the trajectory planner.

Since the robotic system is not mobile but is located in a fixed place with a specific interaction radius, the system operates in a relatively low dynamic environment (compared to highly dynamic environments such as autonomous driving). Therefore, only a limited number of environmental variables can have an impact on the AI component. From a domain analysis or domain expert, it might be known that sensor noise of the camera or brightness variations in the environment can lead to false predictions of the AI component.

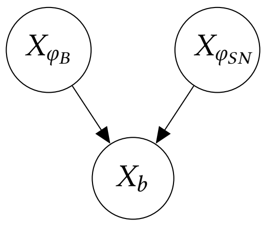


Figure 7.10.: The sensitivity model of the HRI example system.

That is, we have two uncertainties that affect the predictive uncertainty and allow conclusions to be drawn about the true state of b, i.e. φ_B describing varying brightness conditions and φ_{SN} for sensor noise. The respective sensitivity model is shown in Figure 7.10. ■

Finally, the question arises of how to obtain a sensitivity model of the form $P(X_b \mid X_{\varphi_1}, \dots, X_{\varphi_N})$. Depending on how well-researched the domain is or whether it indicates low dynamics in the environment (as in the HRI example system), a domain expert can be consulted to build the sensitivity model manually. Since we represent the model by a BN (i.e. in a graph-based structure), they are fairly understandable and amenable for humans. Alternatively, in the literature, numerous sensitivity approaches are presented. For instance, [76] provides an overview of sensitivity analysis approaches for neural networks. Additionally, the references [170, 217, 179, 102] provide a good starting point for further sensitivity analysis approaches. We would like to emphasise once again that the development of a sensitivity analysis approach is not within the scope of this work, but rather to reuse existing approaches such as those listed previously. Furthermore, there is no universal sensitivity analysis that can be applied to any type of AI model, as it is adjusted to the characteristics of the AI model. For example, an AI model from the field of natural language processing is completely different to a model for object detection. It should also be noted that the aforementioned approaches do not directly provide a sensitivity model of the form $P(X_b \mid X_{\varphi_1}, \dots, X_{\varphi_N})$, but provide the means to derive such a model.

7.1.3. Reliability Prediction of AI-Enabled Systems

In this section, we discuss our extension of the reliability prediction tool PCM-Rel for AI-enabled systems. Recall from section 2.3.2.2 that PCM-Rel considers three failure types: software, hardware and network failure types each associated with a failure probability. In this work, we introduce a fourth type of failure, namely *Uncertainty-induced Failure Type*. Uncertainty-induced failure types refine PCM-Rel-specific failure types in the sense that they are enriched with additional uncertainties that affect the probability of failure of the respective failure type. For AI-enabled systems, for example, the predictive uncertainty of an AI component can be considered a type of software failure which is dependent on a collection of uncertainties. The uncertainties influence the probability of failure of the refined software failure. This refinement or relation of a failure type to a set of uncertainties is established by an uncertainty-induced failure type. In the context of this

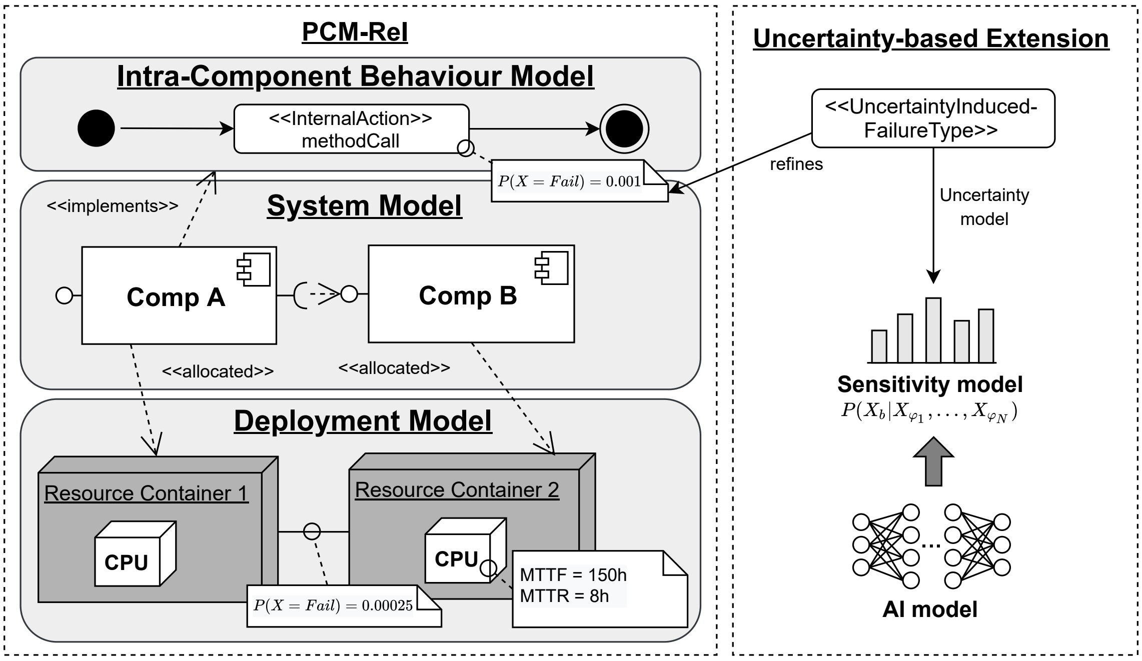


Figure 7.11.: Uncertainty-based extension of PCM-Rel.

work, the uncertainty-induced failure type relates the respective software failure type (capturing predictive uncertainty) to the sensitivity model $P(X_b \mid X_{\varphi_1}, \dots, X_{\varphi_N})$ where $\varphi_1, \dots, \varphi_N$ represents the uncertainties. This idea is illustrated on Figure 7.11.

7.1.3.1. Formal Extension of PCM-Rel

Before we present the practical extension, we start to discuss the formal extension of PCM-Rel to show formal consistency and the formal semantics of our approach.

Therefore, recall from section 2.3.2.2 that in PCM-Rel the probability of whether a software system is experiencing a successful or unsuccessful run for a given usage scenario is defined by the distribution $P(X_{Sys} \mid X_U)$. The probability can be rewritten by taking into account the resource failure patterns ψ: $\sum_{\psi \in \Psi} P(X_{Sys}, X_\Psi = \psi \mid X_U)$. By assuming stochastic independence between the usage scenario and resource failure pattern variables, one obtains (2.3).

We extend PCM-Rel by considering a set of uncertainties that influence the predictive uncertainty of an AI component. More specifically, these uncertainties refer to the set $\varphi_1, \dots, \varphi_N$. Their influence on the prediction result indirectly impacts the overall success (or failure) probability of a system. That is, for different realisations of $X_{\varphi_1}, \dots, X_{\varphi_N}$ (where X_{φ_i} is a random variable associated with property φ_i), one would observe different success and failure probabilities, i.e. $P(X_{Sys} \mid X_U)$. Hereby, the random variable X_{Sys} describes the probability of success or failure of the system including the AI component affected by

$\varphi_1, \dots, \varphi_N$. That is to say, the resulting distribution that we want to predict w.r.t. $\varphi_1, \dots, \varphi_N$ looks as follows:

$$P(X_{Sys} \mid X_U, X_{\varphi_1}, \dots, X_{\varphi_N}) \tag{7.1}$$

By taking into account the hardware failures (such as in equation (2.3)), distribution (7.1) can be rewritten:

$$\begin{aligned} P(X_{Sys} \mid X_U, X_{\varphi_1}, \dots, X_{\varphi_N}) &= \sum_{\psi \in \Psi} P(X_{Sys}, X_\Psi = \psi \mid X_U, X_{\varphi_1}, \dots, X_{\varphi_N}) \\ &= \sum_{\psi \in \Psi} P(X_{Sys} \mid X_\Psi = \psi, X_U, X_{\varphi_1}, \dots, X_{\varphi_N}) \cdot Pr(X_\Psi = \psi \mid X_{\varphi_1}, \dots, X_{\varphi_N}) \\ &= \sum_{\psi \in \Psi} P(X_{Sys} \mid X_\Psi = \psi, X_U, X_{\varphi_1}, \dots, X_{\varphi_N}) \cdot Pr(X_\Psi = \psi) \end{aligned} \tag{7.2}$$

The last step of equation (7.2) follows from the fact that we assume independence between the resource failures ψ and uncertainties $\varphi_1, \dots, \varphi_N$, i.e. $(X_{\varphi_1}, \dots, X_{\varphi_N} \perp\!\!\!\perp X_\Psi)$. Roughly speaking, we assume that the occurrence of any AI-specific uncertainty does not correlate with the observation of a specific resource failure pattern. Regarding the HRI example system, for instance, the occurrence of varying brightness conditions or sensor noise does not have any effect on the probability of observing a hardware failure.

When comparing equation (7.2) with equation (2.3) (i.e. the original equation for PCM-Rel), it can be seen that only the CPD of equation (2.3) is expanded by the uncertainties $\varphi_1, \dots, \varphi_N$. That is, from a formal perspective we must extend PCM-Rel in a way such that we can evaluate CPDs of the form $P(X_{Sys} \mid X_\Psi = \psi, X_U, X_{\varphi_1}, \dots, X_{\varphi_N})$. The details of the extension are explained in the next sections.

7.1.3.2. Metamodeling Uncertainty-induced Failures

In this section, we present the metamodel for describing uncertainty-induced failure types. Before we delve into the details of the metamodel, however, we first have to discuss how the sensitivity model $P(X_b \mid X_{\varphi_1}, \dots, X_{\varphi_N})$ is represented. To this end, we assume that the sensitivity model has already been derived, e.g. by using the methods that we presented in section 7.1.2. Now the question arises of how the model is represented on a metamodel level to make use of it. Therefore, recall from section 7.1.2 that we consider the sensitivity model as a discrete BN. In chapter 5, we presented the *EnvDyn* metamodel that allows the modelling of BNs. Thus, we use the modelling capabilities provided by *EnvDyn* to describe the sensitivity model. Moreover, in section 7.2, we discuss how to generalise the approach to self-adaptive systems, where the dynamics of the environment need to be modelled anyway. Since the uncertainties $\varphi_1, \dots, \varphi_N$ must by definition be considered part of the environment (see definition 17 on page 74), they form a subset of the entire environment (or rather of the set of environmental variables). Figure 7.12 depicts the metamodel.

The root element of the metamodel is the `UncertaintyRepository` which references a set of `UncertaintyInducedFailureTypes`, `ArchitecturalPreconditions` and `Architec-turalCountermeasures`. We start by discussing `ArchitecturalPreconditions` because the

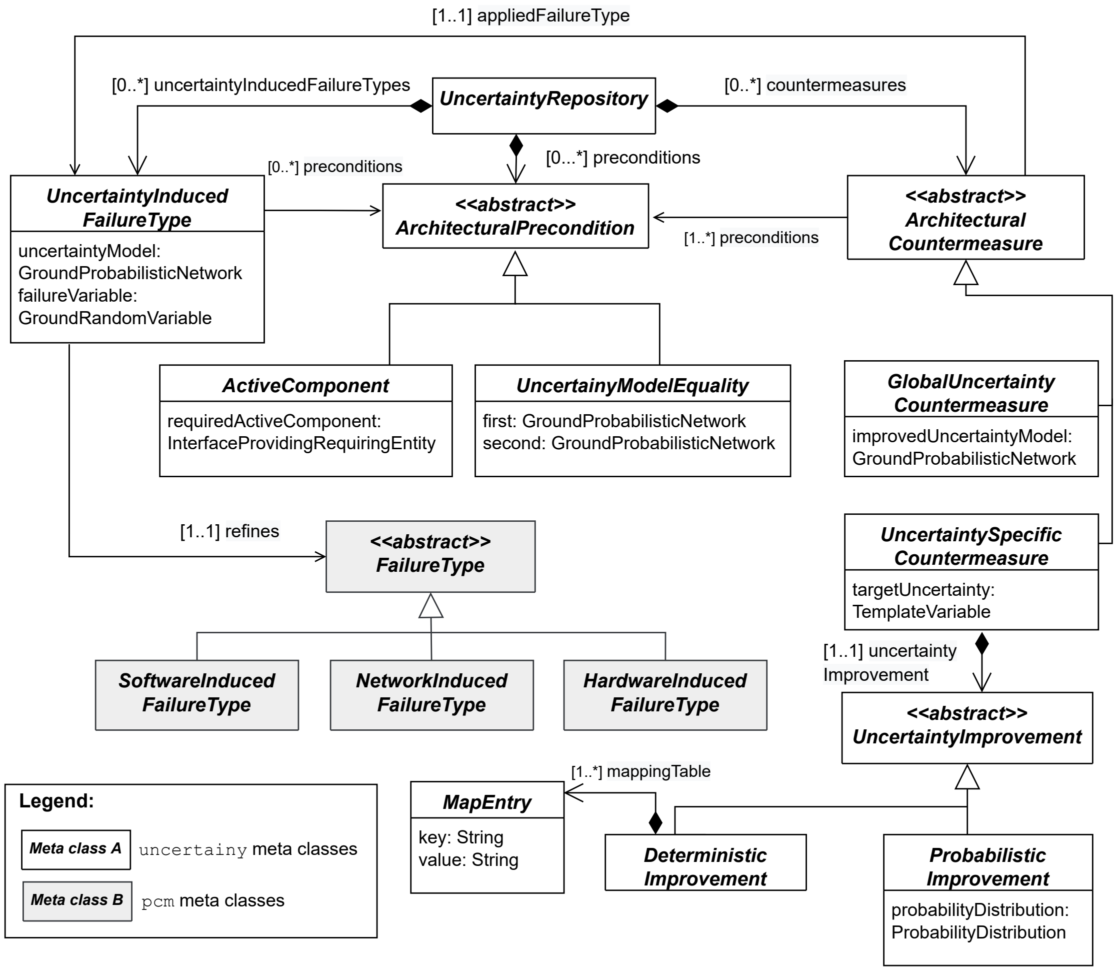

[1..1] appliedFailureType
[0..*] uncertaintyInducedFailureTypes
UncertaintyRepository
[0..*] countermeasures
[0..*] preconditions
[0...*] preconditions
[1..*] preconditions
UncertaintyInduced FailureType
uncertaintyModel: GroundProbabilisticNetwork
failureVariable: GroundRandomVariable
<<abstract>> ArchitecturalPrecondition
<<abstract>> Architectural Countermeasure
ActiveComponent
requiredActiveComponent: InterfaceProvidingRequiringEntity
UncertainyModelEquality
first: GroundProbabilisticNetwork
second: GroundProbabilisticNetwork
GlobalUncertainty Countermeasure
improvedUncertaintyModel: GroundProbabilisticNetwork
[1..1] refines
<<abstract>> FailureType
UncertaintySpecific Countermeasure
targetUncertainty: TemplateVariable
[1..1] uncertainty Improvement
SoftwareInduced FailureType
NetworkInduced FailureType
HardwareInduced FailureType
<<abstract>> UncertaintyImprovement
[1..*] mappingTable
MapEntry
key: String
value: String
Deterministic Improvement
Probabilistic Improvement
probabilityDistribution: ProbabilityDistribution
Legend:
Meta class A
uncertainy meta classes
Meta class B
pcm meta classes

Figure 7.12.: Metamodel of the uncertainty-induced failure types.

`UncertaintyInducedFailureTypes` and `ArchitecturalCountermeasures` depend on the metaclass. In simple terms, `ArchitecturalPreconditions` describe preconditions that the architecture model (or in our case the PCM model) must satisfy. For example, a precondition might require the actual use of a software component in the runtime model. Thus, for components that implement the same interface (i.e. the same service), only the component that is deployed at runtime would satisfy the precondition. The `ActiveComponent` is a sub metaclass of `ArchitecturalPreconditions` that holds the previously explained semantics. Besides `ActiveComponent`, there is `UncertaintyModelEquality` as a second `ArchitecturalPrecondition`. `UncertaintyModelEquality` requires structural equality between two uncertainty models. Suppose two uncertainty models u_1 and u_2 which we consider as BNs, i.e. for each model there is a graph $\mathcal{G}_{u_i}$ describing the structure of model u_i. For u_1 and u_2, the `UncertaintyModelEquality` precondition is satisfied if and only if $\mathcal{G}_{u_1} = \mathcal{G}_{u_2}$. Note that `UncertaintyModelEquality` only requires structural equality but no equality regarding the distributions. For instance, let $P_{u_1}, P_{u_2} \models \mathcal{G}_{u_1}$ (and thus $P_{u_1}, P_{u_2} \models \mathcal{G}_{u_2}$) be two distributions that satisfy the dependency structure of $\mathcal{G}_{u_1}$ and $\mathcal{G}_{u_2}$ (recall the $\models$ notation from section 2.6.1), although the `UncertaintyModelEquality` preconditions hold for the graph structure, it does not require that the same holds for the distributions, i.e. $P_{u_1} = P_{u_2}$ is not required. The `ActiveComponent` and `UncertaintyModelEquality` are currently the only sub-metaclasses of `ActiveComponentPrecondition`, although there might be other preconditions, e.g. the deployment of a component on a specific hardware resource. Therefore, `ArchitecturalPrecondition` is an abstract metaclass and thus extensible. So far we have only discussed the idea of `ArchitecturalPreconditions`, but have not yet given any insight into the purpose of the metaclass. Nevertheless, we deliberately continue with the metaclasses `UncertaintyInducedFailureTypes` and `ArchitecturalCountermeasures` because the purpose of `ArchitecturalPreconditions` becomes clear when we discuss the concepts in which they are used.

An `UncertaintyInducedFailureType` defines the metaclass for describing uncertainty-based failure types. It refines the super metaclass `FailureType` of the PCM-Rel metamodel referencing a single `FailureType` indicating a refinement relationship. Thus, only software, hardware or network failure can be refined by an uncertainty model. As a starting point, however, we focus on refining exclusively software failure types of AI components. Nevertheless, we discuss in section 7.1.3.4 how the concepts can be generalised. An `UncertaintyInducedFailureType` is additionally referencing a `GroundProbabilisticNetwork` which corresponds to the metaclass of the *EnvDyn* metamodel describing BNs. Note that although the reference is named `uncertaintyModel`, we refer here to the sensitivity model. To maintain generalisability, we use the term uncertainty model to be not too restrictive regarding the type of model one can choose to describe the probabilistic influence of uncertainties. Again, we defer the discussion to section 7.1.3.4. The next reference of a `UncertaintyInducedFailureType` is called `failureVariable` and represents the main variable of failure (typed as `GroundRandomVariable`), e.g. predictive uncertainty captured by X_b. The reference can be set optionally. However, if not specified, the failure variable must be identified in the uncertainty model. This cannot be achieved without making assumptions about the model, i.e. there must only be one random variable in the BN that has no descendants. The last reference refers to a set of `ArchitecturalPreconditions`. An `UncertaintyInducedFailureType` can either be active or inactive. Suppose an `UncertaintyInducedFailureType` instance refines a software failure type of an AI component with a sensitivity model. For the AI component, there might exist several implementations, e.g. different kinds of deep neural networks for object detection. Let us now assume that for each model there is an `UncertaintyInducedFailureType` (and sensitivity model, respectively), then only the `UncertaintyInducedFailureType` of the instantiated (or deployed) AI component in the system is considered active; the remaining are considered inactive. We return to this concept in section 7.1.3.3 as we can only consider `UncertaintyInducedFailureTypes` in the prediction process that are active.

The last main concept refers to `ArchitecturalCountermeasures` that describe architectural means that one can apply to deal with uncertainties. By architectural means, we mean architectural patterns describing architectural safeguards (such as those presented in section 7.1.1) that are used to cope with uncertainties and to improve the overall quality of the system. An `ArchitecturalCountermeasure` models the concrete effect of an architectural safeguard on the predictive uncertainty of an AI component. Generally, `ArchitecturalCountermeasures` reference a set of `ArchitecturalPreconditions` and a single `UncertaintyInducedFailureType`. Similarly to `UncertaintyInducedFailureType`, `ArchitecturalCountermeasures` define preconditions to check whether they are applicable (or active) in a specific context. For example, take the filtering pattern where an additional filter component is activated to preprocess the input data. To apply the countermeasure, the corresponding filter component must be instantiated in the system model. In addition to the set of `ArchitecturalPreconditions`, the `UncertaintyInducedFailureType` (for which the countermeasure is to be used) is referenced. The `ArchitecturalCountermeasures` metaclass is abstract. Currently, there are two extensions, namely `GlobalUncertainty-Countermeasure` and `UncertaintySpecificCountermeasure`. `GlobalUncertaintyCounter-measure` refer to countermeasures that have a global impact on the predictive uncertainty of an AI component. With global impact, we mean the direct impact on the probability of failure/success of the uncertainty model. In the case of AI components and sensitivity models, this refers to the distribution $P(X_b \mid X_{\varphi_1}, \dots, X_{\varphi_N})$. In principle, the effect of an architectural countermeasure always has an impact on the uncertainty model for which it is used (see reference `appliedFailureType`). Regarding `GlobalUncertaintyCountermea-sures`, this refers to the impact on the failure variable (w.r.t. uncertainties), e.g. X_b in the case of AI components. On the contrary, we consider `UncertaintySpecificCounter-measures` as countermeasures that have a local impact on a specific uncertainty (this is discussed later). More formally, let $\mathcal{G}$ be the DAG (directed acyclic graph) describing the structure of an uncertainty model and P the respective probability distribution (e.g. obtained by a sensitivity analysis), i.e. $P \models \mathcal{G}$. When a `GlobalUncertaintyCountermea-sure` is applied, one obtains a new probability distribution P' with $P' \models \mathcal{G}$; that is, a global countermeasure solely changes the distribution or parametric setting of the BN describing the uncertainty model but do not modify the structure encoded by $\mathcal{G}$. The new distribution P' is captured by the reference `improvedUncertaintyModel` of a `GlobalUncer-taintyCountermeasure`. Note, however, that both P and P' are defined over $\mathcal{G}$. Similarly, both factorise into a set of CPDs (due to the decomposability property of BNs). That is to say, a `GlobalUncertaintyCountermeasure` solely changes $P(X_b \mid X_{\varphi_1}, \dots, X_{\varphi_N})$ in the set of CPDs to which P' factorises but leaves the other CPDs (e.g. $P(X_{\varphi_i})$) unchanged.

Example 9. Recall the n-version programming pattern discussed in section 7.1.1. In the n-version programming pattern (specifically for AI use cases), N AI components are considered and queried simultaneously for a given input data. Depending on the implementation of the voter, the prediction result is determined from the N predictions made. However, the resulting predictions of the n-version pattern are very likely to differ from the predictions of the single AI component (which is to be replaced by the N versions). Consequently, the quality of the predictions changes as well; thus, we can associate the n-version programming pattern with a new sensitivity model that accounts for the changed prediction quality. For the n-version programming pattern, one can create a `GlobalUncertaintyCountermeasure` where the `improvedUncertaintyModel` points to the new sensitivity model. Additionally, an `UncertaintyModelEquality` precondition must be created to make sure that the new sensitivity model is applicable or structurally equal to the uncertainty model in which distribution P is to be adjusted (or improved). ■

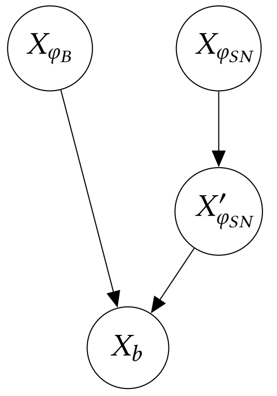


Figure 7.13.: The sensitivity model of the HRI example system after applying filtering pattern as an architectural countermeasure.

In contrast, `UncertaintySpecificCountermeasure` describe countermeasures that have a local effect, i.e. they influence a specific uncertainty. Therefore, an `UncertaintySpecificCountermeasure` references a `TemplateVariable` corresponding to the target uncertainty addressed by the countermeasure. The `TemplateVariable` must be instantiated and represents an uncertainty φ_i of the uncertainty model. The main goal of `UncertaintySpecificCountermeasures` is to contain the target uncertainty. For example, each uncertainty φ_i follows a probability distribution, i.e. $P(X_{\varphi_i})$. An `UncertaintySpecificCountermeasure` cannot change the distribution itself, as any uncertainty is part of the environment, which cannot be controlled (or only to a certain extent). However, it affects whether uncertainties are forwarded to the AI component in exactly the form in which they are observed in the environment.

Example 10. Recall the HRI example system from section 1.5.3 and the corresponding sensitivity model from Figure 7.10. An example of an `UncertaintySpecificCountermeasure` represents the filtering pattern where a filter component is used to preprocess incoming data. In the case of the HRI system, a filter might be employed to filter out noise artefacts of the input image. Thus, the sensor noise uncertainty is contained. The filter component does not directly affect the success/failure probability of the AI component (e.g. as in the n-version programming approaches) but reduces the likelihood of potentially malicious input data being passed to it. Considering the sensitivity model (or uncertainty model), the effect of the filtering pattern manifests itself by inserting an additional random variable $X'_{\varphi_{SN}}$ between X_b and $X_{\varphi_{SN}}$ (see Figure 7.13).

The random variable $X'_{\varphi_{SN}}$ specifies the effect of the filtering process on the sensor noise. More specifically, $X'_{\varphi_{SN}}$ models the probability of eliminating noise artefacts from the input image originally observed in the environment before passing it on to the AI component. ■

More formally, let $\mathcal{G}$ be the DAG describing the structure of an uncertainty model. When an `UncertaintySpecificCountermeasure` is applied, one obtains a new probabilistic structure $\mathcal{G}'$ of the uncertainty model (such as depicted on Figure 7.13); that is, an uncertainty-specific countermeasure changes the original structure of the sensitivity model by inserting a new random variable that describes the impact of the target uncertainty. Therefore, we say that an `UncertaintySpecificCountermeasure` modifies the structure of an uncertainty model but not the parametric setting (opposed to `GlobalUncertaintyCountermeasure`). One may argue that if the structure $\mathcal{G}$ is changed to $\mathcal{G}'$, there must be also a new distribution (and thus a parametric change) satisfying $\mathcal{G}'$, i.e. $P \models \mathcal{G} \wedge P' \models \mathcal{G}' \Rightarrow P \not\models \mathcal{G}'$. However, $\mathcal{G}'$ preserves the (in-)dependency assumptions of $\mathcal{G}$ such that the original parametric setting is preserved as well. More accurately, we can easily obtain P from $\mathcal{G}'$ by marginalising over the additional random variables introduced by the countermeasure: $P(X_b, X_{\varphi_1}, \dots, X_{\varphi_N}) = \sum_{\varphi \in Val(\varphi_i)} P'(X_b, X_{\varphi_1}, \dots, X_{\varphi_N}, X'_{\varphi_i} = \varphi)$. The concrete improvement or impact on the target uncertainty φ_i is modelled by an `UncertaintyImprovement`. Basically, an improvement is defined by a function $f_\varphi : Val(X_\varphi) \rightarrow Val(X_\varphi)$. An `UncertaintyImprovement` represents an abstract metaclass which is extended by two sub metaclasses, namely `DeterministicUncertaintyImprovement` and `ProbabilisticUncertaintyImprovement`. A `DeterministicUncertaintyImprovement` describes a deterministic effect of the improvement on φ_i. In this case, the function f_{φ_i} is bijective and uniquely defines a mapping from $Val(X_\varphi)$ to $Val(X_\varphi)$; this is modelled by key-value pairs (see reference `mappingTable` in Figure 7.12). A `ProbabilisticUncertaintyImprovement` describes a probabilistic effect of the improvement on φ_i. For example, in the HRI example, there might be input images where the noise component is too strong such that even the application of a filter cannot reduce the noise level. The improvement would be only effective for a portion of images which can be described probabilistically. In this case, function f_{φ_i} is modelled as probability distribution, i.e. $f_{\varphi_i} = P(X_{\varphi_i} \mid X_{\varphi_i})$. The distribution is modelled by using the *ProbDist* metamodel from section 5.2.6.

7.1.3.3. Uncertainty-based Reliability Prediction

In this section, we unify the presented concepts together with the reliability prediction approach PCM-Rel to make reliability predictions of AI-enabled software systems. Therefore, recall the main components of the approach (as depicted on Figure 7.1), namely ATs, sensitivity analysis and the prediction tool PCM-Rel. As a prerequisite of the extended reliability prediction, an upstream sensitivity analysis must be conducted such that the sensitivity model $P(X_b \mid X_{\varphi_1}, \dots, X_{\varphi_N})$ is obtained. In addition, the sensitivity model needs to be represented as BN by using the modelling capabilities of the *EnvDyn* metamodel. The sensitivity model, the architecture or PCM model and a model describing the uncertainty-induced failure types from the last section (which we commonly refer to in the following as the uncertainty-refined failure model) are the main components relevant to the prediction approach.

Internally, the prediction process can be viewed as schematically shown in the activity diagram of Figure 7.14. As depicted in the diagram, the prediction process starts by checking whether an AT needs to be applied first before the regular prediction process is carried out. In the following, however, we begin to present the regular prediction process and discuss the application of ATs afterwards. In this case, the process continues

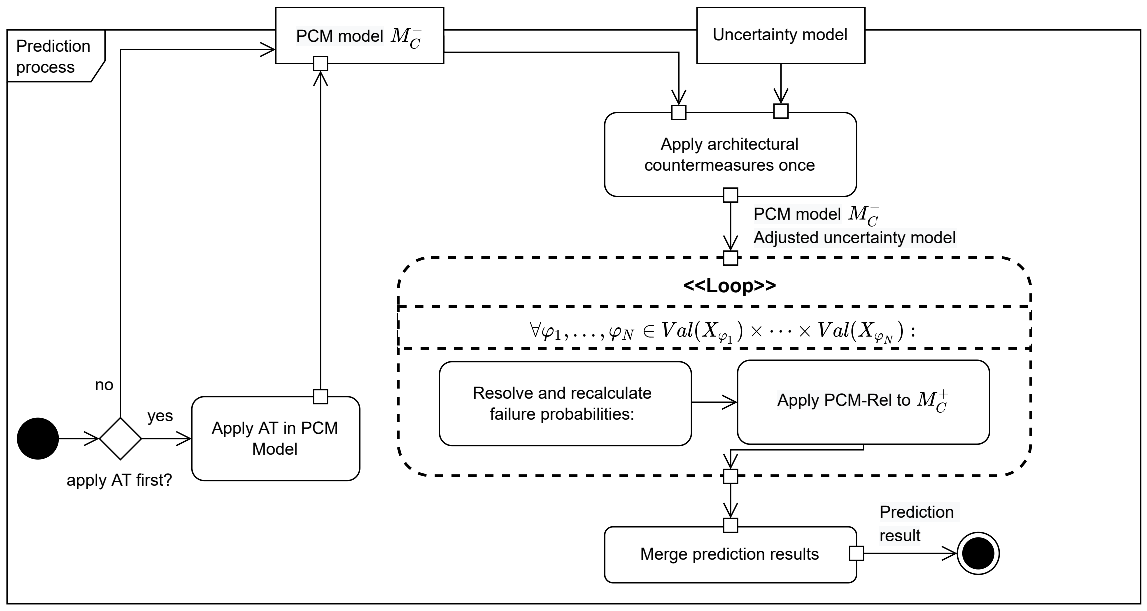
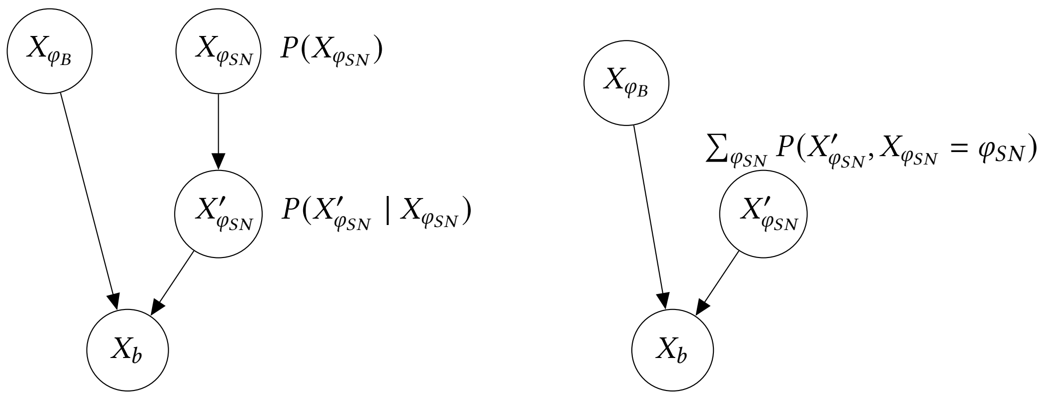

Prediction process
PCM model M_C^-
Uncertainty model
Apply architectural countermeasures once
PCM model M_C^-
Adjusted uncertainty model
<<Loop>>
$\forall \varphi_1, \ldots, \varphi_N \in Val(X_{\varphi_1}) \times \cdots \times Val(X_{\varphi_N})$:
Resolve and recalculate failure probabilities:
Apply PCM-Rel to M_C^+
no
yes
Apply AT in PCM Model
apply AT first?
Prediction result
Merge prediction results

Figure 7.14.: Activity diagram of the prediction process.

by applying architectural countermeasures w.r.t. the PCM model and the uncertainty-refined failure model. Afterwards, the PCM model and the uncertainty-refined failure model are forwarded to a loop which iterates over all uncertainty tuples $\varphi_1, \ldots, \varphi_N$ of the uncertainty space, i.e. the space spanned by the distinct value spaces of each uncertainty φ_i. In each iteration, two steps are performed w.r.t. each uncertainty tuple. In the following, we assume for simplicity that only a single `UncertaintyInducedFailureType` (capturing the AI-induced uncertainties) is defined. We discuss in section 7.1.3.4 how the concepts generalise.

As a first step, architectural countermeasures are applied. In our case, the uncertainty model refers to the sensitivity model which represents the predictive uncertainty of an AI component in the presence of second-order uncertainties. Moreover, the uncertainty model is referenced by an `UncertaintyInducedFailureType`-object of the uncertainty-refined failure model that refines the corresponding `SoftwareInducedFailureType` describing the predictive uncertainty of an AI component. The uncertainty model is the primary source used to calculate the adjusted failure probabilities of each refined failure type (as we will see in the first step of the loop). Since countermeasures act on the uncertainty model, their effects manifest themselves exclusively in the uncertainty model. Therefore, the uncertainty model (or sensitivity model in our case) is adjusted w.r.t. the used countermeasures. However, before a countermeasure is applied, it must be checked if the countermeasure is active, i.e. whether its `ArchitecturalPreconditions` are satisfied and whether the referenced failure type is active as well. Recall that architectural countermeasures act on the uncertainty model either locally (i.e. on a specific uncertainty) or globally (i.e. by changing the overall probability of success and failure). Moreover, the modification is either structural (i.e. related to the structure of the graph) or parametric (i.e. related to the probability distribution of the graph), depending on the kind of countermeasure


X_{φ_B}
$X_{\varphi_{SN}}$
$P(X_{\varphi_{SN}})$
$X'_{\varphi_{SN}}$
$P(X'_{\varphi_{SN}} \mid X_{\varphi_{SN}})$
X_b
X_{φ_B}
$\sum_{\varphi_{SN}} P(X'_{\varphi_{SN}}, X_{\varphi_{SN}} = \varphi_{SN})$
$X'_{\varphi_{SN}}$
X_b

(a) Uncertainty model before pruning. **(b)** Uncertainty model after pruning.

Figure 7.15.: Pruning process: (a) represents the graph $\mathcal{G}$ before pruning and (b) depicts the graph after pruning.

(i.e. `UncertaintySpecificCountermeasure` and `GlobalUncertaintyCountermeasure`). In the case of a `GlobalUncertaintyCountermeasure`, the modification is trivial because, due to the decomposability property of BNs, only the probability $P(X_b \mid X_{\varphi_1}, \ldots, X_{\varphi_N})$ of the old uncertainty model needs to be swapped with the new distribution. In terms of `UncertaintySpecificCountermeasure`, the modification is more complicated due to the structural change of the model. Therefore, consider Figure 7.13 where we illustrated the effect of a structural change caused by an `UncertaintySpecificCountermeasure`. Recall from the activity diagram that it is iterated over all uncertainties. If we stay with the example of Figure 7.13, this involves all combinations of $Val(X_{\varphi_B}) \times Val(X_{\varphi_{SN}})$ such that the probability of failure $P(X_b \mid X_{\varphi_B}, X_{\varphi_{SN}})$ for all combinations is evaluated. However, if the structure of the graph changes and another random variable $X'_{\varphi_{SN}}$ (representing the effect of an `UncertaintySpecificCountermeasure`) is inserted, iterating over all uncertainty combinations is no longer sufficient in some cases. More specifically, this depends on the type of `UncertaintyImprovement` associated with the countermeasure. For instance, let us assume that the improvement is deterministic. In this case, no further steps must be taken because also the probability distribution related to $X'_{\varphi_{SN}}$ becomes deterministic, i.e. $P(X'_{\varphi_{SN}} \mid X_{\varphi_{SN}}) = \mathbb{1}_{f(\varphi_{SN})}$. When the improvement is probabilistic, however, for each iteration of the loop the possible outcomes of distribution $P(X'_{\varphi_{SN}} \mid X_{\varphi_{SN}})$ defined over $X'_{\varphi_{SN}}$ must be considered as well. That is, iterating over all uncertainty combinations is not sufficient anymore. To address this problem, one may expand the number of iterations by considering also the different values of $Val(X'_{\varphi_{SN}})$, i.e. $Val(X_{\varphi_B}) \times Val(X_{\varphi_{SN}}) \times Val(X'_{\varphi_{SN}})$. However, the number of iterations already increase exponentially in the number of uncertainties (we discuss the state space explosion problem at the end of this section). Therefore, we prune the uncertainty model structure by merging $X_{\varphi_{SN}}$ and $X'_{\varphi_{SN}}$ to a single random variable as depicted on Figure 7.15.

The idea of the pruning or merging process is to recover the structure of the original graph before the `UncertaintySpecificCountermeasure` has been applied such that we can iterate over all the uncertainties as before. This is achieved by removing variable $X_{\varphi_{SN}}$ from the graph and by adjusting the probability distribution defined over $X'_{\varphi_{SN}}$, i.e. $P(X'_{\varphi_{SN}})$. Based on the originally provided distributions $P(X_{\varphi_{SN}})$ and $P(X'_{\varphi_{SN}} \mid X_{\varphi_{SN}})$, the new distribution $P(X'_{\varphi_{SN}})$ is constructed as follows:

$$\begin{aligned} P(X'_{\varphi_{SN}}) &= \sum_{\varphi_{SN} \in Val(X_{\varphi_{SN}})} P(X'_{\varphi_{SN}}, X_{\varphi_{SN}} = \varphi_{SN}) \\ &= \sum_{\varphi_{SN} \in Val(X_{\varphi_{SN}})} P(X'_{\varphi_{SN}} \mid X_{\varphi_{SN}} = \varphi_{SN}) \cdot Pr(X_{\varphi_{SN}} = \varphi_{SN}) \end{aligned} \tag{7.3}$$

Note that although the mathematical derivation is based on the uncertainty model of the HRI example system, the concepts generalise to any uncertainty φ_i. Moreover, equation (7.3) reflects the intention of an `UncertaintySpecificCountermeasure`, namely the containment or mitigation of an uncertainty. Based on the pruned graph, the prediction process can be continued as usual.

In the next step of the prediction process, it is iterated over all uncertainty tuples (as described before). The first step within the loop corresponds to the resolution and recalculation of the failure probabilities. Recall that the core idea of the prediction process is to recalculate the failure probability of the failure type associated with an AI component. More precisely, this failure probability is determined taking into account the distinct uncertainty permutations. That is, the probability of failure of the AI component is calculated w.r.t. the considered uncertainty tuple $\varphi_1, \ldots, \varphi_N$, i.e. $Pr(X_b = Fail \mid X_{\varphi_1} = \varphi_1, \ldots, X_{\varphi_N} = \varphi_N)$. Afterwards, the recalculated failure probability is resolved in the (unresolved) PCM model M_C^- by retrieving the refined failure type and updating the failure probability. Finally, the resolved PCM model M_C^+ contains the updated failure probability and can be further analysed.

More specifically, the following step applies the conventional prediction procedure of PCM-Rel to the resolved model M_C^+ to predict the probability of success as usual. As a result, we obtain the predicted success (and failure) probability of the system, given the uncertainty model (or sensitivity model) of the AI component and a tuple of uncertainties, i.e. $P(X_{Sys} \mid X_U, X_{\varphi_1}, \ldots, X_{\varphi_N})$ from (7.1). This is repeated for all uncertainty tuples such that one obtains a set of prediction results for each uncertainty tuple.

Finally, after the loop is completely iterated, the number of prediction results must be merged. The individual success and failure predictions help to understand the impact of each uncertainty on the AI component and the overall system. In the end, however, the major result is to obtain the overall success probability $P(X_{Sys} \mid X_U)$ given a certain usage scenario U (just as originally provided by PCM-Rel). The overall success probability takes into account all conditional success probabilities and summarises the impact of each uncertainty on the system (or AI component) into a single value. Formally, $P(X_{Sys} \mid X_U)$ is computed by adding the individual conditional probabilities of success weighted by the probability of observing an uncertainty tuple to which it is conditional:

$$\begin{aligned}
P(X_{Sys} \mid X_U) &= \sum_{\psi \in \Psi} \sum_{\varphi_1,\ldots,\varphi_N \in \Phi} P(X_{Sys}, X_\Psi = \psi, X_\Phi = \varphi_1, \ldots, \varphi_N \mid X_U) \\
&= \sum_{\psi \in \Psi} \sum_{\varphi_1,\ldots,\varphi_N \in \Phi} Pr(X_\Phi = \varphi_1, \ldots, \varphi_N) \cdot Pr(X_\Psi = \psi \mid X_\Phi = \varphi_1, \ldots, \varphi_N) \\
&\qquad \cdot P(X_{Sys} \mid X_\Psi = \psi, X_\Phi = \varphi_1, \ldots, \varphi_N, X_U) \\
&= \sum_{\varphi_1,\ldots,\varphi_N \in \Phi} Pr(X_\Phi = \varphi_1, \ldots, \varphi_N) \\
&\qquad \underbrace{\cdot \sum_{\psi \in \Psi} Pr(X_\Psi = \psi) \cdot P(X_{Sys} \mid X_\Psi = \psi, X_\Phi = \varphi_1, \ldots, \varphi_N, X_U)}_{(7.2)} \\
&= \sum_{\varphi_1,\ldots,\varphi_N \in \Phi} Pr(X_\Phi = \varphi_1, \ldots, \varphi_N) \cdot P(X_{Sys} \mid X_U, X_\Phi = \varphi_1, \ldots, \varphi_N)
\end{aligned} \tag{7.4}$$

For the sake of clarification, we use the notations Φ to represent $Val(X_{\varphi_1}) \times \cdots \times Val(X_{\varphi_N})$ and $X_\Phi = \varphi_1, \ldots, \varphi_N$ to represent $X_{\varphi_1} = \varphi_1, \ldots, X_{\varphi_N} = \varphi_N$.

The prediction result encompasses a set of CPDs capturing the conditional success and failure probabilities of the system and the overall probability of success and failure. Note that one could have used the uncertainty model to marginalise over the random variables of the uncertainties (i.e. $X_{\varphi_1}, \ldots, X_{\varphi_N}$) to compute the probability $Pr(X_b = Fail)$ (or $Pr(X_b = Success)$). Afterwards, the probability could have been used to describe the probability of failure of the respective failure type in the PCM model and evaluated by using PCM-Rel as usual. From the perspective of a software engineer, however, the individual CPDs provide more insights regarding the impact of certain uncertainties and can be used to prepare appropriate countermeasures. In addition, the approach presented is fully automated and requires no further manual intervention.

Based on the prediction result, software architects might test several architectural countermeasures to deal with uncertainties and enhance the overall reliability of the system. Architectural countermeasures can be defined within the uncertainty-refined failure model. In this work, we use ATs to represent architectural patterns that can be used to deal with uncertainties. We already discussed in section 7.1.1 the filtering and n-version programming pattern which we described by ATs. Hereby, we described how the ATs can be applied to the PCM model by executing the corresponding completion (i.e. model transformation). However, we have not yet discussed how the completions can be extended to enrich the uncertainty-refined failure model with `ArchitecturalCountermeasure` instances to account for the ATs in the reliability prediction for AI-enabled systems. We extended the completion of the filtering pattern creating an `UncertaintySpecificCountermeasure` capturing the effect of the filter component on given target uncertainty. Recall the EMF profile of the filtering pattern which defines a `distributionName` and `targetUncertaintyName` attribute. Both attributes are used to resolve probability distribution describing the `UncertaintyImprovement` and `TemplateVariable` capturing the target uncertainty of the filter. Based on this information, the completion complements the uncertainty-refined failure model by a corresponding countermeasure reflecting the action of the filtering pattern on the uncertainty model. The completion of the n-version programming AT has been extended in the same way. In contrast to the filtering pattern, a `GlobalUncertainty-Countermeasure` is generated because the n-version programming pattern acts globally and not on a specific target uncertainty. The EMF profile of the n-version programming AT defines the name of the improved uncertainty model (see attribute `improvedModelName`). After resolving the improved uncertainty model, the corresponding countermeasure is created and added to the uncertainty-refined failure model. Regarding the reliability prediction process, the application of ATs, if required, is performed before the actual prediction process (as can be seen in the activity diagram of Figure 7.14). Thus, the AT is considered in the PCM model and the respective countermeasure is considered during the prediction process. This empowers software engineers to select from an AT catalogue different patterns which can be checked regarding their reliability impact. Moreover, since the completions are purely PCM-based, further analysis or simulation tools of Palladio are applicable to predict other quality attributes. For instance, the application of the n-version programming pattern may improve reliability but degrade performance. However, the prediction tools support software engineers in making a suitable trade-off decision.

Finally, regarding the efficiency of the presented reliability prediction process, we already noted the exponential complexity of the procedure (the iterations of the loop depicted on Figure 7.14 grows exponentially in the number of uncertainties). Nonetheless, there are several ways to deal with this problem. As mentioned in section 7.1.2, we consider the value space of a property $Val(X_{\varphi_i})$ as a discrete set. Although discretisation drastically reduces the size of the space, it does not eliminate the exponential property, but rather makes it manageable. The degree of discretisation or resolution of each property can be controlled, e.g. by the software architect or domain expert who identifies and models the properties. However, a low resolution is associated with information loss; that is, a high degree of discretisation can potentially affect the quality of the analysis or simulation process. Therefore, the resolution must be balanced against the resulting complexity of the uncertainty space (i.e. the efficiency of the prediction procedure itself) and the quality of the prediction procedure. One last method to tackle the state space explosion problem is to apply Monte Carlo methods, i.e. by sampling property tuples from $P(X_{\varphi_1}, \dots, X_{\varphi_N})$. Although the entire space is not examined, at least the tuples of uncertainties that have high probabilities are taken into account and provide sufficient insights regarding the effect of an architectural safeguard.

7.1.3.4. Generalisation

In the last section, we presented our reliability prediction approach for AI-enabled software systems. We introduced the uncertainty-refined failure model for refining specific failure types of the PCM model by an uncertainty model. Hereby, we restricted our discussion merely to software-induced failure types as the primary means to model AI-induced failure potentials. However, the concepts explained can be generalised to a much wider range of use cases. In this section, we discuss how individual concepts of the approach need to be extended and what challenges need to be considered to apply our prediction approach for any failure type refinement and to analyse PCM models where more than one failure type is refined.

Therefore, recall the metamodel of the uncertainty-refined failure model from section 7.1.3.2. The possible refinements can be defined for any kind of failure type (and not solely software-induced failure types). In this case, the uncertainty-refined failure model includes a set of `UncertaintyInducedFailureTypes` where each refines a certain failure type in the PCM model associated with a respective uncertainty model. To discuss how the reliability prediction process can be generalised, the four main steps from the activity diagram in Figure 7.14 that make up the prediction process are discussed.

Therefore, recall that the procedure starts with applying architectural countermeasures. Regarding a generalised prediction procedure, no further actions must be taken since for each uncertainty-refined failure type the corresponding uncertainty model can be adjusted w.r.t. the modelled and applicable countermeasures.

The prediction process continues with iterating over all uncertainty tuples and which requires no further adjustments. However, as already discussed in the previous section, the complexity of the procedure is exponential in the number of considered uncertainties. Thus, when considering an arbitrary number of uncertainty models or failure refinements, the uncertainty space increases faster. We already discussed how one can tackle the state space explosion problem. However, depending on the number of uncertainty models and considered uncertainties in each model, it is likely that they cause severe computational problems. One way to deal with this problem could be to iterate sequentially over the subspaces spanned by each uncertainty model instead of iterating over the entire uncertainty space induced by all uncertainties of each uncertainty model. In this case, when iterating over the subspace of an uncertainty model, a default or fixed failure probability can be assumed for the other refined failure types, e.g. $\sum_{\varphi_1,\ldots,\varphi_N \in \Phi'} Pr(X_b = Fail, X_{\Phi'} = \varphi_1, \ldots, \varphi_N)$ for subspace $\Phi' \subset \Phi$.

In the resolving step, we assumed stochastic independence between the resource failure patterns and the uncertainties, i.e. $(X_{\varphi_1}, \ldots, X_{\varphi_N} \perp\!\!\!\perp X_\Psi)$. However, the assumption does not hold when we model a set of uncertainties $\varphi_1, \ldots, \varphi_N$ describing the effect on a hardware-induced failure type. This can be addressed by making no assumptions at all but involves the modification of PCM-Rel. Internally, PCM-Rel implements a dedicated process to calculate $Pr(X_\Psi = \psi)$ for any resource pattern ψ. Without the independence assumption $(X_{\varphi_1}, \ldots, X_{\varphi_N} \perp\!\!\!\perp X_\Psi)$, this process must be adjusted to account for distributions of the form $P(X_\Psi \mid X_{\varphi_1}, \ldots, X_{\varphi_N})$.

Finally, after applying PCM-Rel as usual the results of the prediction need to be merged. Both steps do not involve any additional effort and can be used as currently implemented.

In summary, to generalise the presented approach of the reliability process for AI-enabled systems to a wider range of uncertainty-based reliability prediction scenarios, two factors need to be considered. First, the assumptions made must be reconsidered, as they do not apply to specific situations. Secondly, the problem of state-space explosion becomes more challenging the more uncertainty-refined failure types are added.

7.2. Engineering Self-Adaptive Systems to Safeguard AI Components

In section 7.1, we discussed how to evaluate AI-enabled software systems (and non-adaptive architectural safeguards) regarding reliability attributes. However, so far we focused on static systems. In this section, we generalise the concepts to self-adaptive systems.

Although we have discussed how to deal with the black-box nature of AI components (e.g. by applying sensitivity analysis), the concepts tend to support software architects in reliability assessment. For self-adaptive systems, however, the problem of the hidden state of an AI component remains. We consider self-adaptive systems as mechanisms for safeguarding AI components. Dealing with the black-box property is consequently of great significance because self-adaptive systems must draw conclusions about the true state to plan appropriate adaptations. In the following, we start to re-formalise the problem statement of self-adaptive systems as originally stated in section 4.4.2. In particular, we relax the basic assumptions associated with MDPs and take into account the black-box property of AI components.

7.2.1. Problem Statement

Recall from section 4.4.2 that we consider a self-adaptive system as an MDP $\lambda_{SAS} := (\mathcal{S}, \Delta, t_{\mathcal{S}}, r_{\mathcal{S}})$ where the engineering problem is constituted by implementing an adaptation strategy π such that the quality objectives (reflected by reward function $r_{\mathcal{S}}$) are satisfied. Also, recall that we defined a state as a tuple $S := (E, C) \in \mathcal{S}$ consisting of an environmental state $E \in \mathcal{E}$ and an architectural configuration $C \in \mathcal{C}$ of the system. In MDPs, it is assumed that each state $S := (E, C)$ is fully observable. When considering self-adaptive systems as architectural safeguards for AI black-box components, however, this is not entirely true for the environmental state E. In the following, we discuss why this assumption no longer applies.

Recall definition 30 on page 162 of an AI black-box component defined as a function b with input space $\mathcal{X}$ and output space $\mathcal{Y}$. Since the true state of b is not observable, the uncertainty induced by b refers to the inability to determine whether a prediction or output is correct or not. Regarding the state definition $S := (E, C)$, this means that there is a variable $e_b \in E$ (recall the environmental state from definition 17 on page 74) capturing the uncertainty induced by b. According to the hidden state problem, however, the variable e_b is not observable such that the entire state $S := (E, C)$ is not fully observable. We say that S is *Partially Observable* to account for the variables in the state which remain hidden. Instead of directly monitoring the state of b (or e_b), we encounter observations of input/output pairs (i.e. of the form $(x, y) \in \mathcal{X} \times \mathcal{Y}$) that one can use to draw conclusions about the state of b. To account for these observations, we expand the problem statement by considering a self-adaptive system as a *Partially Observable Markov Decision Process* (POMDP) represented by the tuple (λ, Ω, o).

Technically, POMDPs extend MDPs λ by a set of observations Ω and an observation model $o : S \times \Omega \rightarrow [0, 1]$ that evaluates the probability to observe $\omega \in \Omega$ in state $s \in S$. That is, instead of observing a state s directly, one encounters an observation ω generated from the observation model o that serves as the basis for determining s; or, in other words, the observation process (w.r.t. o) is a stochastic process that allows one to draw conclusions about the stochastic process of the hidden states $s \in S$.

We consider self-adaptive systems safeguarding AI black-box components as POMDPs described by the tuple $(\lambda_{SAS}, \Omega, o_{\mathcal{S}})$ where $\Omega \subseteq \mathcal{X} \times \mathcal{Y}$ and $o_{\mathcal{S}} : \mathcal{S} \times \Omega \rightarrow [0, 1]$ represents the instantiated version of the original observation model (i.e. $o : S \times \Omega \rightarrow [0, 1]$). Note that in literature POMDPs are often defined differently; more precisely, the definitions additionally include the taken action in the observation model, i.e. $o : S \times A \times \Omega \rightarrow [0, 1]$. However, when the action is directly reflected or included as a state feature, then the observation model can be written in the form $o : S \times \Omega \rightarrow [0, 1]$ [174]. In our case, an adaptation is included by a state $S := (E, C)$ in the sense that its effect is directly reflected by the architectural configuration C. Additionally, from a probabilistic perspective, one may argue that the adaptation and observation process is conditionally independent given the architectural configuration because only the configuration impacts the way of observing correct or incorrect tuples, i.e. (x, y) and (x, y') where y corresponds to a wrong output and y' to a correct output (because (x, y') might be predicted in a situation where the architectural configuration included additional filter operation or pursued an n-version programming approach).

Extending the formal definition of self-adaptive systems to POMDPs is indicating how the complexity of the problem increases when uncertain AI components are involved. Instead of implementing an adaptation strategy π that must maintain the system's quality objectives over time (as originally motivated), a software architect must additionally deal with the fact that the true state is hidden.

7.2.2. Decoupling of the Observation Process

In the last section, we formally discussed the problem associated with self-adaptive systems that are supposed to manage AI black-box components, namely the hidden state of b and the observation model o as a basis to draw conclusions about the state of b. In this section, we discuss how to tackle the problems induced by uncertainties of AI.

To this end, we need to take a closer look at the problem MAPE-K-based self-adaptive systems have to deal with when safeguarding AI components, namely the *Observation Process*. In this context, the observation process mainly refers to the process of making observations (w.r.t. observation model o) from which the true state must be determined. Let us therefore briefly enumerate the steps of the decision procedure of a self-adaptive system, if one strictly adheres to the semantics of POMDPs. Since we are focused on

MAPE-K-based self-adaptive systems, these steps refer to the MAPE phases. Recall that POMDPs comprise two stochastic processes, one describing the state evolution of the hidden (or partially observable) state and another generating the respective observation of each state. That is, in the monitor phase, the self-adaptive system obtains an observation or input/output pair $\omega := (x, y) \in \Omega$. In the analyse phase, it must be determined whether the state of b is potentially erroneous, i.e. the prediction of output y is not correct. We abstract this situation by considering it as a probabilistic problem: $P(X_{\mathcal{S}} \mid X_\Omega = \omega)$. Note that w.l.o.g., we can formulate this in a probabilistic way because even if there exists a deterministic function $f(\omega)$ that determines the state, we can write this probabilistically, i.e. $Pr(X_{\mathcal{S}} = S \mid X_\Omega = \omega) = \mathbb{1}_{f(\omega)=S}$. Furthermore, recall that a state $S \in \mathcal{S}$ is considered to be partially observable, i.e. merely the variable $e_b \in E$ associated with the AI component is hidden. Thus, we can constrain the probabilistic problem to the environmental variable directly related to b: $P(X_b \mid X_\Omega = \omega)$ where (as before) X_b describes a binary random variable capturing the predictive uncertainty of an AI component. Simply put, in the analysis phase, the distribution $P(X_b \mid X_\Omega = \omega)$ must be considered to infer the true state of the AI component and decide whether an adaptation should be planned. The planning and execution phases are then carried out in the usual way.

It can be seen that, in the analyse phase, it is crucial to have profound knowledge of the distribution $P(X_b \mid X_\Omega = \omega)$ because it determines whether the system is adapted or not. However, estimating $P(X_b \mid X_\Omega = \omega)$ requires deep knowledge of the observation model o and is either tedious or even infeasible due to the complexity of the input space (e.g. the pixel space). Therefore, using $P(X_b \mid X_\Omega = \omega)$ as a criterion for deciding whether to adapt the system is arguably questionable, as it involves considerable theoretical problems.

Instead, we have to focus on a different set of properties that allow conclusions to be drawn of the true state of b. We argue to augment the monitor phase of a self-adaptive system by considering observable properties that allow conclusions to be drawn of the true state of b. Therefore, recall the set of uncertainties $\varphi_1, \ldots, \varphi_N$ (which we now consider as observable properties of b) from the sensitivity model in section 7.1.2. Depending on the domain, the properties manifest themselves in various ways; they range from simple properties such as brightness conditions or sensor noise to more complex properties such as robustness indicators or neuron coverage in deep neural networks (we discuss the nature of properties in more detail later). Nonetheless, we assume that the properties are observable, i.e. either directly from the environment (e.g. sensor noise) or derivable from the input/output of an AI component (e.g. neuron coverage). Thus, runtime monitors can be constructed such that the properties $\varphi_1, \ldots, \varphi_N$ are monitored and taken into consideration for the remaining phases of a self-adaptive system. To account for the monitored properties $\varphi_1, \ldots, \varphi_N$, the distribution $P(X_b \mid X_\Omega = \omega)$ of the analyse phase expands as follows:

$$P(X_b \mid X_\Omega = \omega, X_{\varphi_1} = \varphi_1, \ldots, X_{\varphi_N} = \varphi_N) \tag{7.5}$$

However, the distribution still includes the random variable X_Ω which is associated with severe theoretical problems. Therefore, recall the environmental state E of a self-adaptive system before augmenting the monitor phase with additional properties. The environmental state merely encompasses a single variable, namely the variable describing the state of an AI component: $E := (e_b)$. By considering the set of properties $\varphi_1, \ldots, \varphi_N$ as factors that potentially influence the quality of a prediction made by b, they indirectly impact the quality attributes of the software system including b. Recall that we consider predictive uncertainty as the first-order uncertainty of AI components. The set $\varphi_1, \ldots, \varphi_N$ refer now to what we defined at the beginning of the chapter as AI-related uncertainties of second order. Therefore, they must (according to definition 17 on page 74) also be considered in the environmental state: $E := (e_b, e_{\varphi_1}, \ldots, e_{\varphi_N})$. Given that expanded state definition, the environmental state can be partitioned into a set of observable variables $E_{obs} := \{e_{\varphi_1}, \ldots, e_{\varphi_N}\}$ and a set of hidden variables $E_{hid} := \{e_b\}$. This can also be written in terms of random variables, i.e. $X_{E_{obs}} := \{X_{\varphi_1}, \ldots, X_{\varphi_N}\}$ and $X_{E_{hid}} := \{X_b\}$ (we omit the e's to simplify the notation). Given the random variables $X_{E_{obs}}$, $X_{E_{hid}}$ and $X_{E_{obs}}$, we assume that the *Missing At Random* (MAR) assumption applies:

$$(X_\Omega \perp\!\!\!\perp X_{E_{hid}} \mid X_{E_{obs}}) \tag{7.6}$$

The MAR assumption requires conditional independence between the observations and hidden variables given the observable variables, i.e. independence between X_Ω and $X_{E_{hid}}$ given $X_{E_{obs}}$ [105, P.854]. That is, an observation $\omega := (x, y)$ provides no additional information about the hidden variables if the observed variables are known. Consequently, the following equation applies: $P(X_{E_{hid}} \mid X_{E_{obs}}, X_\Omega) = P(X_{E_{hid}} \mid X_{E_{obs}})$ MAR is applicable in many settings and is primarily used to decouple the observation model to deal with complex likelihood functions (see Koller and Friedman for detailed explanations of MAR [105, P.854]). In this context, however, we make the MAR assumption to decouple the observation process to avoid theoretical problems induced by large and complex input spaces. Instead, we can focus on more reliable and easier-to-handle observations or properties. Based on the MAR assumption and their implications, equation (7.5) simplifies to:

$$P(X_b \mid X_{\varphi_1} = \varphi_1, \ldots, X_{\varphi_N} = \varphi_N) \tag{7.7}$$

In summary, the MAR assumption allows the decoupling of the observation process as we can exclude the observation set Ω and thus the observation process. Instead, we focus on a set of observable properties that are both directly observable and allow more precise statements about the state of b. Note that this does not change the theoretical problem that one is facing, i.e. a self-adaptive system safeguarding an AI component still must be considered as POMDP. However, the MAR assumption is accompanied by several advantages that facilitate dealing with the hidden state of b and also the analysis of such systems at design-time (as we will see in the next section).

According to Koller and Friedman [105], although the MAR assumption is applicable in many settings, it must be considered with care. For a limited or not representative set of observable properties, the MAR assumption may not hold. So far we rather enumerated properties such as brightness, sensor noise, weather, etc. However, in areas such as autonomous driving with extremely dynamic environments (where there are fairly more monitorable properties), taking such primitive properties into account is arguably insufficient or does not accurately satisfy the MAR assumption. Therefore, we consider more sophisticated or expressive properties. Basically, we distinguish between *weak* and *strong* properties. We denote properties such as brightness or sensor noise as weak, even though they allow reasoning about the state of an AI component, but do not provide strong assurances. On the contrary, we consider properties such as the determination of safe input regions [201] or variational inference [26] as strong properties as they provide a higher level of assurance. For example, if the input data can be mapped to a precomputed safe region (i.e. a region of the input space for which predictions have been verified to be correct), the probability of an incorrect prediction is zero. The computation of such safe regions, however, is computationally expensive and may not be applicable in situations that require quick reactions such as autonomous driving [116]. Consequently, strong properties need to be weighed in terms of the computational cost and the level of assurance they provide. The MAR assumption can always be sharpened by considering additional observations. Whether this is achieved by strong or soft properties (or a combination of both) must be evaluated by the software architect in terms of the degree of assurance and the computational cost they entail and is a design decision.

Finally, let us consider the design of a MAPE-K-based self-adaptive system by taking into account the insights gained from the MAR assumption. A design sketch is depicted on Figure 7.16. The monitor phase is augmented by the set of properties $\varphi_1, \ldots, \varphi_N$ which are either directly computable from the input/output pairs (x, y) (e.g. safe input regions) or by additional monitors observing external properties (e.g. brightness conditions by external sensors). In the analyse phase, the distribution (7.7) is used to assess whether, for a given set of properties, the probability of observing an incorrect prediction exceeds a threshold ϵ. Note that the distribution can be obtained by applying an upstream sensitivity analysis (as described in section 7.1.2). If this is the case, the plan-phase is triggered, in which an adaptation is planned (e.g. by considering $\varphi_1, \ldots, \varphi_N$) as a countermeasure or to prevent the system from transitioning to an unsafe state. At the very end, the adjustment is carried out in the execute-phase. The knowledge part contains all the information (architecture model, environment model, sensitivity model, assumptions, etc.) relevant to each phase and required for decision-making.

In the next two sections, we discuss how to evaluate self-adaptive systems that follow the previously outlined design. More specifically, we discuss how adaptation strategies are evaluated in terms of maintaining reliability attributes.

7.2.3. Analysing the Monitorable Space

In the last section, we sketched the design of a MAPE-K-based self-adaptive system without being too restrictive in terms of the design space, i.e. the main adaptation logic (such as adaptation planning) still needs to be designed. Hereby, we made the MAR assumption to decouple the observation process which relates to the problem of dealing with high dimensional input spaces from which one must draw conclusions about the hidden state of an AI component. However, the decoupling of the observation process not only has positive effects on the black-box property of AI components but also on the evaluation. The general evaluation approach of self-adaptive systems to safeguard AI components is discussed in the next section; however, this section discusses how the input spaces of AI

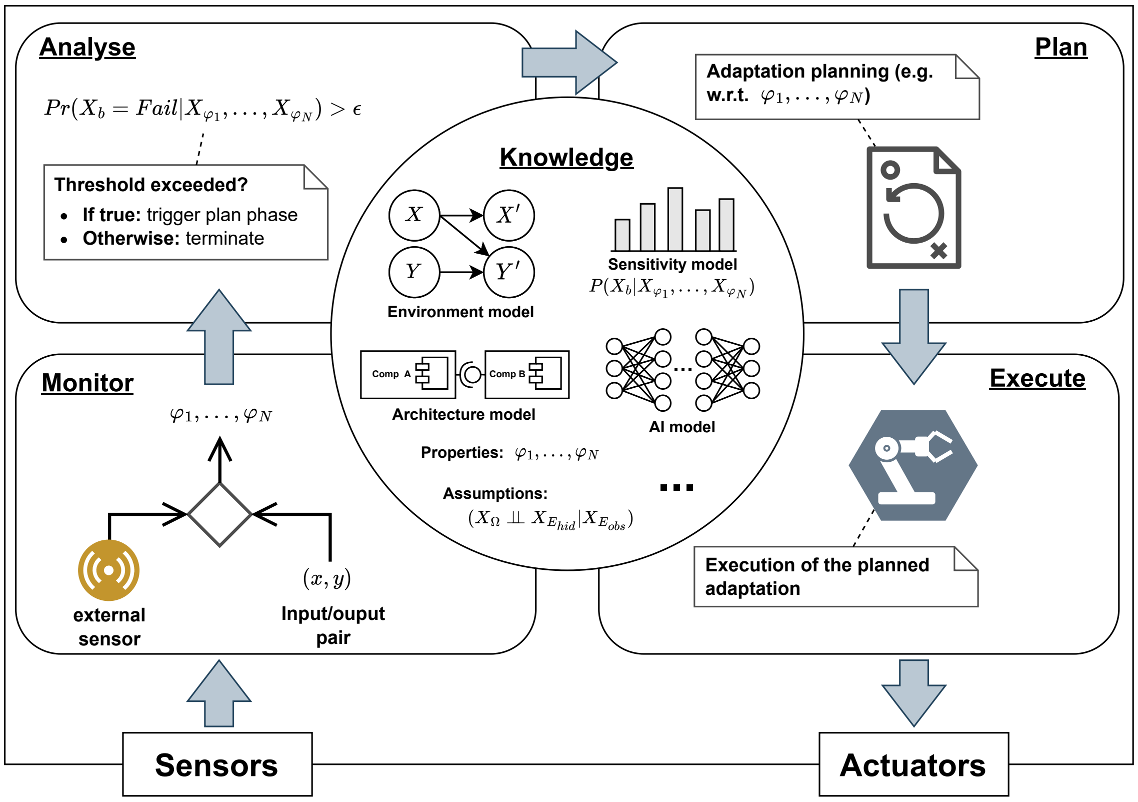

Analyse
$Pr(X_b = Fail|X_{\varphi_1}, \dots, X_{\varphi_N}) > \epsilon$
Threshold exceeded?
If true: trigger plan phase
Otherwise: terminate
Plan
Adaptation planning (e.g. w.r.t. $\varphi_1, \dots, \varphi_N$)
Knowledge
Environment model
Sensitivity model
$P(X_b|X_{\varphi_1}, \dots, X_{\varphi_N})$
Architecture model
AI model
Properties: $\varphi_1, \dots, \varphi_N$
Assumptions:
$(X_\Omega \perp\!\!\!\perp X_{E_{hid}}|X_{E_{obs}})$
Monitor
$\varphi_1, \dots, \varphi_N$
external sensor
(x, y)
Input/ouput pair
Execute
Execution of the planned adaptation
Sensors
Actuators

Figure 7.16.: Design of the MAPE-phases when considering the MAR assumption.

components can be taken into account in the evaluation as peculiarities in the input space are the main source of incorrect predictions and therefore need to be considered in the evaluation.

Without the decoupling, one must represent the observations in the evaluation process. The observations again refer to input-output pairs of the AI component, which is fairly difficult to integrate (especially when dealing with image data where the input space is the pixel space). By the decoupling, however, we focus on a more manageable set of properties $\varphi_1, \dots, \varphi_N$ that allows drawing more precise conclusions of the state of an AI black-box component b; recall distribution $P(X_b \mid X_{\varphi_1} = \varphi_1, \dots, X_{\varphi_N} = \varphi_N)$. That is, instead of considering the input space of b, we rather focus on the distinct property permutations during analysis or evaluation. We denote the space spanned by the set of properties $\varphi_1, \dots, \varphi_N$ the *Monitorable Space*. More formally, we describe the monitorable space Φ as follows:

$$\Phi := Val(X_{\varphi_1}) \times \dots \times Val(X_{\varphi_N}) \tag{7.8}$$

By considering the monitorable space, one can simulate the monitor phase of a self-adaptive system by generating property tuples $(\varphi_1, \dots, \varphi_N)$ from Φ. By decoupling the observation process, we shift the analysis to the monitorable space, bypassing problems caused by large input spaces. We discuss this in more detail in the next section.

However, it should be noted that the monitorable space still grows exponentially in the number of properties such that we possibly encounter state space explosion problems.

As already discussed in section 7.1.3, Monte Carlo methods can be applied or the level of discretisation can be adjusted to reduce the size of the space. However, this might entail a loss of simulation/analysis accuracy. Again, we defer the detailed discussion to section 7.2.4. Moreover, the set of properties must be kept small to circumvent statistical problems (e.g. curse of dimensionality). Consequently, a larger number of properties could be an indication of poor selection. Instead, one can reduce the set of properties by focusing on (if possible) strong properties, i.e. properties that provide stronger assurances. In this way, the monitorable space is reduced because adding strong properties allows the removal of weak properties.

7.2.4. Evaluating Adaptation Strategies

In this section, we discuss how to evaluate adaptation strategies of self-adaptive systems to safeguard AI components at design-time. Therefore, we take all the concepts we have elaborated in this chapter and combine them with the *SimExp* method from chapter 6. Recall that the *SimExp* method provides a framework for evaluating adaptation strategies. In the following, we outline how we extend the *SimExp* framework to evaluate adaptation strategies safeguarding AI components.

For this purpose, however, we have to briefly recap the main concepts of *SimExp*. More specifically, recall Figure 6.1 that provides an overview. The *SimExp* framework needs to be complemented by a reward function and an adaptation strategy provided by the software engineer. Moreover, the framework inputs the initial architecture model, an environment model and a set of model transformations. The latter represents the set of adaptations. The initial architecture model (represented by a PCM model) indicates the start configuration of the system. At this point, the environment model is of most interest because it is directly connected to the uncertainties or properties $\varphi_1, \dots, \varphi_N$. Therefore, recall that during section 7.1 we used the term uncertainties to describe factors that potentially influence the prediction quality of an AI component. In this section, however, we introduced the notion of properties that one can observe to reason about the true state of an AI black-box component. Although both terms suggest different concepts, we consider them the same. In the remainder of the chapter we use the term properties, but would like to emphasise again that we could use the term synonymously with uncertainties. Nonetheless, they form an integral part of the environment because they affect quality objectives. They are represented as environmental variables that make up an environmental state (as already discussed in section 7.2.2). The set of environmental states forms the environmental dynamics which is modelled by the *EnvDyn* metamodel. That is to say, one can use the *EnvDyn* metamodel to model the corresponding DBN that describes the probabilistic behaviour of the properties over time. Based on the DBN, the adaptation strategy can be evaluated based on the sampled environmental states.

It is important to note that the way the *SimExp* framework is used originates from the theoretical insights of decoupling the observation process (see section 7.2.2) and the fact that we can focus the simulation on the monitorable space (see section 7.2.3). Without the decoupling, we would still need to simulate the dynamics of the system governed by the semantics of POMDPs which is associated with severe theoretical issues caused by the observation process. By decoupling the observation process, we focus on the purely observable and monitorable space spanned by a set of properties that allow conclusions to be drawn about the true state of the AI black box component. However, this requires that the MAR assumption is sufficiently approximated by the set of considered properties. Based on both concepts (namely the decoupling of the observation process and the monitorable space), one can model the environment as in any other domain and evaluate adaptation strategies as discussed in Figure 7.16.

Finally, let us return to the reward function, which assigns a reward to each decision of an adaptation strategy. In this context, we are interested in analysing how well a strategy maintains reliability objectives. That is, we have to account for reliability attributes in the reward function. Therefore, we directly reuse our reliability prediction approach from section 7.1 for AI-enabled systems. For reliability prediction, the sensitivity model must be generated beforehand and must match the environmental model, i.e. the uncertainties of the sensitivity model are equal to (or a subset of) the environmental variables of the DBN. That is, the sampled environmental states are used to compute the failure potential of the AI component, which has a global impact on the overall reliability of the system (quantified as probability of success). Recall that we introduced this idea in section 6.3.2.2 where each state of a self-adaptive system (comprising the current architectural configuration and environmental state) is transformed into an analytical model for predicting quality attributes. Suppose that there is a proper uncertainty-refined failure model (referencing the sensitivity model), the environmental state (including the uncertainties) and the architecture model are simply passed to the reliability prediction tool to predict the probability of success. The reliability prediction can be complemented by the application of other simulation/analysis tools, e.g. the Palladio performance prediction tool enables the prediction of additional performance indicators. Based on these predictions the corresponding reward can be determined by the reward function. The idea is illustrated once again in Figure 7.17 for the HRI example system.

The implemented adaptation strategy executes (if required) model transformations to simulate adaptations. To account for the effects of the transformations (or adaptations) on the reliability attributes, the transformations are implemented in a way such that architectural preconditions defined in the uncertainty-refined failure model are satisfied, leading to the activation of a connected architectural countermeasure (see section 7.1.3.3). For example, a model transformation could activate further filter components to preprocess input data before passing it to the AI component. In this case, the filter component is inserted into the architectural model such that a corresponding architectural precondition of the uncertainty-refined failure model evaluates to true and the associated countermeasure is considered in the reliability prediction.

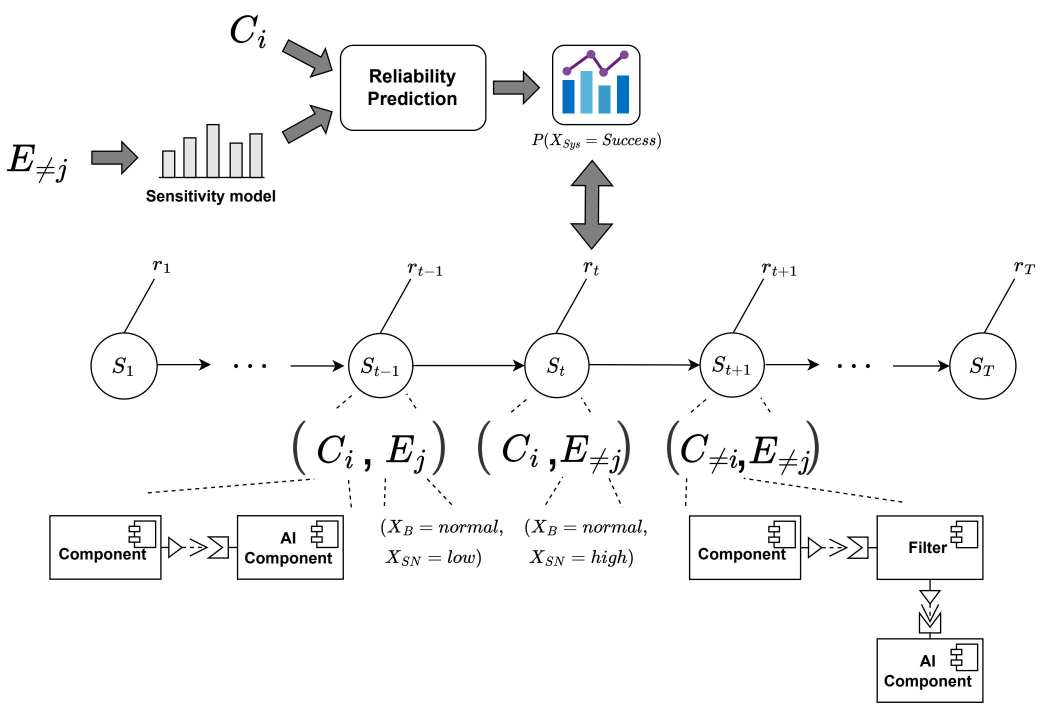


Figure 7.17.: Internal simulation/sample process when evaluating adaptation strategies for safeguarding AI components.

7.3. Implementation

In this section, we briefly provide an overview of the implementation details. Therefore, consider Figure 7.18 which shows an excerpt of the component structure implementing the reliability prediction approach from section 7.1.3.3. For clarity, not all components are displayed, but related components are combined into a single component (e.g. `PCM.Core.*` or `PCM.Reliability.*`).

The approach is implemented by three central components, namely `PCMRelAI.UI`, `PCMRelAI.Predictor` and `PCMRelAI.Model`. Note that we use `PCMRelAI.*` to prefix components related to our presented reliability prediction approach. The `PCMRelAI.UI` component implements the user interface where all necessary details (e.g. model locations and other configurations) are specified by the user. Afterwards, the `PCMRelAI.Predictor` component is used by the provided configuration of the user and implements the core logic of the prediction approach. Since the approach extends the conventional PCM-Rel approach by resolving and updating the failure probabilities of an AI component, the `PCMRelAI.Predictor` component is dependent on the corresponding components implementing PCM-Rel (and also on some components providing core PCM functions, e.g. the PCM model). In addition, the user can apply an AT before the analysis (if needed). Therefore, the `PCMRelAI.Predictor` reuses the appropriate functions provided by the `PCM.AT.*` components to apply the specified AT. Finally, the `PCMRelAI.Model` component is referenced which provides the uncertainty-refined failure model and corresponding code artefacts based on the EMF framework.

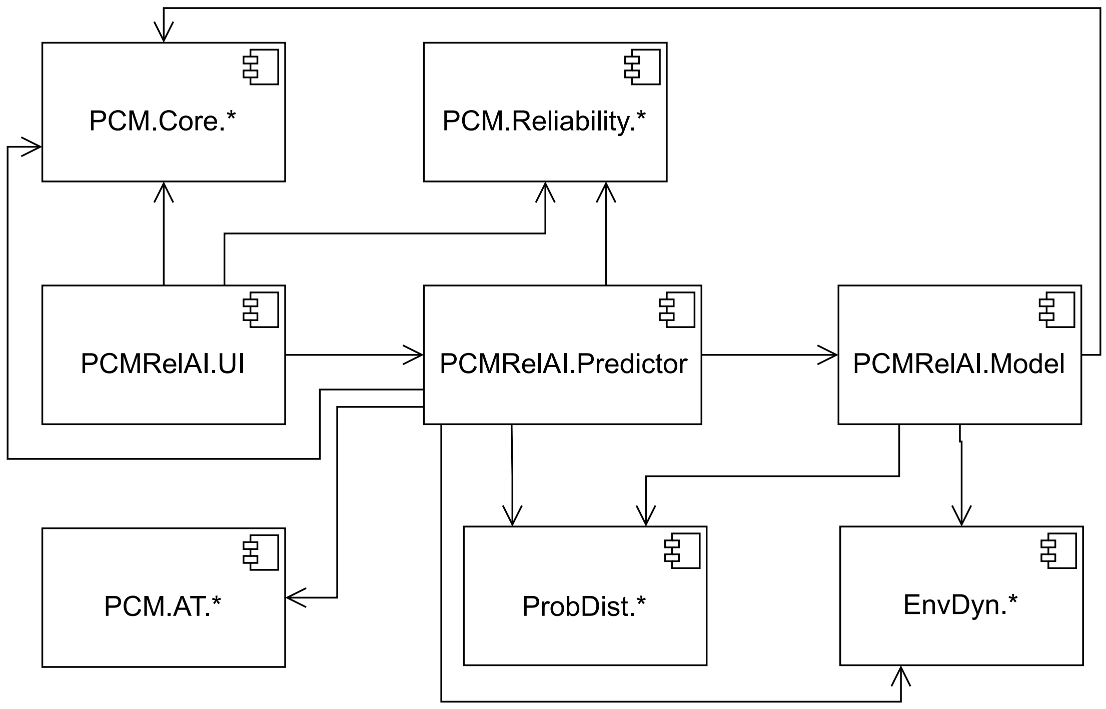


Figure 7.18.: Excerpt from the dependency graph of the components for implementing the reliability prediction approach for AI-based systems

Similar to the implementations in the other chapters, our tooling is implemented in the context of Eclipse, more precisely as an Eclipse plug-in.

7.4. Assumptions and Limitations

In the following, we discuss the assumptions and limitations of the presented approach.

Assumptions As in other chapters, the presented approach is associated with several assumptions. The first assumption to mention is the MAR assumption. We make the MAR assumption to decouple the observation process by focusing on a set of observable properties which allow conclusions to be drawn about the true state of an AI component. As we have already discussed in this chapter, this assumption might be too strong in some settings. However, the MAR assumption can be highly approximated by considering properties that provide high assurances regarding the state of an AI component. We have discussed this issue in detail in section 7.2.2 and refer to said discussion.

Moreover, we assumed independence between hardware resource patterns ψ, usage model U and the observed uncertainties or properties $\varphi_1, \dots, \varphi_N$, i.e. $(X_{\varphi_1}, \dots, X_{\varphi_N} \perp\!\!\!\perp X_\Psi \perp\!\!\!\perp X_U)$. We argue that this assumption is accurate in settings where $\varphi_1, \dots, \varphi_N$ refer exclusively to AI-specific uncertainties, or at least accurate enough to perform design-time analysis where simplifying assumptions are fairly common. Moreover, when considering equation (2.3) on page 34 of PCM-Rel, similar assumptions must have been made regarding X_Ψ and X_U. In settings where $\varphi_1, \ldots, \varphi_N$ are related to hardware failure types, the assumption does not hold anymore (as discussed in section 7.1.3.3). However, this is currently unrelated to our considered use case but is a topic for future work when the our approach is to be extended to a wider range of use cases.

Limitations Also, regarding the implemented functionality of the approach, there are a couple of limitations. First, we outlined in section 7.1.3.3 how our presented reliability prediction approach for AI-enabled systems can be generalised to a wider range of use cases. However, the discussion is only theoretical and not yet implemented. We plan to make up for this in future work.

Our reliability prediction approach for AI-enabled systems is based on PCM-Rel [33] which is based on PCM. Consequently, one must strictly use PCM as ADL to model the software architecture of the AI-enabled system. Although PCM is a very powerful language to describe numerous software systems, it might be not perfectly suited to model the architecture of systems from other domains, e.g. embedded systems. Nonetheless, PCM can be used anyway and thus our reliability prediction approach. However, using an ADL that allows more accurate modelling of other types of systems would be desirable. The same applies to the semantics of the reliability prediction procedure. Our reliability approach builds upon the existing reliability approach PCM-Rel of Brosch [33]. Therefore, we also inherit the semantics of the PCM-Rel reliability prediction procedure. In terms of our reliability prediction approach (and the inherited prediction semantics of PCM-Rel), the effect of a wrong prediction is directly evaluated at the modelled AI component and not in the other parts of the system which might be affected by the corrupted prediction. However, since our primary goal is to evaluate design decisions in terms of architectural safeguards, we argue that the reliability prediction semantics are sufficiently accurate.

In section 7.1, we presented two architectural patterns, namely the filtering and n-version programming pattern. We modelled both patterns by using the formal language of ATs (see section 2.3.1.2). Each AT is accompanied by a completion, i.e. the model transformation, which weaves the pattern into a specific architectural model (or more precisely PCM model). In this context, the completions for both ATs are merely applicable to event-based PCM models. This results from the fact that we considered (at least for AI-enabled systems) only systems adhering to event-based architectures. In future work, however, we plan to extend the completions to apply the ATs to PCM models that communicate synchronously through operation calls.

7.5. Summary

In this chapter, we presented design-time approaches to evaluate architectural safeguards regarding reliability at design-time. We considered predictive uncertainty as first-order uncertainty. Moreover, we defined a set of second-order uncertainties (e.g. brightness, sensor noise) that are directly observable and allow conclusions to be drawn about the true state of the AI component. These secondary uncertainties serve as the main components for dealing with the black-box nature (or hidden state problem) of AI components and form the entry point for the analysis of systems with AI components.

In section 7.1, we presented an approach to model and analyse architectural patterns specifically designed to deal with AI uncertainties. We focused on two patterns, namely the filtering and n-version programming pattern. We used the AT (architectural template) language to represent the patterns as models (see section 7.1.1). This allows a software developer to select an AT and apply it to an architectural model (in this context a PCM model) such that the impact can be analysed. After that, in section 7.1.2, we explained how to use sensitivity analysis to address the black-box property of AI components. The key idea is to generate a sensitivity model to approximate the predictive uncertainty of an AI component given a set of secondary uncertainties which affect the predictive uncertainty. The secondary uncertainties must be known in advance by consulting a domain expert or conducting an upstream domain analysis. Based on the sensitivity model, we presented a reliability prediction approach for AI-enabled systems. More specifically, we extended an existing reliability approach for classic information systems, namely PCM-Rel (see section 7.1.3.3). Internally, PCM-Rel makes use of a failure model for annotating specific elements of the architecture model with failure probabilities. We extended PCM-Rel by using the sensitivity model to represent the predictive uncertainty of the AI component. The extended reliability prediction approach complements the developed ATs by analysing the impact of an architectural safeguard (modelled as AT) in terms of reliability.

In the second part of this chapter, we generalised the previously presented concepts to self-adaptive systems. In this context, we consider self-adaptive systems as a safeguarding mechanism for AI components, where the adaptation strategy (governing the adaptation process) is to be evaluated. Therefore, we reused the *SimExp* framework from chapter 6. Moreover, we extended the framework by integrating the aforementioned reliability approach for AI-enabled systems. Thus, an adaptation strategy is analysed and evaluated in terms of maintaining reliability attributes. Finally, we concluded the chapter by discussing the assumptions and the limitations.

8. Classes of Architectural Dependability Assurance for AI-Enabled Systems

In the previous chapter, we presented a model-based approach for the reliability prediction of AI-enabled systems (for both, static and self-adaptive systems). The approach is based on models that abstract the software architecture, adaptations (in the case of self-adaptive systems) and the environment in which the system is operating. However, the question that arises at this point is whether we can always abstract systems and environments to conduct design-time analysis. In self-driving cars, for example, the dynamics of the environment are so manifold and encompass numerous possible scenarios that these environments are difficult to capture by an environment model. Depending on the quality attributes one wants to analyse, there could be a lack of simulation or analysis tools for prediction. Even if one can predict a quality attribute (or another type of system property), the state space to be analysed might be too large to check whether all states satisfy the system property. The circumstances under which a system can be analysed at design-time depend not only on the type of system (i.e. static or self-adaptive) but also strongly on the system properties that are to be analysed. In particular, for systems with AI components, where the hidden state problem of AI models prohibits checking whether the model is executing incorrectly, one has to resort to particular properties (e.g. neuron coverage, robustness or simply brightness in the input image) that are used to draw conclusions about the true state of the AI component. However, such properties are not always observable (i.e. only for a subset of the input space) or computationally expensive, but are highly relevant to determine system-level properties.

To account for a wide range of system-level properties, we consider those related to *Dependability*. According to Sommerville [173], dependability encompasses four essential quality dimensions, namely *Availability*, *Reliability*, *Safety* and *Security*. Therefore, we consider system-level properties that can be assigned to one of the enumerated quality dimensions.

In this chapter, we propose four classes of architectural dependability assurance, namely *Static Analysability*, *Monitor Analysability*, *A-posteriori Analysability* and *Non-Analysability*. Furthermore, we introduce a classification structure (based on generic classification dimensions) that allows AI-enabled systems and their domain to be classified into one of these classes. The classes form a natural order where static analysability is the most desirable class (i.e. the system is fully analysable at design-time) and non-analysability refers to the worst class (i.e. the system cannot be analysed at all and thus no assurances can be made). Each class is associated with a particular system-level property for which assurance can be given either at design-time or at runtime or not at all. For example, static analysability refers to the ability to fully analyse a system at design-time such that, as a result, assurances w.r.t. the considered system-level property can be made at design-time. Additionally, the classification of the system to be engineered (and w.r.t the operating environment) into one of the classes gives software engineers an intuition about the problem domain itself. Systems which are statically analysable (regarding a system-level property the system must satisfy) are arguably easier to develop than systems that are classified as non-analysable. Finally, we envision the classes to serve as assurance arguments in assurance cases. In assurance cases, claims are made that the system meets the (safety-related) requirements and arguments and evidence are provided to justify these claims [25]. Therefore, we consider our classes of dependability assurance as further argumentation and evidence that contribute to the degree of belief to justify the claims within an assurance case. For example, if it can be argued (in terms of classification structure) that a system is classified into static analysability, this strengthens the assurance argument considerably as it implies that evidence can be provided at design-time. If, on the other hand, a system can only be classified into lower-ordered classes, then the assurance argument is weakened as it implies that either only partial analysis is possible at design-time or even only at runtime.

As a last remark, the classes are inspired by the insights gained from the previous chapters. Nonetheless, they are formulated in a way that generalises the concepts of this thesis. However, due to the rather limited number of approaches to the analysis of AI-enabled systems at design-time, it is difficult to assess whether the dependability assurance classes (and the classification dimensions) are complete. Therefore, we consider them as a starting point, but one that is still preliminary and needs to be further investigated in future work. The contribution of this chapter has been published [160].

In this chapter, we address research question **RQ4**:

Research Question 4: How to assess the extent to which dependability assurances can be given for an AI-enabled system?

Therefore, we developed the aforementioned classes of analysability and respective classification dimensions to address the sub-research questions **RQ4.1** and **RQ4.2**

Research Question 4.1: What are appropriate classes of architectural dependability assurances?

Research Question 4.2: What are the suitable dimensions for classification?

The chapter is structured as follows: In section 8.1, we introduce the four classes of architectural dependability assurance. In section 8.2, we discuss the distinct classification dimensions and provide an overview of the classification structure (w.r.t. the dimensions). Afterwards, in section 8.3, we demonstrate the applicability of the classification structure by applying it to representative AI-enabled systems. Finally, we summarise the chapter in section 8.4.

8.1. Classes of Architectural Dependability Assurance

In this section, we present the distinct classes of architectural dependability assurance. Basically, the idea is to classify systems and the environment (in which the system operates) into one of the classes. Each class indicates the extent to which design-time and runtime assurances can be made. We consider an assurance or assurance case as defined in [68]: "A reasoned and compelling argument, supported by a body of evidence, that a system, service or organisation will operate as intended for a defined application in a defined environment" [68]. Moreover, an argument "[...] is defined as a connected series of claims intended to establish an overall claim" [68]. Our classes determine the time at which these arguments can be given, i.e. at design-time or runtime. Therefore, classification into one of the classes itself becomes an assurance argument. Systems that can be assigned to classes for which assurances can be given at design-time are more favourable because evidence can be given at design-time and not only at runtime (making the assurance case more compelling).

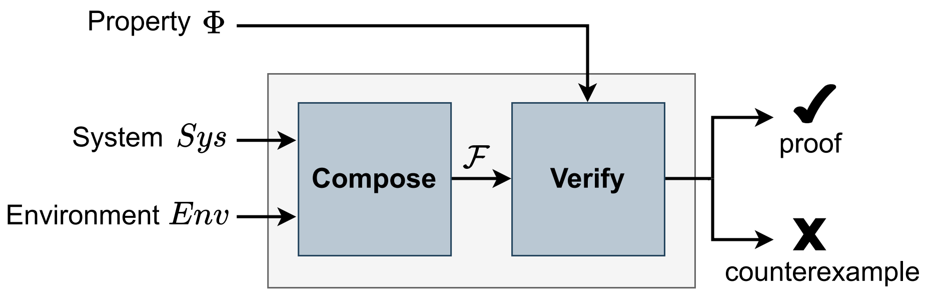


Figure 8.1.: Typical formal verification process taken from [165].

For the definition of the classes, we orient ourselves on the traditional verification process (see Figure 8.1) for software systems or programs. In principle, in formal verification, the primary objective is to verify a property Φ based on a system model *Sys* and environment model *Env* (in which the system operates). The *Sys* and *Env* models are composed and generate a formal model $\mathcal{F}$ which serves as a basis to verify property Φ. The output is a yes or no answer indicating that the property could either be verified (with an optional proof) or not (with a corresponding counterexample that falsifies Φ). Originally, Seshia et al. [165] discuss how this process can be adapted into a holistic procedure for the verification of AI systems. In this work, we make use of the individual elements and their interplay to define our distinct classes.

As a final remark, recall that the classes induce a natural ordering in which static analysability is to be considered the most preferable class, followed by monitor and a-posteriori analysability; non-analysability refers to the worst possible class. Similarly, we argue that the same natural order exists in the classes static analysability, monitor analysability and a posteriori analysability in terms of making assurances. More specifically, assurances that can be made in a particular class must automatically apply to lower-order classes as well, i.e. a property that can be assured at design-time can also be assured at runtime (if the property is efficiently computable). For example, any system-level property that is (say)

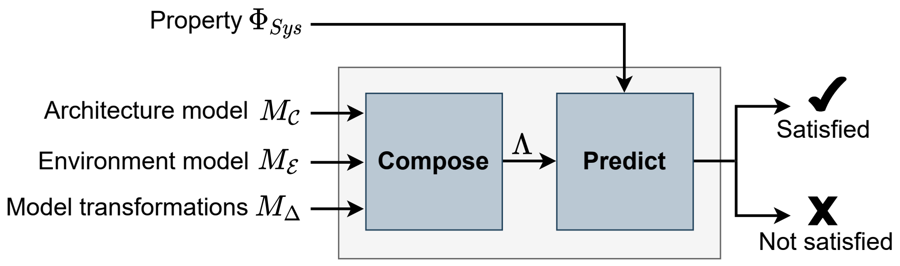


Figure 8.2.: The adapted dependability assurance process for design-time analysis based on the classical verification process [165].

statically analysable is also monitor-analysable, but not vice versa. However, this does not apply to non-analysability because any system-level property that is static, monitor- or a-posteriori analysable cannot be non-analysable.

8.1.1. Static Analysability

In this section, we define the dependability assurance class static analysability which refers to the ability to statically analyse properties at design-time. When classified to static analysability, the environment in which the system operates is considered to be well known or there is sufficient information available to analyse the environment. Environments which are well understood by (e.g.) domain experts allow sufficient information (such as assumptions) to be included to describe how the environment behaves, how it interacts with the system or what the main disturbances are.

To be more formal, let us consider Figure 8.2 which depicts the traditional verification process adapted for our purposes. In this case, the system model *Sys* and environment model *Env* correspond to our architecture model M_C and environment model (capturing the environmental dynamics) $M_{\mathcal{E}}$. As further input, we consider a set of model transformations M_Δ that abstract adaptations. However, the model transformations are optional, i.e. they must be only specified if self-adaptive systems are to be analysed. A set of analytical or formal models Λ is generated by composing the specified models that serve as a foundation to analyse the given properties (e.g. an MDP associated with further analytical models to predict system-level properties for each state). What was originally formulated as property Φ now refers to system property Φ_{Sys} (e.g. the success probability of the system). Finally, based on Λ it is predicted whether Φ_{Sys} is satisfied or not. Note that we deliberately use the term "predict" instead of "verify" because we do not aim to verify whether Φ_{Sys} holds in all states, but sometimes calculate an average or expectation of the individual predictions of Φ_{Sys} and check whether the expectation satisfies Φ_{Sys}. For example, recall our reliability prediction approach from chapter 7, where an overall success probability is derived by aggregating the individual conditional success probabilities (conditional on $\varphi_1, \ldots, \varphi_N$). However, the term "predict" suggests that the considered system-level properties must be computable. Therefore, we require that the prediction procedure is efficiently Turing-computable.

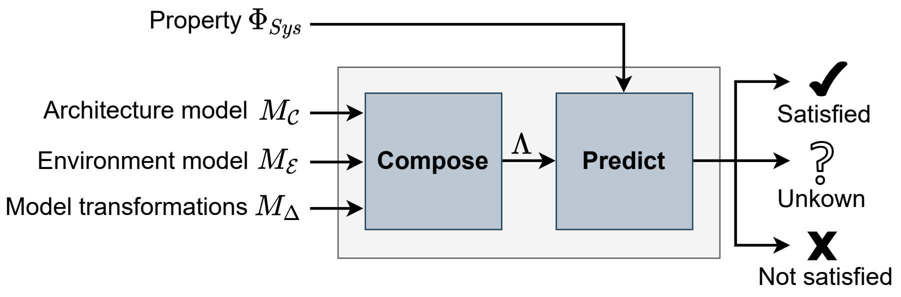


Figure 8.3.: The adapted dependability assurance process for partial design-time analysis based on the classical verification process [165].

Essentially, the dependability assurance process from Figure 8.2 reflects exactly what we define as static analysability:

Definition 31 (Static Analysability). *An AI-enabled system and its operating environment are said to be statically analysable if there exists a system or architecture model M_C, an environment model $M_\mathcal{E}$ and a set of model transformations M_Δ such that a set of formal models Λ are generated or derivable that allow predicting with an acceptable degree of accuracy whether M_C satisfies system property Φ_{Sys} w.r.t. $M_\mathcal{E}$.*

Note again that M_Δ is not required if non-adaptive software systems are analysed.

In simple terms, static analysability refers to settings where, for a given system property Φ_{Sys}, there are models that abstract the system (and adaptations in the case of self-adapting systems) and its operating environment in such a way that it is possible to predict whether the system satisfies Φ_{Sys} w.r.t. the operating environment. Static analysability is the most desirable class into which a system and its environment can be classified, as evidence for the assurance argument can be given at an early stage of development, making the assurance case more reasoned and compelling.

8.1.2. Monitor Analysability

The next class of dependability assurance refers to monitor analysability. Similarly to static analysability, monitor analysability allows analysing whether system properties are satisfied at design-time. This time, however, there might be a situation where for some states we cannot predict whether Φ_{Sys} is satisfied or not, i.e. the fulfilment is unknown. This circumstance is illustrated in Figure 8.3 by extending the possible outcomes to include the case of an unknown state.

The term “monitor” of class monitor analysability seems a bit contradictory because one would rather associate it with runtime monitoring, and yet we consider it a class for which design-time assurances can be given. In fact, the term “monitor” refers to a runtime capability, namely the ability to construct runtime monitors for subregions of the state space for which no assurances can be given. Since the analysability of monitors implies that no assertions about Φ_{Sys} can be made for some subregions of the state space, it is essential to use additional monitors at runtime to reason about Φ_{Sys} and avoid potentially hazardous situations. Restricting the state space to exactly such subspaces allows software engineers to construct additional runtime monitors. For the remaining subspace (in which statements about Φ_{Sys} can be made), assurances can be given and appropriate countermeasures taken.

Definition 32 (Monitor Analysability). *An AI-enabled system and its operating environment are said to be monitor-analysable if there exists a system or architecture model M_C, an environment model $M_{\mathcal{E}}$ and a set of model transformations M_Δ such that a set of formal models Λ are generated or derivable that allow predicting for a subset of states with an acceptable degree of accuracy whether M_C satisfies system property Φ_{Sys} w.r.t. $M_{\mathcal{E}}$.*

The definition of monitor analysability is only different from static analysability in that only for a subset of states it can be predicted or determined whether Φ_{Sys} is satisfied. For example, let us consider an MDP as a formal model for self-adaptive systems, the monitor analysability indicates that only a subset of the state space can be checked w.r.t. Φ_{Sys}. In situations where not every state of a self-adaptive system is fully observable, there may be situations where it cannot be determined with sufficient certainty whether the system is in a particular state (due to a lack of evidence) such that it is not possible to determine whether Φ_{Sys} is satisfied.

Although we do not specify a certain percentage of the states for which one can expect a verdict regarding Φ_{Sys}, it is reasonable to do so in practice. For example, if for a large percentage of states, it is not known whether Φ_{Sys} is satisfied, the knowledge of the remaining states is unlikely to be sufficiently informative.

8.1.3. A-posteriori Analysability

A-posteriori analysis states that the system and its operating environment cannot be analysed at design-time; that is, it can not be determined whether the system satisfies the system-level properties w.r.t. the environment.

Definition 33 (A-posteriori Analysability). *An AI-enabled system and its operating environment are said to be a-posteriori analysable if there exist no models M_C, $M_{\mathcal{E}}$ and M_Δ such that (based on Λ) it can be predicted whether the system property Φ_{Sys} is satisfied. Instead, assurances can only be given at runtime; that is, it can be determined whether Φ_{Sys} is satisfied at runtime.*

In this case, only assurances at runtime can be given. System engineers are advised to perform analysis after a system crash or hazardous event occurred, e.g. based on logging data. Each arising event of the system is eventually logged and documented by the system. In case of a system crash (or other types of events), the logged data can be analysed *A-Posteriori* to investigate the source of the event. This trial-and-error approach incrementally improves the system quality.

8.1.4. Non-Analysability

Non-analysability means that no assurances can be given at all. Non-analysability is the worst possible class a system and its environment can be classified into.

Definition 34 (Non-Analysability)**.** *An AI-enabled system and its operating environment are said to be non-analysable if they are at least non-a-posteriori analysable. That is, no assurances regarding system property Φ_{Sys} can be given.*

The simulation of systems is a common practice in engineering disciplines before realising the system physically. Simulation is essential to analyse the impact of design decisions regarding various quality attributes of interest. The capability of analysing the properties of a system before physically realising it constitutes the central characteristic of traditional engineering. However, if systems cannot be analysed such that no predictions or statements can be made about certain properties, engineers refuse to realise the system. This is mainly because, in traditional engineering, the consequences of system damage usually have an enormous impact on safety or may cause high financial damage. As soon as the properties can not be guaranteed, the system is not engineered.

8.2. Classification Structure

In this section, we present the classification structure that allows classifying systems into one of the dependability assurance classes. Therefore, we first present the distinct classification dimensions and give an overview of the classification structure afterwards. In the end, we envision how to use the classification structure and its dimensions as a blueprint to build dependability cases, i.e. assurance cases for dependability-specific system-level properties.

8.2.1. Classification Dimensions

In the following, we discuss four classification dimensions, namely *Abstractability*, *Approximation of the System Dynamics*, *Analytic Capacity* and *Fail-Safe*. Especially, the analytic capacity refers to a dimension that we explicitly elaborated to assess and classify the analytical potential of an AI component and constitutes a significant part of the contribution presented in this chapter.

We would like to emphasise again that we do not consider the dimensions to be complete, but rather preliminary because they are highly subjective and due to the far too few design-time approaches for AI-enabled systems to assess the completeness and appropriateness of the classification structure. Nevertheless, we consider it as (i) a starting point for evaluating systems and their environments, (ii) a guideline for software engineers and (iii) a basis for building/structuring dependability cases. In addition, the classes are subject to future work.

8.2.1.1. Abstractability

The first dimension refers to abstractability, i.e. the ability to find model abstractions of the system M_C and operating environment $M_{\mathcal{E}}$ (and M_{Δ} in the case of self-adaptive systems) such that a system-level property Φ_{Sys} is predicted with a sufficient degree of accuracy. Abstractability simply requires that there exist models that capture the essential characteristics of a system and its environment. For some systems, for instance, there might be no suitable ADL that captures all the relevant details of the system being analysed; or the operating environment is too complex to be modelled, i.e. there are too many variables and relationships between variables to be modelled with acceptable effort. Abstractability is to be considered as an entry point that a system is analysable at design-time w.r.t. Φ_{Sys}.

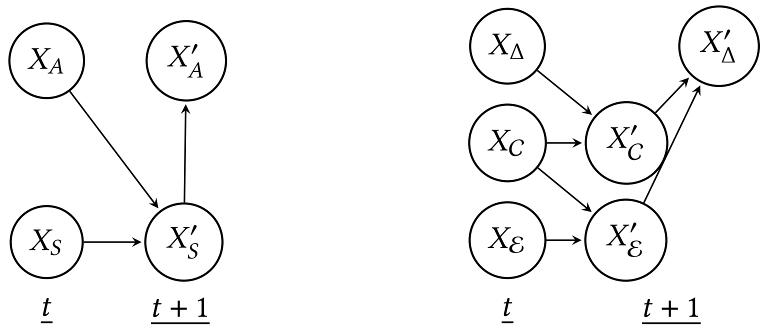
In general, it is difficult to objectively determine whether abstractability is met, and there is also no universal metric that measures whether a system and its environment are sufficiently abstracted for design-time analysis (this applies to other dimensions as well). Nevertheless, we argue that depending on the modelling languages and the respective analysis tools used, it can be argued whether abstractability is sufficiently satisfied. We return to the discussion in the section 8.2.3.

8.2.1.2. Approximation of the System Dynamics

The second dimension is concerned with the approximation of the system dynamics. Let us assume that a system and its environment can be accurately abstracted to predict a system property Φ_{Sys}. If abstractability is given, we can determine whether Φ_{Sys} is satisfied for a particular architectural configuration of a system (i.e. M_C) w.r.t. some environmental state (i.e. M_E). However, we still have to account for the dynamics of the system, i.e. how the system moves through the state space.

Therefore, we consider a software system as a stochastic process. In the case of self-adaptive systems, for example, this stochastic process is described by an MDP (see section 4.3.1). But also for static systems, one can consider Markov chains to describe the dynamics (where only the environment probabilistically evolves while the system configuration always remains the same). The system dynamics describe now how a system is potentially evolving through the state space. To analyse whether Φ_{Sys} is sufficiently satisfied by the system, one must simulate or sample the distinct trajectories which embody the system dynamics.

We argue, however, that this is rather relevant for self-adaptive systems. Although one might model static systems as (e.g.) Markov chains, sampling trajectories from the Markovian is arguably superfluous because only the environment is changing but not the system configuration. This means that it is sufficient to check (e.g. in a brute force manner) whether Φ_{Sys} is satisfied for each environmental state, see for example [149]. In self-adaptive systems, however, capturing the system dynamics accurately is paramount since the dynamics are strongly influenced by the adaptation strategy. That is, the way how self-adaptive systems move through the state space is dictated by the adaptation strategy



(a) The general MDP represented as DBN. **(b)** The MDP of a self-adaptive system represented as DBN.

Figure 8.4.: DBN representation of an MDP based on [160] (originally adopted from [190]).

which adapts the system configuration in certain environmental states. Thus, some states are visited with high probability and others may never be. Moreover, since adaptation strategies must be evaluated in terms of the uncertainty *Parameter over time*, the long-term effects of adaptations manifest themselves exclusively in the trajectory space. Therefore, the system dynamics must be adequately captured during the evaluation of distinct strategies.

To further explain what we consider to be a good approximation of the system dynamics, recall how we defined the system dynamics of a self-adaptive system (i.e. by MDPs), which is primarily captured by the transition function $t_\mathcal{S}$. Therefore, consider Figure 8.4 which depicts the system dynamics of an MDP conventionally (i.e. Figure 8.4a) and the instantiated version for self-adaptive systems (i.e. Figure 8.4b). Now, consider the equations (8.1) and (8.2) which describe the system dynamics of Figure 8.4a and Figure 8.4b, respectively.

$$P(X'_S, X'_A | X_S, X_A) = P(X'_S | X_S, X_A) \cdot P(X'_A | X'_S) \tag{8.1}$$

$$P(X'_\mathcal{S}, X'_\Delta | X_\mathcal{S}, X_\Delta) = \underbrace{P(X'_\mathcal{C} | X_\mathcal{C}, X_\Delta) \cdot P(X'_\mathcal{E} | X_\mathcal{C}, X_\mathcal{E})}_{P(X'_\mathcal{S}|X_\mathcal{S}, X_\Delta) = t_\mathcal{S}} \cdot \underbrace{P(X'_\Delta | X'_\mathcal{C}, X'_\mathcal{E})}_{P(X'_\Delta|X'_\mathcal{S}) = \pi} \tag{8.2}$$

It can be seen that $P(X'_S, X'_A | X_S, X_A)$ describe the system dynamics of an MDP which factorises into two products. The first product (i.e. $P(X'_S | X_S, X_A)$) refers to the transition function t; the second distribution describes the policy π (recall from section 2.4.2). When applied to self-adaptive systems (see equation (8.2)), the system dynamics factorise into three distributions. The first two represent the transition function $t_\mathcal{S}$ and the last reflects the adaptation strategy (i.e. policy π). Because we evaluate adaptation strategies, π is a deterministic function which can be represented as probability distribution by using the indicator function, i.e. $\mathbb{1}_{\pi(S)=\delta}$. Moreover, from theorem 4.3.1 on page 83 we know that the transition function factorises into two distributions, i.e. exactly those as shown in equation (8.2). Hereby, the first distribution (i.e. $P(X'_\mathcal{C} | X_\mathcal{C}, X_\Delta)$) is again represented by the indicator function (see theorem 4.3.1). Thus, only the distribution $P(X'_\mathcal{E} | X_\mathcal{C}, X_\mathcal{E})$ remains which refers precisely to the interdependency of the system and its environment from section 4.3.2. The interdependency of the system and its environment embodies what we understand under approximating the system dynamics. When considering self-adaptive systems in the same way as we did in section 4, it is crucial to make accurate assumptions about the interplay between the system and its environment because it (along with the adaptation strategy) determines how a self-adaptive system evolves.

Although we have motivated the dimension "approximation of system dynamics" from the perspective of our consideration of system dynamics, the bottom line is to find suitable assumptions or approximations of system dynamics that are strongly correlated with the behaviour of the system and its environment. How these assumptions or approximations are encoded is dependent on the tool used for design-time analysis. Some approaches or tools already have these implemented internally in a simulator (e.g. [54, 16]); other approaches require these to be encoded manually (like our approach or [38]). In any case, it is of utmost importance that the assumptions are sufficiently accurate to be able to conduct a design-time analysis.

8.2.1.3. Analytic Capacity of AI-enabled Systems

In this section, we present the concept of the analytic capacity of AI-enabled systems. While the abstractability and approximation of system dynamics dimensions are mainly to determine whether an AI-enabled system can be abstracted and simulated/analysed regarding system property Φ_{Sys} at design-time, the analytic capacity indicates the potential of analysing AI-specific properties in general (i.e. at design-time or runtime). The analytic capacity is spanned by three factors, namely the *State Space Complexity*, *Φ_b-Monitorability* and *Input-Output Monitorability*.

For the analytic capacity, we consider the AI-specific property Φ_b. In contrast to system property Φ_{Sys}, Φ_b is purely related to AI-specific properties that, for instance, allow conclusions to be drawn about the state of an AI component. Thus, the properties Φ_{Sys} and Φ_b are distinct. However, we assume that property Φ_{Sys} depends on Φ_b such that the capability of monitoring Φ_b strongly impacts the way of observing or analysing Φ_{Sys}. In terms of our reliability prediction approach for AI-enabled systems, for example, the capability of observing $\varphi_1, \ldots, \varphi_N$ is paramount to derive a sensitivity model (capturing the predictive uncertainty) which is required to predict the system-level property $P(X_{Sys} \mid X_U)$ (i.e. the system's success probability).

State Space Complexity The first factor of the analytic capacity refers to the state space complexity of the AI-enabled system. The state space complexity refers to the various states of an AI-enabled system one must analyse to determine whether Φ_b applies or not. Intuitively, the larger the state space the more states need to be checked to provide assurances. Moreover, large state spaces indicate that there are potentially more malicious states or corner cases leading to erroneous behaviour.

As discussed in section 4.4.1, the state space complexity of static systems refers to the design space. When using static systems instead of self-adaptive, it is assumed that there exists at least one system configuration which satisfies the quality requirements sufficiently. In this case, software engineers have to explore the design space to identify the best possible configuration, e.g. by applying optimisation techniques such as [106]. When developing self-adaptive systems, however, exploring the design space is not sufficient because an adaptation strategy is to be engineered. The temporal aspect of self-adaptive systems induces the trajectory space, i.e. the number of individual trajectories a self-adaptive system might encounter. Thus, the adaption strategy must be engineered in a sense that the strategy selects only "good" trajectories, i.e. the sequences of states in which the quality requirements or objectives are met. Intuitively, one would associate static software systems with low state space complexity and self-adaptive system with high state space complexity. However, this may not be the case in general, nor is there any guarantee for this intuition.

Therefore, we distinguish three cases when determining the complexity of state spaces, namely Φ_b-explorable, Φ_b-sufficiently explorable and non-explorable. State spaces that are considered to be Φ_b-explorable are either fully explorable (i.e. it is possible at design-time to check each state) or statistically sufficiently explorable (i.e. not all states can be visited but the portion of visited states is sufficient to make statements about Φ_{Sys} w.r.t. Φ_b). A system is said to be Φ_b-sufficiently explorable if a large portion of the state space can be explored, i.e. a sufficiently large partition of the state space can be analysed to reason about Φ_b. Finally, non-explorable refers to state spaces that are neither Φ_b-explorable nor Φ_b-sufficiently explorable. We will see examples of all three variants in section 8.3.

Φ_b-Monitorability The second factor relates to Φ_b-monitorability which defines the extent to which an AI-specific property Φ_b of an AI component is monitorable. More specifically, we consider four types of monitorability of an AI component w.r.t. Φ_b, namely *Verifiability*, *Fully Monitorability*, *Partially Monitorability* and *Non-Monitorability*.

The first type refers to AI components where one can prove that Φ_b holds:

Definition 35 (Verifiability)**.** *An AI component is said to be verifiable w.r.t. Φ_b if it is possible to prove with a justifiable effort that Φ_b is always satisfied.*

Clearly, verifiability of AI components is rather hard to observe in practice for AI components which have a certain level of complexity. However, for very simple AI components with small input spaces, it might be possible to prove (e.g.) in a brute-force manner that the AI component always produces the correct output for any input. Note that we deliberately do not further specify the term "justifiable" because it depends heavily on various factors, such as experience (how experienced is the developer?), time (how much time does the verification process take?), computational power (how much computational resources are needed?), complexity (how complex is the AI component?), etc.

The next type of monitorability, namely fully monitorability, relaxes this hard requirement that Φ_b can be proven:

Definition 36 (Fully Monitorability)**.** *An AI component is said to be fully monitorable if there exists a decision procedure which decides in a reasonable time whether the components' current behaviour satisfies Φ_b or not.*

Instead of proving that an AI component satisfies Φ_b, fully monitorability relates to situations where a monitor component can be constructed that determines whether Φ_b is satisfied or not at any point in time. This relaxes the assumption regarding the existence of proof that either shows the correctness or incorrectness of an AI component working properly. Additionally, in practice, it is more likely that one can construct a monitor instead of finding proof. The performance of the decision procedure, however, is crucial. For example, if there is a decision procedure that determines whether Φ_b is satisfied or not, but takes too much time to reach that conclusion, the decision procedure is not applicable. This refers to what we mean by "reasonable time". Again, "reasonable time" is intentionally not specified further because it depends strongly on the context of the application.

Technically, the performance of a decision procedure depends on particular factors, e.g. for some input data it might take more time to check whether the AI component satisfies Φ_b. In this case, for a subset of states, the procedure might reach a verdict. Furthermore, it is known from the field of "explainable AI" that for some AI components only local explainers can be constructed that generate explanations for a subset of the input space. That is, we are only able to check for a subset of states whether Φ_b is satisfied. Partially monitorability accounts for such situations and relaxes the requirements made for fully monitorable AI components.

Definition 37 (Partially Monitorability)**.** *An AI component is said to be partially monitorable if there exists a decision procedure which decides in a reasonable time whether the component's current behaviour satisfies Φ_b or not or reaches an inconclusive verdict.*

Again, we assume that a verdict is reached for a sufficiently large percentage of states

Finally, the last type of monitorability refers to non-monitorability:

Definition 38 (Non-Monitorability)**.** *An AI component is said to be non-monitorable if it is neither partially monitorable nor verifiable.*

Non-monitorability represents the worst-case scenario because no statements can be made w.r.t. Φ_b.

Just as for the classes of architectural dependability assurance, the monitorability types establish a total order where the most favourable type refers to verifiability and the worst to non-monitorability. Fully monitorability is more advantageous than partially monitorability but not as desirable as verifiability.

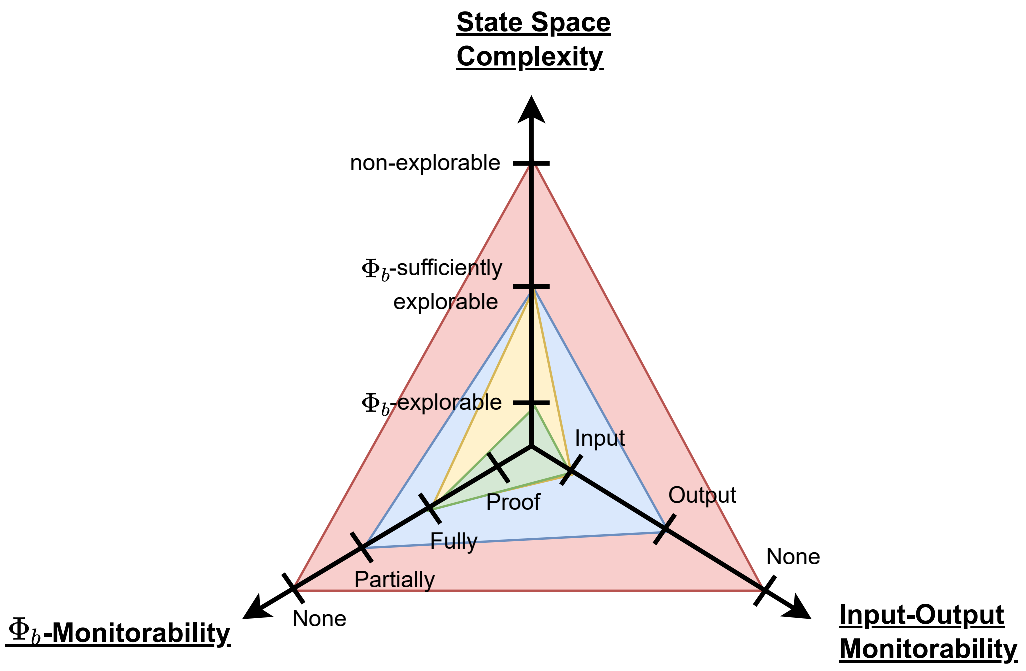


Figure 8.5.: Overview of the analytic capacity with schematically drawn regions indicating distinct analysis potentials.

Input-Output Monitorability The third factor of the analytic capacity is called input-output monitorability. Input-output monitorability complements Φ_b-Monitorability in that it determines whether the properties Φ_b can be derived by solely analysing the input data or by considering the produced output of the AI black-box b. An example of input monitorability is, for instance, variational inference [26] where it is determined if a new input data was produced by the same data generation process as the training data. An example of output monitorability is, e.g., the use of outcome explainers [76] which generate explanations for produced outputs that can be checked against some formal constraints. Hereby, input monitorability is more favourable because it enables the detection of malicious input data before being forwarded to the AI component. That is, in case of detection, appropriate countermeasures can be taken proactively.

Characteristics of the Analytic Capacity After we presented the several factors of the analytic capacity, we now discuss its main characteristics, i.e. the manifestation of the analytic capacity for AI-enabled systems that indicate distinct analytical potentials. Therefore, consider Figure 8.5 which depicts the analytic capacity spanned by its three factors as a radar chart. Depending on how the different factors manifest themselves, the analytic capacity of an AI component takes different forms. However, to provide a better intuition, we schematically draw regions Figure 8.5 to highlight different analytical potentials.

We extend input-output monitorability to include the categorical value “none”. In addition, we have ordered the values of the individual factors (which together span the three-dimensional analytic capacity) according to their natural order, with the most desirable values (e.g. verifiability or Φ_b-explorable) being close to the origin of the radar chart. Therefore, AI components whose analytical capacity spans small regions around the origin are assumed to have a high analytical capacity, e.g. the green region of Figure 8.5. Consequently, AI components that occupy a large area are considered to have a low analytic capacity, e.g. the red region depicted on Figure 8.5.

For example, AI components with high analytic capacity are considered to be at least fully monitorable in terms of Φ_b-monitorability. In addition, the AI component must at least be output-monitorable, i.e. one can determine whether the AI component exhibits erroneous execution at the latest when looking at the outputs. AI-enabled systems that are partially monitorable and at least output-monitorable (e.g. the yellow and blue region of Figure 8.5) exhibit a rather medium-like analytic capacity. AI-enabled systems with low analytic capacity (e.g. the red area of Figure 8.5) are non-monitorable and thus neither input- nor output-monitorable.

Finally, it should be noted that some factors are more relevant to design-time analysis; others are more interesting at runtime. The factors of state space complexity and Φ_b-monitorability, for instance, strongly influence the extent to which an AI-enabled system is analysable at design-time. Intuitively, systems that have small state spaces and are (say) fully monitorable are potentially well suited for design-time analysis. On the contrary, at runtime, the factors of Φ_b-monitorability and input-output monitorability are more important. To make assurances at runtime, one must verify that Φ_b (or Φ_{Sys}, which depends on Φ_b) is satisfied. In this case, it does not matter if we encounter large state spaces, because we only need to check the current system state in terms of Φ_b. That is, we are more interested in having a high Φ_b-monitorability and preferably input-monitorability to act proactively.

8.2.1.4. Fail-Safe

The last dimension concerns whether there is a fail-safe mode the system can transition to. Intuitively, the ability to enter the fail-safe mode neither contributes to nor has any direct analytical implications on whether one can analyse system property Φ_{Sys}. Nonetheless, we consider it as the last dimension which supports software engineers in the classification process. Especially in domains where it is not possible to give reliable assurances about system properties, the availability of a fail-safe mode is always a last resort when situations occur for which no assurances can be given or where it is difficult to predict whether the system might crash or malfunction severely. Therefore, the presence of a fail-safe mode can possibly influence the classification of the system. We will see in section 8.2.2 that the capability to transition to a fail-safe mode affects the decision process regarding the classification of a system to a-posteriori analysability or non-analysability.

8.2.2. Overview of the Classification Structure

After examining the different classes and classification dimensions in the previous section, we now introduce the classification structure to categorise AI-enabled systems. To simplify the structure, we introduce the following convention regarding the analytic capacity: With Φ_b-analysable, we refer to AI-enabled systems that are at least partially monitorable, at least output-monitorable and the state space is at least Φ_b-sufficiently explorable. Note that all criteria must apply to consider an AI-enabled system to be Φ_b-analysable. For example, a system which is fully monitorable, output-monitorable and Φ_b-explorable is Φ_b-analysable. In contrast, a system which is fully monitorable, output-monitorable but the state space is non-explorable, is not Φ_b-analysable.

Consider Figure 8.6 which depicts the classification structure. The entry point of the classification structure is the analytic capacity. More specifically, for an AI-enabled system, it must be determined whether it is Φ_b-analysable. Recall that assurances about a system-level property Φ_{Sys} can only be made whenever any statements about the fulfilment of Φ_b can be given. Thus, an AI-enabled system must be at least Φ_b-analysable. In this case, one is at least able to make statements about Φ_b for a percentage-sufficient subset of the state space. We now follow the branch in which this is not the case (i.e. the AI system or Φ_b is not Φ_b-analysable) and later return to the part where the AI system is Φ_b-analysable.

If an AI system is not Φ_b-analysable, no assurances at design-time can be given; that is, the system cannot be classified into static or monitor analysability. Instead, the Φ_b-monitorability is queried. If the analytic capacity of the AI-enabled system is non-monitorability, the system is classified to be non-analysable because no statement can be given about Φ_b and thus Φ_{Sys}. If the Φ_b-monitorability of the analytic capacity is verifiable, the system is a-posteriori analysable because we can prove that Φ_b holds. Only if the AI component is either partially or fully monitorable, the presence of a fail-safe mode is crucial. If so, one can transition to fail-safe if either Φ_b is not satisfied or there is an inconclusive verdict. Therefore, if the fail-safe mode is available or the system is not safety-critical, the system is classified into a-posteriori analysability. Otherwise, the system is considered to be non-analysable. As already mentioned in section 8.2.1.4, the presence of a fail-safe mode is analytical of no relevance (therefore only marked with dashed lines in Figure 8.6) but crucially affects the classification process regarding the classes a-posteriori analysability and non-analysability. It may seem contradictory that a fully monitorable system can be classified as non-analysable. However, it should be remembered that with full monitorability, although it is possible to determine whether Φ_b is satisfied for any state, there are still possible cases where Φ_b may not be satisfied. For safety-critical systems, it is paramount to transition to fail-safe in such cases; otherwise, there is no way to fail safely.

Now let us return to the case where the analytic capacity is Φ_b-analysable. At this point, the next question that arises is that of abstractability: Are there models that abstract the system, environment and adaptations (in the case of self-adaptive systems) in such a way that we can accurately predict Φ_{Sys}? (Note that at this point Φ_b-analysability already ensures that Φ_b can be checked to some extent which is required to reason about Φ_{Sys}). If

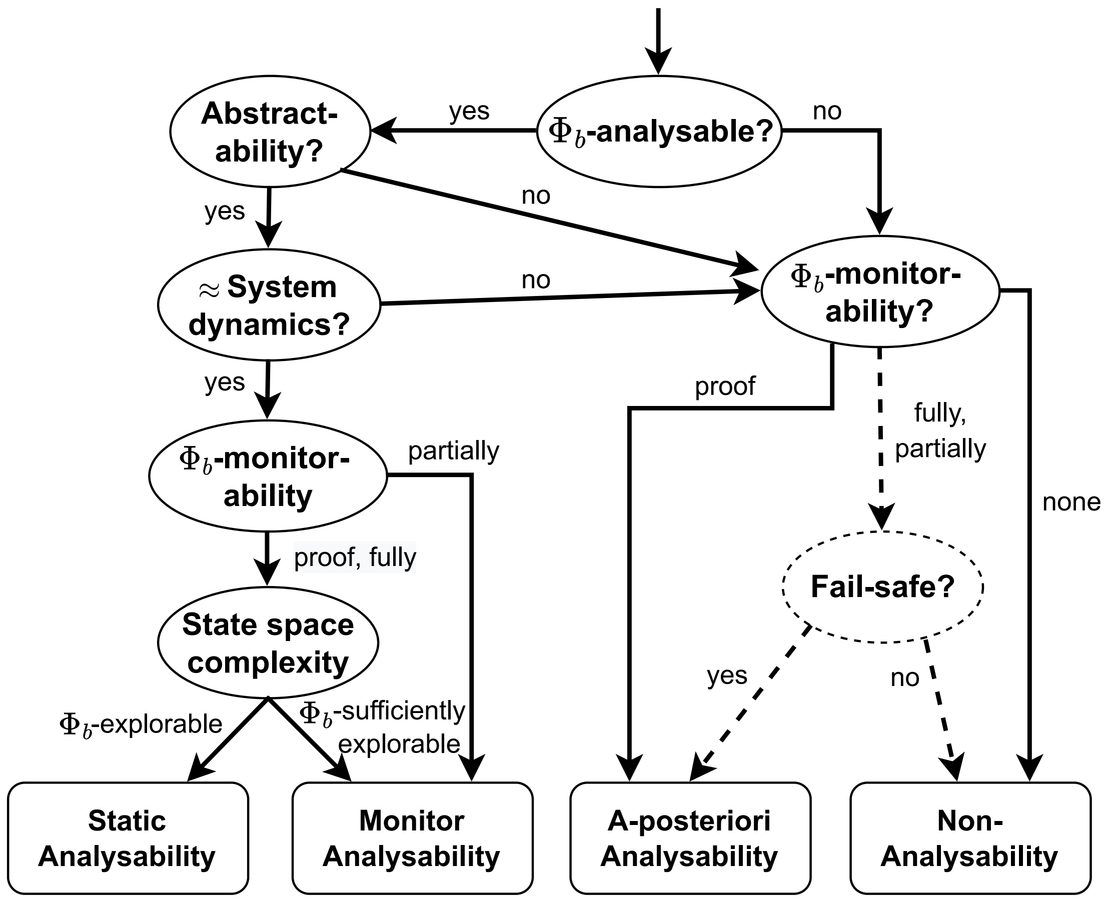


Figure 8.6.: Overview of the classification structure.

so, we can move on to the next dimension. Otherwise, the system cannot be classified into static or monitor analysability but rather in one of the remaining classes.

If abstractability is given, it must be determined whether the system dynamics are sufficiently approximated as it arguably makes no sense to analyse states that (w.r.t. the system dynamics) are never visited (see section 8.2.1.2). If the system dynamics are not accurately approximated, it is not possible to make assurances at design-time.

Finally, if also the system dynamics are sufficiently approximated, it is generally possible to make assurances at design-time. At this point, if the system is partially monitorable in terms of Φ_b-monitorability, it can be directly classified into monitor analysability. If the system is either verifiable or fully monitorable, the state space complexity determines whether the system is statically analysable or monitor analysable. More specifically, in terms of Φ_b-explorable state spaces, we consider the system to be statically analysable because we can explore each state and check whether system-level property Φ_{Sys} is satisfied (w.r.t. Φ_b). In terms of state spaces that are Φ_b-sufficiently explorable, there are still small regions within the state space for which the fulfilment of Φ_{Sys} cannot be checked. For these small regions, we do not know whether Φ_{Sys} is satisfied. Thus, systems are classified into monitor analysability. It is important to note that not knowing whether Φ_{Sys} is satisfied is not directly associated with the inability of monitoring Φ_b in some states (as for partial monitorability) but could also be a result of lacking exploration capabilities of the entire state space, i.e. the state space could be reduced to small regions in which no assurance can be made regarding Φ_{Sys}. However, if the system is either partially monitorable and Φ_b-sufficiently explorable, it must be ensured that the portion of the explorable state space is sufficient w.r.t. the portion of states for which statements regarding Φ_b can be made.

It may seem unusual that for the branch where an AI system is not Φ_b-analysability, the availability of fail-safe mode is explicitly checked, but not if we already know that we can classify into static or monitor analysability, i.e. for the Φ_b-analysability branch. However, in case of static or monitor analysability we can make assurances at design-time; that is, we can determine for which states Φ_{Sys} is not satisfied. Moreover, we can evaluate countermeasures to deal with such states. This is in contrast to the remaining classes, as the knowledge gained at design-time is completely missing and one has to rely exclusively on the mechanism used at runtime.

8.2.3. Deriving Dependability Assurance Cases

After we introduced the classification structure with its distinct dimensions, we now envision or outline how the classification structure can be used to generate dependability assurance cases. Generally, assurance cases (such as safety assurance cases) are structured by common notations, such as the *Goal Structuring Notation* (GSN) [68] or *Claims-Argument-Evidence* notation [27]. In this section, we discuss how dependability assurance cases can be derived after a system and its environment have been classified according to our classification structure. We also outline what such a dependability assurance case might look like, using GSN as a prevalent notation for assurance cases.

As we have seen in section 8.2.2 the classification structure and its dimensions exhibit a high degree of abstraction. This is mainly because we aim to classify as many AI-enabled systems as possible, each of which has its particularities. Consequently, the dimensions are difficult to assess objectively; for example, the dimensions abstractability, approximation of system dynamics or Φ_b-analysability (or the analytic capacity in general) are arguably difficult to assess in terms of their fulfilment. Moreover, there are probably no universal metrics that allow determining whether the dimensions are sufficiently satisfied (as it is also a domain-specific matter). Instead, software engineers need to find (domain-specific) evidence which provides adequate justification. However, in developing assurance cases, claims are made about the system (e.g. properties that the system exhibits or countermeasures implemented to mitigate hazardous behaviour) that are substantiated by evidence. Thus, the question arises whether one can derive a dependability assurance case based on the arguments and evidence gained during the classification process.

In the following, we illustrate how our classification structure can be used as guidance or a blueprint to generate dependability assurance cases. For this purpose, we consider GSN as a notation to structure the assurance case. Figure 8.7 illustrates a simplified GSN-based example of the dependability assurance case of an AI-enabled system which we assume

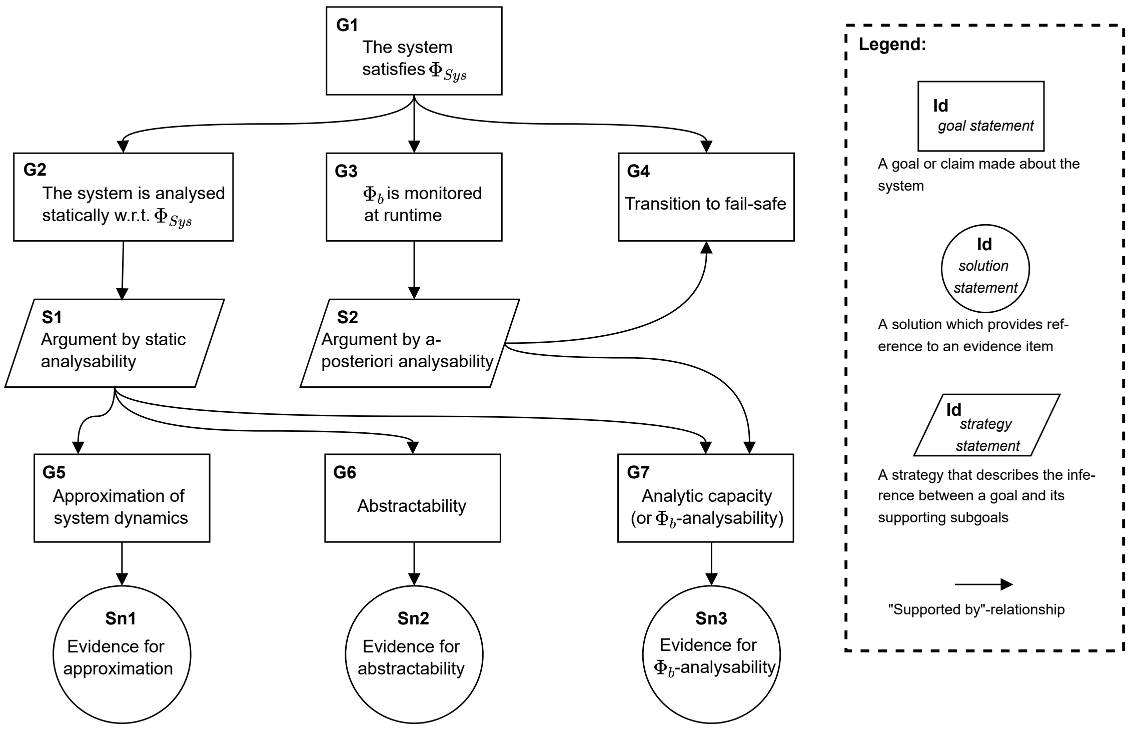

G1
The system satisfies Φ_{Sys}
G2
The system is analysed statically w.r.t. Φ_{Sys}
G3
Φ_b is monitored at runtime
G4
Transition to fail-safe
S1
Argument by static analysability
S2
Argument by a-posteriori analysability
G5
Approximation of system dynamics
G6
Abstractability
G7
Analytic capacity (or Φ_b-analysability)
Sn1
Evidence for approximation
Sn2
Evidence for abstractability
Sn3
Evidence for Φ_b-analysability
Legend:
Id
goal statement
A goal or claim made about the system
Id
solution statement
A solution which provides reference to an evidence item
Id
strategy statement
A strategy that describes the inference between a goal and its supporting subgoals
"Supported by"-relationship

Figure 8.7.: Illustration of a dependability assurance case based on the goal structuring notation [68].

to be statically analysable. In principle, a GSN is a directed acyclic graph that consists of *Goals* (i.e. top-level claims made about the system), *Sub-goals* (i.e. claims that together refine a top-level claim) and *Solutions* (i.e. the evidence that support the made claims). Moreover, a *Strategy* is a multi-argument approach to support a top-level goal (or claim). In our case, we have created a GSN-based assurance case where the top-level goal is to satisfy the system property Φ_{Sys}. For example, a system-level property could comprise the success probability $P(X_{Sys} \mid X_U)$ (e.g. $P(X_{Sys} \mid X_U) > \epsilon \in [0, 1]$) that the system must satisfy.

The top-level goal **G1** is supported by three sub-goals. Note that the "supported by" relationship implies that all sub-objectives are supported by a sufficient amount of evidence, and does not mean that only a subset of sub-objectives must be assured. The first sub-goal **G2** claims that the system is statically analysable (which results from the classification process). The argument strategy encompasses in this case the same line of reasoning as when the AI-enabled system is classified as static analysability. That is, the strategy is supported by three sub-goals (i.e. **G5**, **G6** and **G7**), each of which claims the fulfilment of the respective dimension that, taken as a whole, justifies static analysability. Finally, each sub-goal is associated with the respective evidence gathered during the classification process. Note that the sub-graph starting at **S1** could have been also replaced by a single solution denoted “Reliability analysis”. However, when constructing the assurance case based on the line of argumentation provided by the classification structure, the assurance case becomes in its entirety more compelling and is substantiated by a larger body of evidence.

Besides static analysability, we can support the top-level goal **G1** by further sub-goals. Therefore, recall that our dependability assurance classes form a hierarchy. Because we assumed the system under consideration to be statically analysable, we are also able to make assurances at runtime. If we further assume the system to be fully monitorable in terms of Φ_b, we apply runtime monitoring to check whether Φ_b is satisfied; this constitutes the second sub-goal **G3**. Again we can reuse the same line of reasoning and refer to the respective evidence that justifies a-posteriori analysability. The dependability assurance case is completed by sub-goal **G4** which claims that the system can transition to fail-safe (assuming that there exists a fail-safe mode).

In summary, the simplified dependability assurance case is derived for a statically analysable AI-enabled system. The system-level property Φ_{Sys} is supported by the claim that the system is statically analysable at design-time in terms of Φ_{Sys}. Thus, software engineers can evaluate design decisions such as architectural patterns (e.g. N-version programming pattern) that address Φ_{Sys}. Since there are still situations where Φ_{Sys} is not satisfied, runtime monitors (e.g. by using neuron activation pattern monitors [46] or outcome explainers [76]) are employed to detect these states and transitions to fail-safe if necessary. Finally, we would like to emphasise that the outlined process serves only as an inspiration for how the classification structure might be used to guide software engineers through the development process of dependability assurance cases.

8.3. Classifying AI-enabled Systems

In this section, we apply our classification structure to AI-enabled systems from the literature. However, the classification of an AI-enabled system w.r.t. our classification structure is highly subjective. Thus, a comprehensive evaluation was not possible because it would require the knowledge of domain experts to assess the individual classification dimensions. Therefore, we focused on the applicability of our classification structure. More specifically, we apply our classification structure to three representative domains where AI have been commonly used, namely AI-supported assistance in automated driving, human-robot-interaction systems and aircraft collision avoidance systems. Hereby, we consider various system-level properties. Due to the limited number of design-time approaches for analysing AI-enabled systems, we apply our reliability prediction approach (if possible) for each considered system and discuss how they are classified when considering $P(X_{Sys} \mid X_U)$ as additional system-level property. It is important to note here that the way we apply the classification structure is only to show general applicability and is based only on the information provided in the papers. We are no experts in any of the domains discussed; that is, a domain expert might classify the system differently based on domain-specific knowledge that we are not aware of. As a starting point, however, we demonstrate the applicability of our classification structure by classifying a representative set of AI-enabled systems.

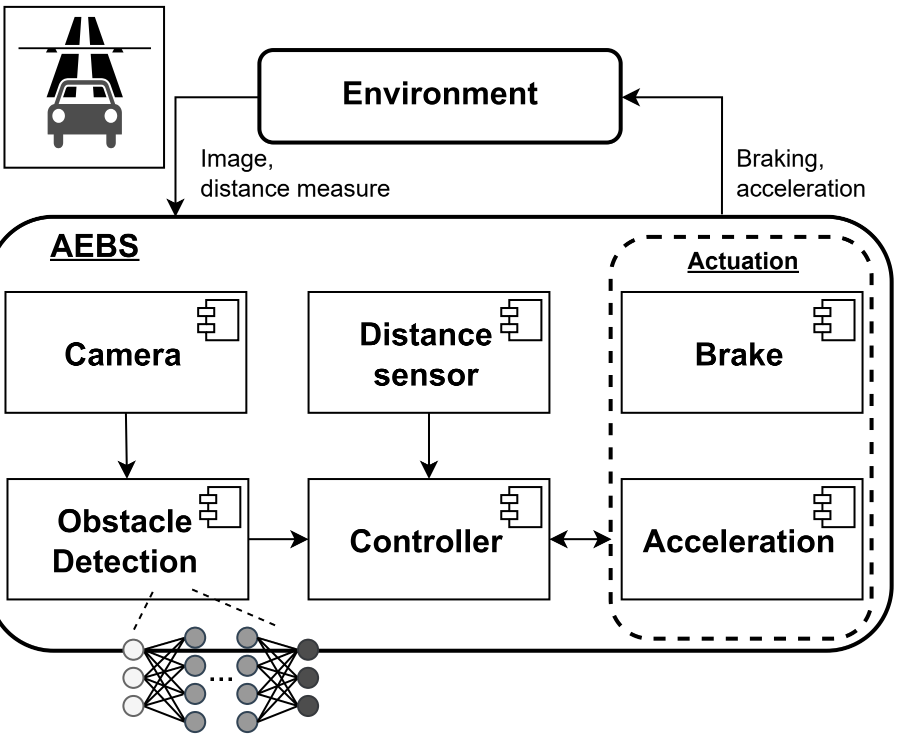


Figure 8.8.: Overview of the AEBS based on [54].

8.3.1. AI-supported Assistance in Automated Driving

In [93], a taxonomy for automated driving vehicles is presented, which includes six levels of automation, ranging from level 0 (no automation of driving) to (e.g.) level 2 (partial automation of driving) to level 5 (full automation of driving). At each level, AI can be used to assist during driving (e.g. object detection in automatic braking systems). The higher the level, the more challenging the learning task. Therefore, AI-enabled systems that are categorised in higher levels of automation are likely to be classified as lower-order dependability assurance classes. In the following, we discuss two AI systems. The first system corresponds to an *Automatic Emergency Braking System* (AEBS), which relies on AI-based object recognition to detect vehicles ahead and actuates the brakes (if necessary) to avoid a collision. The second system corresponds to a more advanced autonomous driving system, i.e. perception systems.

8.3.1.1. Automatic Emergency Braking System

As a representative example for the discussion, we consider an AEBS presented in [54]. The AEBS comprises several components that are depicted on Figure 8.8.

The first component of the *Cyber-Physical System* (CPS) refers to a controller responsible for regulating acceleration and braking. For the regulation, the controller makes use of the second component: the plant (vehicles subsystem under control). Moreover, the controller component makes its decisions based on a sensor (more precisely a camera) which is equipped with a DNN-based obstacle detector. The AEBS and its operating environment form a closed-loop control system. The controller component (which regulates the braking and acceleration control signals) relies on the accuracy of the obstacle detector. Thus, it is paramount to make assurances regarding safety-relevant system properties.

Originally, Dreossi et al. presented in their work [54] a compositional falsification framework, where they identify counterexamples (by considering misclassifications of the DNN) for which the AEBS exhibits erroneous execution. However, the classification structure is designed more to assess the degree of dependability assurance for a system property of an AI system. In the absence of other approaches to evaluating AI-enabled systems, we nevertheless consider the falsification approach. As we will see later, this approach provides the possibility of making assurances anyway.

After we classified the compositional falsification approach, we discuss how our approach for reliability prediction can be applied to analyse the system-level property $P(X_{Sys} \mid X_U)$.

Compositional falsification of CPS with machine learning components In the following, we apply our classification structure to the compositional falsification approach of Dreossi et al. [54]. As a system-level property, it is required that the AEBS avoids collisions, i.e. the system must maintain a certain distance $dist(t)$ (relative to a certain distance limit τ) from an obstacle at any time t: $dist(t) \geq \tau$.

In summary, the approach presented by Dreossi et al. involves two main analysis components (namely a CPS analyser and an ML analyser) which are composed for the detection of counterexamples. The CPS analyser acts as the primary component which checks whether the AEBS violates the system property Φ_{Sys}. Hereby, the input space of the AEBS is considered of three variables, namely the *Distance* between the vehicle and the preceding obstacle, the *Velocity* of the vehicle and the *Input Images* of the camera (forwarded to the DNN for classification). The CPS analyser abstracts away the DNN b by considering two extremes: A perfect image classifier (i.e. producing always the correct output; denoted as b^+) and the worst possible image classifier (i.e. producing always the wrong output; denoted as b^-). Internally, the CPS analyser is implemented by using a simulation model (in this case a Simulink model) to simulate the AEBS and a verification tool (namely Breach [53]) to falsify the Φ_{Sys} at system-level. Hereby, system property Φ_{Sys} is falsified by considering two cases where the image classifier is either perfect (i.e. b^+) or operates poorly (i.e. b^-). For both cases, the input space can be partitioned into regions where Φ_{Sys} is violated or satisfied. When the partitioned input space of both cases is combined, the input space can be reduced to an overlapping region where the fulfilment of Φ_{Sys} solely depends on the prediction correctness of b; this region is called *Region of Uncertainty* (ROU). The task of the ML analyser is now to determine the images within the ROU for which the DNN makes incorrect predictions and based on which counterexamples are identified. The DNN of the AEBS inputs images which requires analysis of the pixel space. However, the pixel space is too large to be analysed; instead, the ML analyser makes use of feature space abstraction in which the original DNN $b : \mathcal{X} \rightarrow \mathcal{Y}$ is approximated by $\tilde{b} : A \rightarrow \mathcal{Y}$. The approximated version $\tilde{b}$ acts on an abstraction (called abstract domain A) of the original input space (or feature space) $\mathcal{X}$. The abstraction is achieved by focusing only on a constrained feature space $\tilde{\mathcal{X}} \subseteq \mathcal{X}$ restricted to the scenario under investigation (in this case desert road scenarios with a single car on the highway) and three dimensions along which the scene can be varied, namely the lateral position of the car, the distance from the sensor position of the vehicle, and the brightness of the image. Within the three dimensions, the scene can be varied and analysed regarding misclassifications. More specifically, the approximated classifier $\tilde{b}$ allows analysis of regions within the abstract domain A for which the original classifier b potentially produces misclassifications. These regions are used to identify counterexamples within the ROU. To connect A with $\mathcal{X}$, a so-called abstraction function $\alpha : \tilde{\mathcal{X}} \rightarrow A$ and concretization function $\gamma : A \rightarrow \tilde{\mathcal{X}}$ are used. The exact details of the ML analyzer would go beyond the scope of this section; thus, we refer to the original work of Dreossi et al. [54] to look up the details of the ML analyser. In combination, the CPS and ML analyser allows determining counterexamples for which the system-level property Φ_{Sys} is violated.

The classification result is depicted on Table 8.1. In terms of the analytic capacity, we consider the AEBS to be Φ_b-analysable. The AI-specific properties Φ_b refer to input space regions for which the DNN potentially produces misclassifications w.r.t. the distance and lateral position to the next obstacle as well as the brightness of an image. Based on the factors, one can determine (w.r.t. the approximated classifier $\tilde{b}$) whether b is likely to produce wrong predictions. Assuming that all factors are derivable from the sensor inputs, the AI component is fully monitorable regarding Φ_b (i.e. lateral position, distance and brightness). Moreover, from the information provided by the authors, we consider the state space complexity to be Φ_b-sufficiently explorable. This results mainly from the fact that the input space of the AEBS is already highly reduced to the ROU. Although the ROU is unlikely to be fully explored in terms of misclassifications gained by analysing $\tilde{b}$, the ROU can still be refined based on identified misclassification clusters. So while a large part of the state space has already been analysed as non-safety critical, other (much smaller) parts are known to be potentially unsafe. For these regions, system-level countermeasures or additional AI-specific monitors can be used to safeguard the DNN image classifier. Based on the results provided by Dreossi et al., we argue that the AEBS is Φ_b-sufficiently explorable. Moreover, due to the state space analysis, Φ_b is input monitorable, i.e. we could already determine the critical subspaces of the state space.

Abstractability is given by using a Simulink model (for physical simulation of the AEBS) and Breach [53] (a verification tool used to falsify Φ_{Sys}). The concrete interplay between the Simulink model and Breach is not discussed; however, the authors pointed out that simulation-based verification is well-studied in the literature. Therefore, we consider abstractability to be sufficiently addressed (also taking into account the evaluation results of the approach). The same applies to the approximation of the system dynamics classification dimension. Since the simulation-based verification of CPS is a well-researched field, we assume the system dynamics to be sufficiently approximated. Moreover, the AEBS is a static software system for which the approximation of the system dynamics is not as important as for self-adaptive systems (see section 8.2.1.2).

Analytic Capacity	
Φ_b-monitorability	**Explanation**
Fully monitorable	Approximation of b and feature space abstraction.
State space complexity	
Φ_b-sufficiently explorable	CPS input space and ML feature space abstraction; reduction of CPS input space to ROU.
Input-output-monitor.	
Input-monitorable	Input space analysis of $\tilde{b}$.
	Φ_b-analysable: Yes
Abstractability	
Conclusion	**Explanation**
Yes	CPS-analyser: Simulink model of the AEBS and the verification tool Breach [53]. ML-analyser: Approximation of b by abstract $\tilde{b} : A \rightarrow \mathcal{Y}$.
$\approx$ System dynamics	
Conclusion	**Explanation**
Yes	Simulation-based verification of CPS is a well-studied field. The software system is static.
Fail-safe	
Conclusion	**Explanation**
-	Irrelevant as Φ_b is Φ_b-analysable.
Class: Monitor analysability	

Table 8.1.: Classification result of the AEBS by considering the compositional falsification approach of Dreossi et al. [54].

Overall, we classify the AEBS into monitor analysability. This is mainly because the system is only Φ_b-sufficiently explorable, i.e. there exist states for which no conclusion can be made regarding system property Φ_{Sys}.

Reliability prediction Now, we discuss how the AEBS is classified when applying our approach to reliability prediction, i.e. we want to assure the system's probability of success as a system-level property. At this point, we assume that the AEBS is modelled with PCM. However, the core of the reliability prediction approach is the sensitivity model which represents the AI component (or in this case the DNN for image classification). Originally, Dreossi et al. discussed three factors in how an image or scene is varied, namely brightness, distance and lateral position. Based on the factors misclassification ranges were determined. In our approach, we consider the factors as sources of uncertainty that could force the AI component to make incorrect predictions. Therefore, we view the sensitivity model with three variables (one for each factor) and one variable capturing the failure/success probability (i.e. the predictive uncertainty) of the AI component. Dreossi et al. presented a scene generator in their compositional falsification framework that concretises images based on samples of the abstract domain (i.e. images with distinct image brightness, distance of the nearest vehicle and lateral position). We argue that the generator can be used to produce a dataset from which the sensitivity model can be derived (e.g. by considering one of the sensitivity analysis approaches from section 7.1.2). Moreover, we assume a discretisation of the value spaces for each factor. For example, image brightness is discretised in categories (e.g. high, low, normal) or distance values are discretised into intervals.

Based on these considerations, we can now apply our classification structure. Therefore, consider Table 8.2 which summarises the classification results. We consider the analytic capacity to be Φ_b-analysable. In this case, the AI-specific property Φ_b refers to the success/-failure probability of the AI component w.r.t. the aforementioned factors (i.e. brightness, distance and lateral position). We described how the factors could be discretised into categorical values. The Cartesian product spans the space to be explored. However, due to the discretisation (and the fact that we form the Cartesian product of only three variables), we argue that the space is Φ_b-explorable. Because brightness, distance and lateral position are observable at runtime, we consider the AEBS to be fully monitorable. In terms of input-output monitorability, the same reasoning applies as for the compositional falsification approach, i.e. Φ_b is input-monitorable because we can observe brightness, distance and lateral position and use the sensitivity model to check whether we might encounter an unsafe state.

Abstractability is given under the assumption that the AEBS can be modelled accurately with PCM and that we can generate a sensitivity model from the DNN. In terms of approximating the system dynamics, we argue (again) that since we analyse a static software system, the system dynamics are negligible. More precisely, we analyse how the probability of success for a given system configuration varies by considering different combinations of the three factors. However, this is done in a brute-force manner that does not take into account the temporal evolution of the system.

Overall, we conclude that the AEBS is statically analysable regarding system-level property $P(X_{Sys} \mid X_U)$.

8.3.1.2. Autonomous Driving

In this section, we move further up the taxonomy of automation and look at the higher levels of automation for automated driving vehicles. More specifically, we consider AI components that are part of the entire cognition process of the automated vehicle. First, we classify an uncertainty estimation method for software architecture of autonomous driving vehicles provided by Serban, Poll and Visser [163] and our reliability approach afterwards.

Uncertainty estimation of software architectures for autonomous driving vehicles Serban et al. [163] applied their uncertainty estimation method to a perception system for scene understanding of an autonomously driving system. The perception system comprises three

Analytic Capacity	
Φ_b-monitorability	**Explanation**
Fully monitorable	Brightness, distance and lateral position are observable.
State space complexity	
Φ_b-explorable	All uncertainty combinations can be iterated.
Input-output-monitor.	
Input-monitorable	Due to sensitivity analysis.
	Φ_b-analysable: Yes
Abstractability	
Conclusion	**Explanation**
Yes	PCM to abstract the software architecture. Sensitivity model to abstract the DNN. Markov chain transformation to predict $P(X_{Sys} \mid X_U)$.
$\approx$ System dynamics	
Conclusion	**Explanation**
Yes	The software system is static.
Fail-safe	
Conclusion	**Explanation**
-	Irrelevant as Φ_b is Φ_b-analysable.
Class: Static analysability	

Table 8.2.: Classification result of the AEBS by considering our reliability prediction approach.

components, namely object detection (to identify the location of all obstacles in an image), semantic segmentation (which associates each pixel in an image to a class) and depth estimation (to determine the position of obstacles or the road surface). Each component is implemented by using a dedicated DNN. The result of the perception system is forwarded to a planning component for trajectory computation. Consequently, the prediction accuracy highly affects the safety of the system as incorrect/inaccurate predictions may lead to wrongly computed trajectories.

We already discussed the approach in related work (see section 3.1.2.2). In a nutshell, the approach annotates software components of an architecture (e.g. provided by an architecture model) by the two ML-specific uncertainty types: epistemic and stochastic (or aleatoric) uncertainty. Based on the annotated components, a BN (Bayesian network) is generated in which the annotated components as well as their annotated uncertainties are represented as nodes (i.e. random variables). The graph structure of the BN describes a kind of control flow along which uncertainty could be potentially propagated. The probability distributions associated with each node (or random variable) of the BN can be determined by a domain expert or simulation. In the case of the perception system, the last node of the BN refers to the random variable describing the planning component of the system (subsequently denoted as X_{Plan}) which is highly affected by the predictions of the

Analytic Capacity	
Φ_b-monitorability	**Explanation**
Fully monitorable	By assumption.
State space complexity	
Φ_b-explorable	Due to Bayesian inference rules.
Input-output-monitor.	
Output-monitorable	At least output-monitorable.
Φ_b-analysable: Yes	

Abstractability	
Conclusion	**Explanation**
Yes	Software components and their connections are abstracted by BNs.

≈ System dynamics	
Conclusion	**Explanation**
-	Irrelevant.

Fail-safe	
Conclusion	**Explanation**
-	Irrelevant as Φ_b is Φ_b-analysable.

Class: Static analysability

Table 8.3.: Classification result of a perception system of a self-driving vehicle by considering the uncertainty estimation approach of Serban et al. [163].

individual DNNs. Based on the structure and defined distributions of the BN, inference rules of BNs are applied to reason about properties such as $Pr(X_{Plan} = high)$, $high \in Val(X_{Plan})$. Therefore, we consider $P(X_{Plan})$ as the system-level property. Moreover, by using Bayesian inference rules, one can determine how $Pr(X_{Plan} = high)$ is affected when varying the uncertainty values of components of the perception system.

Table 8.3 summarises our classification results. The AI-specific properties Φ_b refer to the epistemic and aleatoric uncertainty of each AI component. Based on the information provided by the authors, we could not reliably assess whether the methods used to analyse the uncertainty types are also suitable to be applied at runtime (predictions are assessed whether they exhibit epistemic or aleatoric uncertainty). To determine the class of Φ_b-monitorability, more knowledge of the methods is required. For example, Phan et al. [139] describe an approach where epistemic or aleatoric uncertainties are estimated as part of the prediction. However, they focused on a specific class of DNNs (Bayesian deep learning models) such that we cannot use the approach for runtime estimation. Nonetheless, Serban et al. pointed out that for both uncertainty types: "The methods used to measure them can be different, depending on the ML algorithm employed" [163]. This suggests that there are several methods, one can take into consideration when estimating the uncertainties. Therefore, we assume that the properties are fully monitorable. Moreover, if they are fully monitorable, they must be at least output-monitorable. In their estimation approach, the authors showed how to apply inference rules of BNs to check the fulfilment of system property $P(X_{Plan})$ when varying the uncertainty values of components of the perception system. Since this can be done with reasonable effort for all the different types of uncertainties that the components of the perceptual system may encounter, we consider the state space to be Φ_b-explorable. Thus, we evaluate the analytic capacity to be Φ_b-analysable.

Abstractability is given in that BNs are used to model the software architecture or component structure of the perception system. The approximation of the system dynamics is not relevant as we are (again) assessing a static software system; but most importantly, the uncertainty estimation method is to be considered as an inter-component analysis that aims to assess how uncertainty might propagate and does not incorporate system dynamics.

Overall, we classify the perceptual system as statically analysable w.r.t. assuring the system property $P(X_{plan})$. However, it should be noted that uncertainty estimation is highly dependent on how well the epistemic and aleatoric uncertainty can be estimated from the AI components. Thus, it is paramount that the estimation results are rigorously checked by domain experts.

Reliability prediction In this section, we again apply our reliability prediction approach to make assurances about the system-level property $P(X_{Sys} \mid X_U)$. However, we do not consider the perception system we classified earlier. Instead, we consider a more advanced AI system whose capabilities go beyond mere perception and which also includes (partial) planning tasks. More specifically, we consider an AI system that predicts the steering angles of a self-driving car based on image data. Several variants of such systems are presented by Tian et al. [186]. Since the steering angle prediction strongly influences the movement of the vehicle, it is of paramount importance that the AI components operate accurately. The reason we consider a different system is to provide and discuss an example of a non-analysable system in terms of Φ_{Sys} (or $P(X_{Sys} \mid X_U)$).

Recall that the AI-specific property Φ_b refers to the distribution $P(X_b \mid X_{\varphi_1}, \dots, X_{\varphi_N})$ which is conditioned on the uncertainty factors $\varphi_1, \dots, \varphi_N$. Let us assume that the factors refer to events in the environment which potentially affect the prediction result. However, the set of possible events is potentially quite large and difficult to fully determine at design-time. In [139, 186] alone, enumerates together 14 factors, namely brightness variation, changing contrast, translation, scaling, horizontal, shearing, rotation, blurring, fog effect, rain effect, depth, occlusion, clouds, and puddles. Due to the highly dynamic environment of autonomously driving cars, it is unlikely that the list is complete. Even if we assume that each factor can be discretised into binary values, we would encounter $2^{14} = 16384$ entries of the probability mass function $P(X_b \mid X_{\varphi_1}, \dots, X_{\varphi_N})$. Apart from the fact that such a high-dimensional probability distribution cannot be modelled manually, it is not only difficult to estimate such a distribution (as a large data set is required to build the sensitivity model) but also entails dimensionality issues (i.e. the curse of dimensionality).

Analytic Capacity	
Φ_b-monitorability	**Explanation**
Non-monitorable	Due to dimensionality problems and the inability to observe certain properties.
State space complexity	
Φ_b-explorable	By assumption.
Input-output-monitor.	
None	As a consequence of non-monitorability.
Φ_b-analysable: No	
Abstractability	
Conclusion	**Explanation**
-	Irrelevant as Φ_b is not Φ_b-analysable.
$\approx$ System dynamics	
Conclusion	**Explanation**
-	Irrelevant as Φ_b is not Φ_b-analysable.
Fail-safe	
Conclusion	**Explanation**
-	Irrelevant as Φ_b is non-monitorable.
Class: Non-analysability	

Table 8.4.: Classification result of an AI-based steering angle prediction system of an autonomously driving vehicle by considering our reliability prediction approach.

Our classification result is summarised in Table 8.4. Although the system itself could be modelled by using PCM (i.e. abstractability and approximation of system dynamics are possibly satisfied), the system is not Φ_b-analysable which results mainly from the dimensionality problems discussed above. However, the most important aspect is that most of the uncertainty factors $\varphi_1, \ldots, \varphi_N$ are not monitorable at runtime. For instance, how is one able to measure properties like translation, scaling, shearing or rotation? This is generally not possible such that we have to consider Φ_b to be non-monitorable. Moreover, non-monitorability directly classifies systems into non-analysability.

Note that the example discussed is rather artificially constructed and is unlikely to be encountered in practice. Nevertheless, it conveys an intuition about systems that are non-analysable. Even if a system is non-analysable, this simply means that no assurances can be made about the system-level property under consideration. In this case, however, one can instead focus on a set of system properties that are more eligible in terms of making assurances and that, in combination, provide a strong assurance case. Furthermore, non-analysability should not prevent software engineers from using testing approaches such as [186].

Finally, as a last remark, the example reveals an interesting corner case. We explained that the abstractability and approximation of the system dynamics are not relevant because Φ_b is not Φ_b-analysability. However, let us ignore for a moment the dimension of analytical capacity and follow the path in the classification structure when a system is Φ_b-analysable. As already noted, the system could be modelled using PCM and the static nature allows the neglect of the system dynamics. Even though there are a minimum of $2^{14} = 16384$ uncertainty combinations that need to be evaluated to determine $P(X_{Sys} \mid X_U)$, this is still manageable or can be done efficiently with our reliability prediction approach. Thus, we assume the state space complexity Φ_b-explorable because we can at least statistically sufficiently infer $P(X_{Sys} \mid X_U)$. In summary, the system could be classified into static analysability but is deemed to be non-analysable due to the inability to monitor $\varphi_1, \ldots, \varphi_N$ at runtime. In the testing approach of Tian et al. [186], for example, synthetic data is used to generate images that indicate exactly those uncertainty factors. Thus, we can generate a dataset in which each generated image can be labelled by the corresponding factor inserted in the image. However, this allows us to determine the sensitivity model at design-time and also to analyse the system regarding $P(X_{Sys} \mid X_U)$ (ignoring the problems discussed earlier). Therefore, it may seem unreasonable to classify a system into non-analysablility even though assurances can be given at design-time. However, we argue that a fully assured system requires assurances at design-time and runtime; unless one can prove that Φ_{Sys} is satisfied at design-time (e.g. Julian and Kochenderfer [95]) or at runtime (e.g. Thumm and Althoff [185]). One of the main advantages of static (or monitor) analysability is that it not only allows us to design our system (e.g. by evaluating design decisions) but also guides us during the design of the system. More specifically, if we have a statically (or monitor) analysable system, we can determine the set of potentially unsafe states. This knowledge gained should not only be used to design the system accordingly but should also be reused when monitoring the AI component. Having a system in a safety-critical context for which assurances can be given merely at design-time (assuming the evidence include no proof) but being incapable of making statements regarding Φ_{Sys} and Φ_b at runtime cannot be considered to be sufficiently assured.

8.3.2. Human-Robot-Interaction Systems

In this section, we classify two more systems, namely a robotic manipulation system in human environments and the HRI system we described in section 1.5.3.

Robotic manipulation in human environments Now we classify an approach for provably safe deep RL (reinforcement learning) for robotic manipulation by Thumm and Althoff [185]. In their work, the authors describe a shielding mechanism that ensures human safety during the manipulation task of an RL-based controller. The considered scenario involves a modular robot with six degrees of freedom which is mounted on a working table. Because a human may work at the same table, it must be assured that the robot operates safely, i.e. there must be no trajectories where the robot collides with the human. Although not explicitly mentioned by the authors, we assume the state of the robot to be represented by a vector. Each element of the vector corresponds to an observation including the current joint position, velocity, episode goal, Cartesian end-effector position and the relative Cartesian positions of the human wrists and head.

In essence, the safety shield presented by the authors consists of two planners. The first planner refers to a long-term planner acting on a low frequency; the second planner refers to a fail-safe planner acting on a high frequency. Roughly summarised, the idea is to use the long-term planner for computing an *Intermediate Trajectory* in larger time steps ΔT (i.e. at lower frequency) which is complemented by the fail-safe planner which calculates safe sub-trajectories within smaller time steps Δt (i.e. at higher frequency) with $\Delta t < \Delta T$. Note that the original intermediate trajectory could be getting unsafe whenever a dynamically moving object interferes with the computed trajectory. Therefore, the fail-safe planner must verify potential collision at high frequency. In principle, the approach works as follows: During the execution of a trajectory between t_i and t_{i+1} (where $\Delta t = t_{i+1} - t_i$) a fail-safe trajectory is computed starting from $t + 2$. If there is no fail-safe trajectory (as a result of the verification process), the fails-safe trajectory of $t + 1$ is executed. In addition, it is assumed that the robot starts in a safe state at t_0. Thus, by induction safety can be guaranteed for any time horizon.

Informally, we define the system-level property "The robot is guaranteed to cause no collision". Moreover, we consider as AI-specific property whether there exists an action for any time t_i such that a fails-safe trajectory (starting at t_i) can be constructed. The classification results are summarised in Table 8.5. We consider the system to be not Φ_b-analysable because the state space complexity is non-explorable. This is mainly because the state and action space are both continuous spaces which makes it practically impossible to be at least sufficiently explorable. Moreover, we consider input-monitorability because it can be determined prior to t_{i+1} whether the trajectory starting at t_i is safe.

Finally, the fact that the system is fully monitorable and is guaranteed to transition to fail-safe allows the classification into a-posteriori analysability.

Reliability prediction of HRI system In this section, we discuss the HRI example system from section 1.5.3. Recall that in the HRI example system, a robotic arm is considered which supports humans in the assembly tasks of some parts. Since the robot implements an AI-based object detection component, the safety of the human (with whom the robot collaborates) highly depends on the detection accuracy. Incorrect detection (e.g. the hand of the human worker) can lead to collisions and crushing injuries. However, in this setting, it is known that variations in image brightness and sensor noise may potentially lead to incorrect predictions. For the HRI system, the system-level property Φ_{Sys} refers to $P(X_{Sys} \mid X_U)$; the AI-specific property Φ_b refers to $P(X_b \mid X_{\varphi_B}, X_{\varphi_{SN}})$ where φ_B and φ_{SN} describes the brightness and sensor noise uncertainties.

Our classification result is summarised in Table 8.6. The result is fairly similar to the classification result of the AEBS when considering our reliability prediction approach. We analyse the HRI system as part of our validation (albeit for self-adaptive systems); thus, more details about the reliability prediction or used models can be found in section 9.4.2. The uncertainty factors are fully observable in the environment. Moreover, they can be discretised into finite sets of categorical values that are fully iterable (when considering the Cartesian product). Again, the sensitivity analysis allows for determining whether

Analytic Capacity	
Φ_b-monitorability	**Explanation**
Fully monitorable	Due to the fail-safe planner.
State space complexity	
Non-explorable	Continuous state and action space.
Input-output-monitor.	
Input-monitorable	Due to the fail-safe planner.
Φ_b-analysable: No	
Abstractability	
Conclusion	**Explanation**
-	Irrelevant as Φ_b is not Φ_b-analysable.
$\approx$ System dynamics	
Conclusion	**Explanation**
-	Irrelevant as Φ_b is not Φ_b-analysable.
Fail-safe	
Conclusion	**Explanation**
Yes	Due to the fail-safe planner. A full stop of the robot is possible.
Class: A-posteriori analysability	

Table 8.5.: Classification result of a robotic system for manipulation tasks in human environments of Thumm and Althoff [185].

input images are potentially malicious. Thus, we conclude that overall the system is Φ_b-analysable.

As we will see in section 9.4.2, the HRI system can be abstracted by using PCM models; also, the AI component can be represented by estimating a sensitivity model. The system dynamics are not relevant because we are analysing a static system in which each uncertainty combination is visited and analysed.

Therefore, we classify the HRI system in class static analysability. In section 9.4.2, we evaluate self-adaptive systems that are used to safeguard the AI-based object detection component of the HRI system. The results of the evaluation indicate that each adaptation strategy converges towards a fixed reward value. Thus, it can be concluded that the state space is sufficiently explored. If we now assume that the system dynamics are accurately approximated, the HRI system would also have been classified in static analysability when self-adaptive systems are evaluated in terms of $P(X_{Sys} \mid X_U)$.

8.3.3. Aircraft Collision Avoidance Systems

In this section, we discuss the classification of a safety-guaranteeing approach for DNN-based *Aircraft Collision Avoidance System* (ACAS) of Julian and Kochenderfer [95]. In their

Analytic Capacity	
Φ_b-monitorability	**Explanation**
Fully monitorable	Image brightness and sensor noise are observable.
State space complexity	
Φ_b-explorable	All uncertainty combinations can be iterated.
Input-output-monitor.	
Input-monitorable	Due to sensitivity analysis.
Φ_b-analysable: Yes	
Abstractability	
Conclusion	**Explanation**
Yes	PCM to abstract the software architecture. Sensitivity model to abstract the DNN. Markov chain transformation to predict $P(X_{Sys} \mid X_U)$.
$\approx$ System dynamics	
Conclusion	**Explanation**
Yes	The software system is static.
Fail-safe	
Conclusion	**Explanation**
-	Irrelevant as Φ_b is Φ_b-analysable.
Class: Static analysability	

Table 8.6.: Classification result of the HRI example system by considering our reliability prediction approach.

work, the authors proved safety properties of two kinds of collision avoidance systems, namely *VerticalCAS* which issues vertical rate advisories to an aircraft to avoid *Near Midair Collisions* (NMACs) with another aircraft and *HorizontalCAS* which issues turn rate advisories to an aircraft to avoid NMACs. The collision avoidance problem is formulated by using MDPs. A state is composed of several (physical) variables, e.g. in *VerticalCAS* a state consists of five variables (three variables describing the vertical encounter geometry, one variable capturing the horizontal geometry and one variable representing the previous advisory). The transition function t is constructed based on the dynamic model of aircraft. The action space relates to the set of advisories the system can issue to the pilot. Based on dynamic programming techniques, the policy π can be computed which maximises the accumulated reward over time (w.r.t. some reward function).

As a result, π is represented by a large table of states to advisory mappings. Due to storage constraints of certified avionics hardware, however, the table violates the storage constraints and is not applicable. Thus, DNNs are used to learn or approximate the table to compress the data and to meet storage requirements [96, 97]. However, DNNs are highly complex, continuous and non-linear functions for which it is difficult to predict whether the outputs are correct or not (recall the black-box nature of DNNs). Moreover, Julian and Kochenderfer [95] pointed out that pure simulation of the system is not sufficient because it cannot be guaranteed whether the DNN performs correctly in all possible states.

Guaranteeing safety for DNN-based ACASs To make safety guarantees of the advisories produced by the DNN, Julian and Kochenderfer developed an approach which makes use of DNN verification techniques (e.g. [99, 202]) and a reachability analysis which verifies whether an NMAC is reachable. If it can be shown that no advisory results in an NMAC, the system is guaranteed to be safe (for more details we refer to the original work of Julian and Kochenderfer [95]).

Informally, we define the system-level property Φ_{Sys}: "The ACAS is guaranteed to cause no NMAC". In summary, the reachability analysis can be divided into two main procedures. The first procedure splits the state space (which is the input space of the DNN) into smaller regions. By applying formal verification techniques for DNNs (in this case, symbolic bound propagation [201]), it is verified which advisories $\mathcal{A}_c$ can be given within a region (or cell) c. The AI-specific property Φ_b thus refers to an input-output property, where output bounds are proved for a given input range. The second procedure starts to identify those regions or cells c representing the states that could occur before the DNN-based ACAS takes action. Afterwards, the system makes use of the system dynamics to compute the next state regions, i.e. the regions of the state space at time $t + 1$ one would observe when following the advisories and system dynamics given the previous regions at time t. Because the state space (or input space) has been already split into regions $\mathcal{A}_c$ for which some advisories are proven, the next advisories of the regions computed at $t + 1$ are determined. This procedure is repeated until either an NMAC cell is reached or it converges to a set of reachable regions with no NMAC. If no NMAC cell is reached, the ACAS is guaranteed to be safe.

The classification of the DNN-based ACAS is shown in Table 8.7. We consider the analytic capacity of the discussed ACAS to be Φ_b-analysable. More specifically, the Φ_b-monitorability is given by a proof, i.e. the input space is partitioned into a set of regions $\mathcal{A}_c$ for which it can be proven that the DNN produces a certain set of advisories by using symbolic bound propagation [201]. Moreover, we consider the state space to be Φ_b-explorable because all relevant states are verified in terms of Φ_b (or rather $\mathcal{A}_c$) by exploring the state space w.r.t. the system dynamics. In this case, it is not relevant whether the AI component is input or output monitorable since we can prove the ACAS to operate safely.

In terms of abstractability, we consider the ACAS to be sufficiently abstracted. More specifically, the system is represented as an MDP where the state space is spanned by several (physical) variables. In addition, the transition function is based on the well-researched physical dynamics of an aircraft. Because the transition function represents the dynamics of the system, we consider the system dynamics to be accurately approximated (again, due to the well-known aerodynamic properties of an aircraft). In conclusion, the DNN-based ACAS is classified into static analysability.

Reliability prediction In this part of the section, we would now apply our reliability prediction approach. However, our approach is barely applicable to ACAS for two reasons: First, since the safe operation has been already proven there is no need to apply a reliability

Analytic Capacity	
Φ_b-monitorability	**Explanation**
Verifiable	Proof by input space partitioning $\mathcal{A}_c$ (using neural network verification tools like [99, 202]).
State space complexity	
Φ_b-explorable	All safety-critical states are verified by the reachability analysis.
Input-output-monitor.	
-	Irrelevant due to verifiability of Φ_b.
	Φ_b-analysable: Yes
Abstractability	
Conclusion	**Explanation**
Yes	MDPs are used to represent the logic of collision avoidance systems. Transition function t is constructed according to the well-known physics of an aircraft.
$\approx$ System dynamics	
Conclusion	**Explanation**
Yes	Known from the physical dynamics of an aircraft.
Fail-safe	
Conclusion	**Explanation**
-	Irrelevant as Φ_b is Φ_b-analysable.
Class: Static analysability	

Table 8.7.: Classification result of an aircraft collision avoidance system by considering the safety guaranteeing approach of Julian and Kochenderfer [95].

analysis. Secondly, the environmental dynamics of an ACAS are rather low for aircraft; that is, there are fewer environmental variables or uncertainties that could force the AI component to make wrong predictions (in contrast to, e.g., self-driving cars). Julian and Kochenderfer pointed out that pilot delays (to respond to advisories) or sensor errors are potential sources of uncertainty; however, the reachability analysis can be expanded to account for both uncertainties. Therefore, we do not discuss our reliability prediction approach.

8.3.4. Discussion

After applying our classification structure to several AI-enabled systems, we now discuss the results. We could show that the classification structure could be applied to each system. Thus, we conclude the general applicability of the classification structure and architectural dependability assurance classes. Although we considered only a limited number of AI-enabled systems, we chose very generic and representative AI-enabled systems from different domains. Since we are no experts, it is debatable whether some classification decisions can be made as we have done. However, even if different decisions were made for some classification dimensions, we have not encountered situations where the dimensions were neither appropriate nor applicable. Therefore, we conclude that the classification structure and its dimensions are applicable.

We now discuss research question **RQ4** and its sub-questions. As our dependability assurance classes and dimensions are to be regarded as preliminary and still subject to research, it is not possible to answer the research question comprehensibly. Therefore, we can only give preliminary answers. To improve readability, we recap the individual research questions.

Research Question 4: How to assess the extent to which dependability assurances can be given for an AI-enabled system?

However, before answering the research question, we must first answer the sub-research questions. Therefore, recall research question **RQ4.1**:

Research Question 4.1: What are appropriate classes of architectural dependability assurances?

We addressed the research question by our four classes of architectural dependability assurance: Static analysability, monitor analysability, a-posteriori analysability and non-analysability. The most favourable classes are static and monitor analysability because they allow assurances to be given at design-time. A-posteriori analysability involves only runtime assurances while for non-analysable systems no assurances can be given. The assurances are always associated with a particular system-level property, e.g. the system success probability or the ability to avoid collision with another object at any time. This means, however, that the system is always classified w.r.t. to the system-level property. In the 8.3 section, we could assign all the systems to one of the classes. Therefore, we could demonstrate the applicability of the classes.

Recall research question **RQ4.2** which asks for suitable classification dimensions:

Research Question 4.2: What are the suitable dimensions for classification?

We identified four classification dimensions consisting of the analytic capacity (which itself is defined by three dimensions: state space complexity, Φ_b-monitorability and input-output-monitorability), abstractability, approximation of the system dynamics and fail-safe. Based on the classification dimensions, we have developed a classification structure that makes it possible to assign a particular system and its environment to one of our classes. In section 8.3, we were able to classify each AI-enabled system w.r.t. the dimensions. Due to the limited number of systems that we classified, it cannot be said for certain whether the dimensions are complete or whether they must be refined to some extent. However, in

8.3 section, we demonstrated that they form a solid foundation, but need to be researched further.

Considering **RQ4.1** and **RQ4.2** together, we conclude that we addressed the research question with our architectural dependability assurance classes and dimensions. The dimensions and classification structure classifies systems and their environments (w.r.t. Φ_{Sys}) into one of the classes. Nevertheless, research question **RQ4** cannot be fully answered due to the restricted discussion of the classification structure. To have a more informed discussion, we would have to interview experts for each area. However, this involves a great deal of effort, which was no longer feasible within the scope of this work, but is the subject of future work. However, the classification of AI-enabled systems in section 8.3 has demonstrated general applicability and strongly suggests that the classification structure and classes already provide a solid foundation. Furthermore, we argue that our classes and classification structure provide software engineers with guidelines to judge at what level assurances can be given for a system-level property and whether the system-level property needs to be refined and broken down into more analytically amenable properties. Especially for the last point, our class and classification structure supports software engineers to identify system properties that are difficult to assure as such. For example, we argue that an AI system which is classified into a lower-ordered class (e.g., non-analysability) is symptomatic of a poorly defined and hard-to-assure system property. In such cases, it is arguably more advisable to split the original system property into multiple (and analytically more amenable) properties that can be classified into higher-order classes.

8.4. Summary

In this chapter, we introduced four classes of architectural dependability assurance, namely static analysability, monitor analysability, a-posteriori analysability and non-analysability. Each class is associated with different dependability assurances one can make regarding a particular system-level property. AI-enabled systems and the environments (in which they operate) can be classified into one of the classes that indicate the extent to which assurances can be made. Moreover, we discussed a classification structure consisting of several classification dimensions, namely abstractability, approximation of system dynamics, analytic capacity and fail-safe. Based on the dimensions, we discussed how to classify AI-enabled systems. Afterwards, we envisioned how the classification structure can be used to build dependability assurance cases. Finally, we applied the classification structure to representative and well-known examples of AI-enabled systems and discussed its applicability.

Part V.

Validation

9. Validation

In this section, we validate our presented approaches by considering a series of case studies of different domains. The primary target of the validation is to answer our research questions stated in section 1.3. The validation process of this thesis is organised and structured similarly to the validation process of the dissertation from Stier [178]. That is, we start with an overview of the validation (including the presentation of the GQM plan, the case study systems and the discussion of the validation process), carry out the validation for each validation goal and discuss the results afterwards. Just like in the work of Stier, our validation is guided by a GQM (goal-question-metric) plan based on Basili et al. [36] (see section 2.8.1). Therefore, we start to elaborate the GQM plan by defining four central validation goals. The validation goals are complemented by validation questions that need to be answered to determine whether the respective validation goal is achieved. Each validation question is associated with a set of metrics that aim to answer the validation question under consideration. For each validation goal, we have created a separate section in which the corresponding validation goal is examined.

The remainder of this chapter is structured as follows: In section 9.1, we provide an overview of the validation and introduce the GQM plan. In sections 9.2, 9.3 and 9.4 the performed validations of the individual validation goals are presented. Finally, the results of the validation are discussed in section 9.5.

9.1. Overview

In this section, we give an overview of the validation. Therefore, we start to present our GQM plan in the next section. Afterwards, we discuss the validation levels (based on Böhme and Reussner [28]) associated with the corresponding validation goal and considered case studies. Moreover, we explain the structure of the validation process and its accomplishment.

9.1.1. Validation Goals, Questions and Metrics

In this section, we present the GQM plan and its validation goals, questions and metrics. We relate each validation goal with its addressed research questions. Therefore, we start to present the GQM plan as such and discuss the individual validation goals and the interpretation of the metrics subsequently.

Before we present the GQM plan, we have to introduce further notations that we use in the metrics:

- As introduced in definition 30, we consider an AI black-box component as a function b which maps inputs of the input space to outputs of the output space. In addition, with b^+ we denote an AI component which produces for any input the correct output; with b^- we denote an AI component which produces for any input the incorrect output.
- We use the notation C_b to express that architectural configuration C includes AI component b; in addition, M_{C_b} indicates the corresponding architecture model (or PCM model) that describes C_b.
- The function $acc(b)$ abstracts a performance measure for a trained AI model b, e.g. based on RMSE (root-mean-squared error).
- For simplification, we use the function $rel(M_{C_b} \mid \varphi_1, \dots, \varphi_N)$ to represent a prediction run with our reliability prediction approach for AI-enabled systems from section 7, i.e. $rel(M_{C_b} \mid \varphi_1, \dots, \varphi_N) = Pr(X_{Sys} = Success \mid X_U = U, X_{\varphi_1} = \varphi_1, \dots, X_{\varphi_N} = \varphi_N)$ (recall from formula (7.1)) where the probability of success of the system (defined over X_{Sys}) is investigated w.r.t. M_{C_b} and for given uncertainty tuple $(\varphi_1, \dots, \varphi_N)$ and a fixed usage scenario U. Moreover, with $rel(M_{C_b})$ we refer to the overall probability of success of the system (or M_{C_b}), i.e. $rel(M_{C_b}) = Pr(X_{Sys} = Success \mid X_U = U) = \sum_{\varphi_1,\dots,\varphi_N \in \Phi} Pr(X_\Phi = \varphi_1, \dots, \varphi_N) \cdot rel(M_{C_b} \mid \varphi_1, \dots, \varphi_N)$ (recall from equation (7.4)).
- The set $\Phi_{b_i,b_j} := \{(\varphi_1, \dots, \varphi_N) \in \Phi \mid Pr(X_{b_i} = Success \mid X_{\varphi_1} = \varphi_1, \dots, X_{\varphi_N} = \varphi_N) \geq Pr(X_{b_j} = Success \mid X_{\varphi_1} = \varphi_1, \dots, X_{\varphi_N} = \varphi_N)\}$ is defined over two distinct AI components b_i and b_j and contains all uncertainty tuples $(\varphi_1, \dots, \varphi_N)$ for which the sensitivity model of b_i indicates higher success probabilities than the sensitivity model of b_j.
- We write an adaptation strategy $\pi[b]$ if the strategy π is safeguarding AI component b.

Based on the previous notations, we define a set of plausibility assertions. Let $\{b_1, b_2, \dots\}$ be a set of AI components under investigation where $acc(b_i)$ induce a partial order over the set, i.e. $b_1 \leq b_2 \leq \cdots \Leftrightarrow acc(b_1) \leq acc(b_2) \leq \dots$ Moreover, let denote $M_{C_{b_i}}$ the corresponding architecture model including b_i where the architecture models exclusively differ in the used AI component. We consider the following plausibility assertions:

$$b^- \leq b_1 \leq \cdots \leq b^+ \Leftrightarrow rel(M_{C_{b^-}}) \leq rel(M_{C_{b_1}}) \leq \cdots \leq rel(M_{C_{b^+}}) \tag{9.1}$$

$$\begin{aligned} &\forall(\varphi_1, \dots, \varphi_N) \in \Phi_{b_i,b_j} : \\ &\qquad rel(M_{C_{b_i}} \mid \varphi_1, \dots, \varphi_N) \geq rel(M_{C_{b_j}} \mid \varphi_1, \dots, \varphi_N) \end{aligned} \tag{9.2}$$

$$\begin{aligned} &\forall(\varphi_1, \dots, \varphi_N) \in \Phi_{b_i,b_j} \cap \Phi_{b_j,b_i} : \\ &\qquad rel(M_{C_{b_i}} \mid \varphi_1, \dots, \varphi_N) = rel(M_{C_{b_j}} \mid \varphi_1, \dots, \varphi_N) \end{aligned} \tag{9.3}$$

$$b^- \leq b_1 \leq \cdots \leq b^+ \Leftrightarrow \pi[b^-] \leq \pi[b_1] \leq \cdots \leq \pi[b^+] \tag{9.4}$$

The first plausibility assertion (9.1) states that the ordering of the AI components must be preserved by our reliability prediction approach. That is, for any distinct b_i, b_j with $b_i \leq b_j$ our reliability prediction approach applied to the respective models $M_{C_{b_i}}$ and $M_{C_{b_j}}$ must indicate the same ordering in terms of the success probability (recall again that both models only differ in the used AI component). We extend the set of AI components that are subject to validation with b^- and b^+. The order of the set must therefore start with b^- (reflecting the worst possible AI component) and end with b^+ (reflecting the best possible AI component). When applied to our reliability prediction model, the prediction order must preserve the two extremes.

The plausibility assertion (9.2) states that for any pair b_i, b_j where the sensitivity model of b_i yields higher success probabilities than the sensitivity model of b_j for some uncertainty tuples, this must also be reflected by the individual prediction results of our reliability prediction tool. For AI components whose sensitivity models indicate the same success probabilities for some uncertainties, the reliability results for these uncertainty tuples must be equal; this is captured by the plausibility assertion (9.3). Note that (9.3) directly follows from (9.2); nonetheless, we make the assertion explicit to clearly emphasise that this plausibility relation must hold.

In essence, the plausibility assertion (9.4) states that any adaptation strategy π used to safeguard the AI components b_i and b_j (i.e. $\pi[b_i]$ and $\pi[b_j]$) must generate higher rewards in safeguarding b_i if and only if $b_i \leq b_j$. Hereby, we assume that each adaptation strategy $\pi[b]$ implements the same adaptation logic and only differs in the AI black-box component b to be safeguarded. The ordering of strategies is based on formula (6.6). Moreover, it is assumed that each strategy is evaluated with the same reward function where a reward directly reflects the reliability of the system, i.e. the success probability.

9.1.1.1. Evaluating Adaptation Strategies

The validation goals of this section are related to the first research questions and their sub-questions:

Research Question 1: How to evaluate adaptation strategies of self-adaptive systems at design-time regarding the ability to meet quality objectives?

Research Question 1.1: How can environmental dynamics be formalised domain-independently at design-time?

Research Question 1.2: What is an appropriate level of abstraction to represent the environmental dynamics domain independently? By appropriateness, we mean that

- adaptation strategies can be analysed at design-time with sufficient accuracy.
- environmental state spaces can be described flexibly and compactly.

Research Question 1.3: What is an appropriate analytical model to enable design-time analyses of self-adaptive systems?

Research Question 1.4: Are the predictions sufficiently accurate to yield plausible results?

The validation goals are divided to validate key aspects of **RQ1**, namely the applicability of the formal modelling language of environmental dynamics and the appropriateness of using MDPs as an analytical model for evaluating adaptation strategies. The GQM plan presented in the next two sections adheres to the same structure as in Stier's dissertation [178].

Goal 1. Analyse the applicability of the *EnvDyn* formal modelling language to describe the operating environment of self-adaptive systems.

Addressed RQs: RQ1, RQ1.1, RQ1.2

Question 1.1. Can we instantiate and apply the *EnvDyn* modelling language domain-independently?

Metric 1.1.1. Rank of adaptation strategies evaluated by the DeltaIoT simulator compared to adaptation strategy ranking of the *SimExp* method.

Metric 1.1.2. Rank of adaptation strategies evaluated by SimuLizar compared to adaptation strategy ranking of the *SimExp* method.

Metric 1.1.3. Plausibility assertion checks based on measured reliability properties of the AI components. More specifically, plausibility assertions (9.1)-(9.4) are validated.

Addressed RQs: RQ1, RQ1.1

Question 1.2. Are the essential characteristics of an operating environment captured by the *EnvDyn* modelling language?

Metric 1.2.1. Rank of adaptation strategies evaluated by the DeltaIoT simulator compared to adaptation strategy ranking of the *SimExp* method.

Metric 1.2.2. Rank of adaptation strategies evaluated by SimuLizar compared to adaptation strategy ranking of the *SimExp* method.

Metric 1.2.3. Plausibility assertion checks based on measured reliability properties of the AI components. More specifically, plausibility assertions (9.1)-(9.4) are validated.

Addressed RQs: RQ1, RQ1.2

Goal 2. Analyse the appropriateness of our *SimExp* method to evaluate adaptation strategies of self-adaptive systems at architecture-level.

Addressed RQs: RQ1, RQ1.3, RQ1.4

Question 2.1. Does our *SimExp* method achieve the evaluation results of comparable quality for adaptation strategies compared to domain-specific simulators?

Metric 2.1.1. Rank of adaptation strategies evaluated by the DeltaIoT simulator compared to adaptation strategy ranking of the *SimExp* method.

Metric 2.1.2. Rank of adaptation strategies evaluated by SimuLizar compared to adaptation strategy ranking of the *SimExp* method.

Addressed RQs: RQ1, RQ1.3, RQ1.4

Question 2.2. Can our *SimExp* method evaluate design decisions within an adaptation strategy family or the comparison of distinct adaptation strategies?

Metric 2.2.1. Rank of adaptation strategies evaluated by the DeltaIoT simulator compared to adaptation strategy ranking of the *SimExp* method.

Metric 2.2.2. Rank of adaptation strategies evaluated by SimuLizar compared to adaptation strategy ranking of the *SimExp* method.

Addressed RQs: RQ1, RQ1.3

Question 2.3. Does our *SimExp* method support the decision-making process regarding the design decision of whether to use a self-adaptive system or static software systems?

Metric 2.3.1. Rank of adaptation strategies evaluated by the DeltaIoT simulator compared to adaptation strategy ranking of the *SimExp* method.

Metric 2.3.2. Rank of adaptation strategies evaluated by SimuLizar compared to adaptation strategy ranking of the *SimExp* method.

Addressed RQs: RQ1, RQ1.3

Note that the first validation goal (the applicability of the *EnvDyn* modelling language) is implicitly covered by the remaining validation goals. The first part of validation goal 1 is concerned with the domain-independent application. As we use four case study systems during the entire validation of this work, we instantiate the *EnvDyn* metamodel in each case study. That is, we show the applicability of the modelling language in four different domains. The second part of validation goal 1 relates to the appropriateness in terms of coverage of the essential characteristics of a domain. For each case study, a corresponding environment model is created and used in adaptation strategy evaluation and reliability prediction. That is, if the resulting evaluation and prediction results are shown to be valid in the respective validation goals, we can conclude the applicability of the *EnvDyn* metamodel.

9.1.1.2. Analysis of Architectural Safeguards for AI-enabled Systems

The validation goals of this section are related to the second research question and their sub-questions:

Research Question 2: How can software systems that contain AI black-box components be evaluated in terms of meeting reliability attributes at design-time?

Research Question 2.1: How to deal with the hidden state problem of AI black-box components?

Research Question 2.2: How to systematically consider the influence of predictive uncertainty and causally related environmental variables in the reliability prediction?

Research Question 3: How can adaptation strategies of self-adaptive systems that safeguard uncertain AI black-box components be evaluated in terms of reliability at design-time?

Goal 3. Analyse the plausibility of our reliability predictions of AI-enabled software systems at architectural-level.

Addressed RQs: RQ2, RQ2.1, RQ2.2

Question 3.1. Do sensitivity models adequately capture AI black-box components in reliability prediction?

Metric 3.1.1. Bhattacharyya distance to measure the similarity between the success/failure probabilities of the sensitivity model and the reliability predictions.

Metric 3.1.2. Plausibility assertion checks based on measured properties of the AI components. More specifically, plausibility assertion (9.1) is validated.

Addressed RQs: RQ2, RQ2.1

Question 3.2. Do the prediction results of our holistic approach reflect reliability attributes measured from AI components?

Metric 3.2.1. Plausibility assertion checks based on measured properties of the AI components. More specifically, plausibility assertions (9.1)-(9.3) are validated.

Addressed RQs: RQ2, RQ2.2

Question 3.3. Does the approach allow for the evaluation of AI-specific design decisions?

Metric 3.3.1. Plausibility assertion checks based on measured properties of the AI components. More specifically, plausibility assertions (9.2)-(9.3) must hold after applying a design decision.

Addressed RQs: RQ2, RQ2.2

Goal 4. Analyse the plausibility of the instantiated *SimExp* method for the evaluation of adaptation strategies of self-adaptive systems safeguarding AI black-box components at architecture-level.

Addressed RQs: RQ3(, RQ1.3)

Question 4.1. Do the generated rewards of an adaptation strategy reflect reliability attributes derived from AI components?

Metric 4.1.1. Plausibility assertion checks based on measured properties of the AI components. More specifically, plausibility assertion (9.4) is validated.

9.1.2. Case Study Systems

In this section, we briefly enumerate the case study systems that we consider in the validation. In total, we consider four case studies:

CS1 **DeltaIoT system**: We use the DeltaIoT system as a case study which has been introduced in section 1.5.2. The DeltaIoT system is a widely known case study in the self-adaptive system community. Moreover, it is complemented with a simulator for the evaluation of adaptation strategies implemented for the DeltaIoT system.

CS2 **Load balancing based on the ZNN.com system**: The second case study that we consider corresponds to the load balancer example system from section 1.5.1 which is based on the ZNN.com community case study [48]. The case study has already been used in the context of SimuLizar [15]. SimuLizar provides means for scenario-based performance evaluation of adaptation strategies which we use to compare with the results of our approach based on the load balancer case study.

CS3 **Human-Robot-Interaction system**: The HRI system has been introduced in section 1.5.3. The case study is primarily used to evaluate adaptation strategies for self-adaptive systems safeguarding AI black-box components.

CS4 **Udacity self-driving car challenge**: Finally, we consider a case study from the autonomous driving domain. The case study is part of the Udacity self-driving car challenge [192]. In the challenge, several teams were developing DNNs (deep neural networks) for predicting the steering angles of a self-driving car based on image data. For our validation, we consider two trained DNNs from the set of developed DNNs which we use for further investigation.

9.1.3. Validation Process

In this section, we discuss the validation process. We start to classify the validation goals and questions into the validation levels of Böhme and Reussner [28] and assign the considered case studies to the respective validation question. Moreover, the validation question specific interpretation of metrics are explained. Afterwards, we discuss the validation accomplishment, i.e. the structure of the validation process, ordering of the validation goals, etc.

9.1.3.1. Classification into Validation Levels

For the classification of the validation goals and questions, we use the validation levels of Böhme and Reussner [28]. The validation levels have been introduced in section 2.8.2. We associate each validation question of a validation goal with its corresponding validation levels. In the following, we discuss the validation levels of each goal.

Validation goal 1 Table 9.1 provides an overview of the validation question to validation level assignment and the considered case study systems as validation foundation.

Validation question	Validation level	Case study systems
1.1	Level II	CS1-CS4
1.2	Level II	CS1-CS4

Table 9.1.: Overview of the assignment of validation level to question of goal 1.

The validation goal 1 is about analysing the applicability of our *EnvDyn* metamodel and is linked to two validation questions. Question 1.1 is concerned with the domain-independent applicability of the modelling language; question 1.2 is about whether the essential characteristics of an operating environment are captured. In the following, we examine the levels at which both questions are validated, the case study systems and the interpretation of the metrics for each question.

We conducted a level II validation to address both questions. To validate whether the *EnvDyn* metamodel is both applicable domain-independently and captures the key characteristics of an operating environment is determined by considering the validity of the results validated in the remaining goals (and w.r.t. all case study systems). The *EnvDyn* modelling language is used in both the evaluation of adaptation strategies (i.e. the *SimExp* method) and the reliability prediction of AI-enabled systems; thus, their respective validity strongly depends on the applicability of the *EnvDyn* metamodel. That is, if the results are valid, both questions are positively answered. Therefore, we link each question to each metric used to validate either the *SimExp* method or the reliability prediction approach. In this case, however, the metrics are not generally interpretable (or at least not in the context of goal 1). Instead, one must take into account the interpretations of each metric in the remaining validation goals where the metric is used to answer a validation question.

Validation goal 2 Table 9.2 shows the validation question to validation level assignment and the considered use cases to validate the question.

Validation Question	Validation level	Case study systems
2.1	Level I	CS1-CS2
2.2	Level I & II	CS1-CS2
2.3	Level I & II	CS1-CS2

Table 9.2.: Overview of the assignment of validation level to question of goal 2.

For validation goal 2, we formulated three questions. Question 2.1 is concerned with whether our *SimExp* approach produces comparable results by considering domain-specific simulators for evaluating adaptation strategies. We conducted a level I validation to answer the question. Moreover, we consider metrics 2.1.1 and 2.1.2. The metrics are intended to validate that the evaluation order of adaptation strategies made by domain-specific simulators are preserved by the *SimExp* method (which builds upon MDPs). As domain-specific simulators, we consider the DeltaIoT simulator and SimuLizar. Therefore, the used case study systems are CS1 and CS2 because both simulators are applicable to the respective case study. The DeltaIoT simulator has been used in many evaluation settings (e.g. [194, 168]). Consequently, we assume that the simulator is sufficiently accurate to be used in our context. Also, SimuLizar has been evaluated for the load balancer case study that we consider [15]. The rationale behind the metrics is to compare the results of our MDP based *SimExp* approach with two established simulators. If *SimExp* produces the same ranking of strategies as the domain-specific simulators, we can conclude the appropriateness of *SimExp* and the appropriateness of MDPs as an underlying analytical model. We do not aim to be more accurate than the simulators; this is arguably hard to achieve due to the level of abstraction and simplifications associated with the *SimExp* method. On the other hand, however, to obtain the ranking of adaptation strategies, *SimExp* must at least satisfy a particular level of accuracy, otherwise one would observe different ranks resulting from our *SimExp* method.

For question 2.2, we conduct a level I validation. The question is about whether our *SimExp* method allows the evaluation of design decisions within adaptation strategy families and whether distinct strategies are comparable. We use the metrics as before, i.e. 2.2.1 and 2.2.2 (and thus the same case study systems, i.e. CS1 and CS2). For the level I validation, the metrics must be interpreted as discussed in question 2.1. Only if the rank of an adaptation strategy family is preserved, one can evaluate design decisions and compare distinct strategies based on the rank generated by *SimExp*.

For validation question 2.3, we use the same metrics and case study systems as in question 2.2. The measurements of the metrics are also interpreted in the same way. The only difference is that we compare the results of static and self-adaptive systems produced by the domain-specific simulators compared to the results of the *SimExp* method.

For validation questions 2.2 and 2.3, we conduct also a level II validation. We use the same metrics as for Level I validation, but focus on whether our *SimExp* method enables software engineers to evaluate meaningful design decisions based on the models acquired from the respective case study systems and relevant to *SimExp*.

Validation goal 3 Table 9.3 shows the validation question to validation level assignment and considered case study systems.

Validation Question	Validation level	Case study systems
3.1	Level I	CS4
3.2	Level I	CS4
3.3	Level II	CS4

Table 9.3.: Overview of the assignment of validation level to question of goal 3.

The validation goal is associated with three validation questions. The first two questions are validated by a level I validation; question 3.3 is conducted in the context of a level II validation.

Question 3.1 is about whether sensitivity models adequately capture AI black-box components. We address the question by two metrics, namely metric 3.1.1 and metric 3.1.2. Moreover, for the validation of the goal we use solely case study CS4. The first metric measures the similarity of the success/failure probability of the sensitivity model (or rather its distribution) with the predicted success probabilities w.r.t. the individual uncertainty tuples. Recall that for each reliability prediction run, we only change the AI component b such that only b affects the reliability of the entire system. As a result, the predicted success/failure probabilities of the system M_{C_b} must indicate some degree of similarity with the success/failure probabilities of the respective sensibility model of b. Therefore, we calculate the Bhattacharyya distance (see section 2.8.3) to measure the similarity of both distributions. For this purpose, we predict the individual success and failure probabilities for each uncertainty tuple by our reliability prediction approach and compare the results with the success and failure probability with the sensitivity model of the AI component under investigation. More formally, for all uncertainty tuple $(\varphi_1, \dots, \varphi_N)$, we compute the Bhattacharyya distance of the distributions $P(X_b \mid X_{\varphi_1} = \varphi_1, \dots, X_{\varphi_N} = \varphi_N)$ and $P(X_{Sys} \mid X_U = U, X_{\varphi_1} = \varphi_1, \dots, X_{\varphi_N} = \varphi_N)$. If the value is close to 0, we can conclude that both distributions are similar. With metric 3.1.2, we check whether plausibility assertion

(9.1) holds which requires for a given ordering of AI components (w.r.t. some performance measure), the same ordering of reliability predictions.

For question 3.2, we consider metric 3.2.1. Hereby, we take into account plausibility assertions (9.1)-(9.3). For question 3.1, we required that assertion (9.1) is satisfied from the perspective of the sensitivity model. In this case, we require that assertions (9.1)-(9.3) are satisfied from the perspective of our reliability prediction approach. More specifically, in terms of assertion (9.1), it is required that for a given set of AI components, our holistic reliability prediction approach reflects their individual reliability properties, i.e. the accuracy of an AI component. This is complemented by the assertions (9.2) and (9.3), which for any two AI components, say b and b', giving different or equal prediction confidences (i.e. predictive uncertainty) for particular $(\varphi_1, \dots, \varphi_N)$, require that exactly these prediction confidences are reflected in the individual reliability predictions $rel(C_b \mid \varphi_1, \dots, \varphi_N)$ and $rel(C_{b'} \mid \varphi_1, \dots, \varphi_N)$ under identical architectural model C.

For question 3.3, we conduct a level II validation and, similarly to goal 2, focus on whether our reliability prediction approach supports software engineers in evaluating design decisions based on the models acquired from the respective case study systems. More specifically, we consider metric 3.3.1 which is associated with plausibility assertions (9.2)-(9.3). The metric intends to measure whether the assertions still hold after applying a design decision (e.g. the application of the filtering or n-version pattern). For question 3.2, we validate whether the assertions (9.2)-(9.3) hold for a fixed architecture model M_{C_b} where only the used AI component b is exchangeable. Now, we require that if we apply a specific design decision to the architecture model M_{C_b} such that we obtain $M_{C'_b}$, the plausibility assertions must still hold. Note that a design decision has an impact on the sensitivity model of b (either structural or parametric, recall architectural countermeasures from section 7.1.3.2). That is, the sensitivity model originally associated with b is modified, i.e. it accounts for the effects of the design decision. Consequently, the sets Φ_{b_i,b_j} are modified accordingly. However, the plausibility assertions (9.2)-(9.3) for the modified sets Φ_{b_i,b_j} must still hold. If this is the case, our reliability prediction approach has been shown to enable the evaluation of AI-specific design decisions.

Validation goal 4 Finally, Table 9.4 shows the mapping of the validation questions of goal 4 to the corresponding validation levels.

Validation Question	Validation level	Case study systems
4.1	Level I	CS3-CS4

Table 9.4.: Overview of the assignment of validation level to question of goal 4.

For validation goal 4, we conduct a level I validation. The goal is related to a single validation question. Question 4.1 is about whether the generated rewards of an adaptation strategy reflect the reliability attributes derived from an AI component. In terms of question 4.1, metric 4.1.1 is used to measure whether plausibility assertion (9.4) is satisfied or maintained by our *SimExp* approach. Suppose a set of AI components with varying degrees of accuracy and an adaptation strategy that implements some adaptation logic. The assertion 9.4 states that when the adaptation strategy is evaluated for each AI component, the individual evaluation results must indicate the same order as given by the order of the AI components w.r.t. their accuracies. In other words, for two AI components, say b and b' with $acc(b') \leq acc(b)$, it is required that the overall result of the adaptation strategy safeguarding b is higher compared to the strategy safeguarding b' (where the strategies implement the same adaptation logic but differ in b and b' respectively).

9.1.3.2. Validation Accomplishment

In this section, we discuss how we conduct the validation for the distinct validation goals.

We start to validate goals 2-4 and goal 1 afterwards. The reason for this has already been explained in the previous section. To validate the goal 1, we must first validate the other goals, as the validity of their results is strongly related to the validity of goal 1.

Therefore, we start with validation goal 2. To the best of our knowledge, there is no gold standard approach that enables the evaluation of adaptation strategies domain-independently. However, there are domain-specific simulators that one can use to evaluate adaptation strategies for a given domain, e.g. the DeltaIoT simulator [92] in the field of IoT. We use these domain-specific simulators as a baseline to reason about the appropriateness of using MDPs as analytical models to evaluate adaptation strategies. Once we have shown the appropriateness of using MDPs, we continue to validate our *SimExp* method in terms of evaluating adaption strategies that safeguard AI black-box components.

However, we have to validate goal 3 before as it is concerned with relevant concepts. The validation of these concepts must be considered as a prerequisite for validating goal 4, namely the reliability prediction of AI-enabled systems. Also, we are facing here the same problems as in goal 2; to the best of our knowledge, there are no comparable approaches that we can use as a baseline or ground truth. Another possibility would be to implement an AI-enabled case study system to compare the runtime measurements with our predictions. However, besides the high effort associated with such a validation, obtaining accurate measurements for reliability metrics is hard to achieve due to the rare nature of events influencing the reliability of the system [28]. Therefore, we follow a different validation approach. More precisely, we check plausibility assertions based on measurable reliability attributes in the considered domain or case study system, which have to be preserved by our reliability prediction approach. At first glance, this seems contradictory, as one would expect the accuracy of our approach to be validated. Looking at the components that make up our reliability prediction approach (sensitivity model, PCM-Rel and ATs) and the way they are connected, it is clear that there is no need to validate accuracy. More specifically, recall that the reliability prediction approach demands an upstream sensitivity analysis of an AI component. For this purpose, we enumerated a set of approaches that can be used to conduct the sensitivity analysis (see section 7.1.2). Based on the resulting sensitivity model, our approach predicts the respective success probabilities of the system for distinct uncertainty tuples by using the conventional PCM-Rel prediction approach. At this stage, it is important to note that we did not modify the code and thus the prediction logic of PCM-Rel (see section 7.1.3). Instead, we implemented an upstream resolving procedure for recalculating the failure probability of the analysed AI component based on the sensitivity model. Consequently, validating the accuracy of our reliability prediction approach is equal to validating the accuracy of PCM-Rel which was extensively validated by Brosch [33]. Additionally, our reliability prediction approach is complemented by ATs which can be applied to improve the reliability of the system. The AT approach was also validated by Lehrig [113]. Thus, our reliability prediction approach for AI-enabled systems is composed of well-validated components such that only their joint interaction needs to be validated holistically. Therefore, we focus the validation on plausibility assertions that need to be preserved to reason about the validity of the joint interaction of the components that make up our approach to predicting reliability. The plausibility assertions have been discussed in previous sections.

Following validation goal 3, we finally validate our *SimExp* method for evaluating adaptation strategies that safeguard AI components in validation goal 4. As before, there is no gold standard or baseline against which we can compare our approach. However, as we have already validated the *SimExp* method in validation goal 2, we validated the appropriateness of the *SimExp* method; thus, we validate in this context again the preservation of plausibility assertions. The results of this validation goal also complement the validation results of goal 2 and increase confidence and reasoning that MDPs are appropriate for evaluating adaptation strategies.

9.2. Evaluating Adaptation Strategies of Self-Adaptive Systems

In this section, we validate validation goal 2 by considering the DeltaIoT case study and the load balancer case study in conjunction with SimuLizar, a performance simulator for self-adaptive systems.

9.2.1. DeltaIoT

In the following, we validate validation goal 2 by considering the DeltaIoT community case study. Therefore, we start to outline how we instantiated the *SimExp* method for the DeltaIoT system. The case study itself has already been introduced in section 1.5.2. Subsequently, we discuss the experimental setup and present the results afterwards.

9.2.1.1. Instantiation of *SimExp*

For the instantiation of the *SimExp* method, we created the models and other artefacts required to analyse adaptation strategies with *SimExp*, namely the architecture model

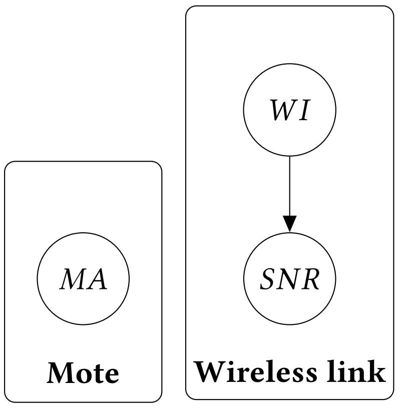
Link	Power	Distribution
2 to 4	15	100
3 to 1	15	100
4 to 1	15	100
5 to 9	15	100
6 to 4	15	100
7 to 2	15	0
7 to 3	15	100
8 to 1	15	100
9 to 1	15	100
10 to 6	15	50
10 to 5	15	50
11 to 7	15	100
12 to 7	15	0
12 to 3	15	100
13 to 11	15	100
14 to 12	15	100
15 to 12	15	100

Table 9.5.: Initial architectural configuration of the (PCM) modelled DeltaIoT system.

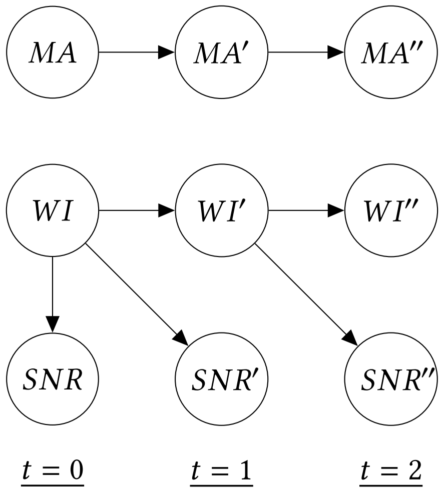
(i.e. PCM model), the model transformation representing the adaptation, the environment model (based on our *EnvDyn* metamodel), the adaptation strategy subject to evaluation and the reward function to evaluate decisions made by the strategy. Moreover, recall from section 4.3.2 that the architecture possibly influences the environmental dynamics. By default, we assume that the environment evolves independently of the architecture; however, in the context of the DeltaIoT system, this is not the case. Therefore, we briefly discuss the assumption made (and its rationale).

Initial Architecture Model We created the PCM model for describing the DeltaIoT system based on the information provided by [92]. However, the models are too large and complex to be discussed here. Therefore, we refer to [157] where all models and validation results are located. Nonetheless, Table 9.5 shows the initial architectural configuration (and adaptable elements) of the DeltaIoT system modelled with PCM.

Therefore, recall that each mote can send a packet via a communication link to another mote. Links are modelled in PCM by so-called *Linking Resources* (see [149, P.57]). Each connection is assigned a transmission power value between 0 and 15 (see the second column of Table 9.5), with high values increasing the probability that the packet will not be lost during transmission. Moreover, if a mote has two communication endpoints, the distribution factor (see the third column of Table 9.5) determines the distribution of sending the packet to the respective mote.



(a) Overview of the template variables in the plate model notation.



(b) The DBN describing the environmental dynamics unrolled for three time steps.

Figure 9.1.: Overview of the essential environmental variables of the DeltaIoT system and their relationships.

Adaptations Effectively, there are two atomic adaptations in the context of the DeltaIoT system. In the first adaptation, the transmission power of a particular mote i (or its communication link) is adjusted by adding a value, say $tp_i \in \{+1, -1, 0\}$, to the current transmission power, where the ranges are fixed, i.e. it is not possible to set $tp = 1$ if the transmission power to be adjusted is already set to 15, or $tp = -1$ if the transmission power is set to 0. The second adaptation adjusts the distribution factors of a mote which has two communication links. More specifically, a distribution factor $df_{ij} \in \{+10, -10, 0\}$ modifies the current distribution of a mote i which communicates via link (i, j) with mote j where again the ranges are fixed, i.e. if the distribution of link (i, j) is already set to 100, $df_{ij} \neq +10$ and $df_{ij} \neq -10$ if the distribution of link (i, j) is set to 0.

In the case of an adaptation, however, the system is not adapted by (e.g.) modifying the transmission power of a single mote. Instead, an adaptation is considered a composition of several atomic adaptations. That is, an adaptation consists of several atomic adaptations which are summarised as a tuple $\delta := (tp_1, \ldots, tp_i, \ldots, tp_{15}, df_{ij}, \ldots)$. The whole tuple is finally applied to adjust the network configuration of the system.

In terms of *SimExp*, we implemented a single model transformation which takes a tuple of network configuration δ and transforms the PCM model accordingly. More specifically, the transformation traverses over all motes and their links and reconfigures the network configurations according to the new values tp_i and df_{ij}.

Environment Model We modelled the environment of the DeltaIoT system by using our *EnvDyn* metamodel. Basically, for the DeltIoT systems, there are three template variables, namely mote activation MA, wireless interference WI and the signal-to-noise-ratio SNR (recall the definitions from section 5.2.3.2). Figure 9.1 provides an overview of the essential environmental variables.

	Mote 5	Mote 7	Mote 11	Mote 12	Default
Mote activation	$0.8 + \Delta$	$0.8 + \Delta$	$0.8 + \Delta$	$0.9 + \Delta$	$1 + \Delta$
Disturbance Δ	$[-0.1, 0.1]$	$[-0.2, 0.2]$	$[-0.1, 0.1]$	$[-0.1, 0.1]$	$\Delta = 0$

Table 9.6.: Overview of the individual mote activations taken from [168].

The MA template variable is instantiated for each mote in the PCM model. Similarly, the WI and SNR templates are instantiated for each wireless or communication link within the PCM model.

For the parameter setting of the variables, we considered the work of Shevtsov et al. [168] where the DeltaIoT system and simulator are used. The work contains information about the static/initial (but non-temporal) distributions of all three templates, i.e. the local CPDs of each ground Bayesian network (instantiated for each mote and wireless link). Therefore, consider Table 9.6 which depicts the individual mote activations.

For example, the table indicates for mote 5 a basis mote activation of 80% which varies w.r.t. a disturbance factor $\Delta \in [-0.1, 0.1]$. We discretised the intervals of the disturbance factors, e.g. $[-0.1, 0.1]$ is discretised to the set $\{-0.1, 0, 0.1\}$. Because we have no information about how the values are distributed, we assumed uniform distribution. The activation variations only apply to a couple of motes; the remaining motes produce sensor data with constant activation. Since we could not find any information on the temporal evolution of mote activation, we assumed that the same distribution applies to the temporal behaviour, i.e. $P(X_{MA'} \mid X_{MA})$.

For the distribution parameters of the SNR and WI templates, consider table Table 9.7. The table shows the static distribution parameters for the individual SNRs and wireless interferences. For example, for link $(2, 4)$ the basis SNR value is 8.0 which varies w.r.t. disturbance factor $\Delta \in [-5, 5]$ (i.e. wireless interference) which is added to the basis SNR value. Recall that the SNR value is determined w.r.t. the transmission power, i.e. the SNR value increases for high power values and decreases for small power values, respectively. Since all transmission powers of the initial architectural configuration are configured to the highest value (i.e. 15), we determined the basis SNR value according to values of [168] which specifies the SNR values for all possible power values. Moreover, table Table 9.7 indicates that there are merely two possible wireless interference intervals, namely $[-2, 2]$ and $[-5, 5]$. Just as for the mote activation, we discretised both intervals and assumed uniform distributions. Also, there is no information about the probabilistic temporal behaviour of WI such that we assumed the same distributions as for the static case. Finally, we defer the discussion of the stochastic evolution of the SNR values to a subsequent section that discusses the interdependency of the architectural configuration and the environment. We will see that the architectural configurations (or rather their transmission power configurations) determine how the SNR values evolve.

Adaptation Strategies In the context of DeltaIoT, we considered three adaptation strategies. For the first strategy, we consider a non-adaptive behaviour; that is to say, we reflect

Link	**SNR**	**Wireless interference Δ**
2 to 4	$8.0 + \Delta$	$[-5, 5]$
3 to 1	$7.63 + \Delta$	$[-2, 2]$
4 to 1	$3.0 + \Delta$	$[-2, 2]$
5 to 9	$2.6 + \Delta$	$[-2, 2]$
6 to 4	$7.6 + \Delta$	$[-2, 2]$
7 to 2	$5.0 + \Delta$	$[-5, 5]$
7 to 3	$0.8 + \Delta$	$[-5, 5]$
8 to 1	$7.0 + \Delta$	$[-5, 5]$
9 to 1	$4.4 + \Delta$	$[-2, 2]$
10 to 6	$4.0 + \Delta$	$[-5, 5]$
10 to 5	$6.0 + \Delta$	$[-5, 5]$
11 to 7	$6.0 + \Delta$	$[-2, 2]$
12 to 7	$-3.0 + \Delta$	$[-2, 2]$
12 to 3	$7.5 + \Delta$	$[-2, 2]$
13 to 11	$4.7 + \Delta$	$[-5, 5]$
14 to 12	$1.0 + \Delta$	$[-5, 5]$
15 to 12	$0.4 + \Delta$	$[-2, 2]$

Table 9.7.: Overview of the individual SNR and wireless interference probabilities taken from [168].

the behaviour of a static system. In terms of *SimExp* or self-adaptive systems in general, this is simply achieved by constructing an adaptation strategy, say $\pi_{\delta_\emptyset}$, which returns for all states the empty adaptation $\delta_\emptyset$ (recall from property 1), i.e. $\forall S \in \mathcal{S} : \pi_{\delta_\emptyset}(S) = \delta_\emptyset$. To put it another way, strategy $\pi_{\delta_\emptyset}$ does not adapt the system and thus simulates the behaviour of a static system. In the remainder of this chapter, we denote the strategy non-adaptive strategy.

The second adaptation strategy is taken from [92] and was presented within the context of DeltaIoT as a strategy example. Thus, we refer to this strategy as the default strategy denoted as π_D. The adaptation logic is shown in listing 9.1.

```java
public class DefaultStrategy extends
    ReconfigurationStrategy<QVToReconfiguration> {
    ...
    @Override
    protected boolean analyse(State source, SharedKnowledge knowledge) {
        for (MoteContext eachMote : getAllMoteContexts(knowledge)) {
            for (WirelessLink eachLink : eachMote.links) {
                boolean isPowerOptimal = (eachLink.SNR > 0 &&
                    eachLink.transmissionPower > 0) || (eachLink.SNR < 0 &&
                    eachLink.transmissionPower < 15)
                if (isPowerOptimal == false) {
                    return true;
                }
            }

            if (eachMote.hasTwoLinks()) {
                if (eachMote.hasUnequalTransmissionPower()) {
                    return true;
                }
            }
        }
        return false;
    }

    @Override
    protected QVToReconfiguration plan(State source, Set<QVToReconfiguration> options, SharedKnowledge knowledge) {
        DeltaIoTNetworkReconfiguration reconfiguration = getFirstElementOf(options);

        boolean powerChanging = false;
        for (MoteContext eachMote : getAllMoteContexts(knowledge)) {
            for (WirelessLink eachLink : eachMote.links) {
                powerChanging = false;
                if (eachLink.SNR > 0 && eachLink.transmissionPower > 0) {
                    decreaseTransmissionPower(eachMote.mote, eachLink, reconfiguration);
                    powerChanging = true;
                } else if (eachLink.SNR < 0 && eachLink.transmissionPower < 15) {
                    increaseTransmissionPower(eachMote.mote, eachLink, reconfiguration);
                    powerChanging = true;
                }
            }

            if (eachMote.hasTwoLinks() && powerChanging == false) {
                if (eachMote.hasUnequalTransmissionPower()) {
                    WirelessLink left = getLeftLink(eachMote);
                    WirelessLink right = getRightLink(eachMote);

                    if (left.distributionFactor == 1 && right.distributionFactor == 1) {
                        setDistributionFactorsUniformally(
                            eachMote.mote,
                            reconfiguration
                        );
                    }

                    if (left.transmissionPower > right.transmissionPower && left.distributionFactor < 1) {
                        adjustDistributionFactor(right, eachMote,
                            reconfiguration);
                    } else if (right.distributionFactor < 1) {
                        adjustDistributionFactor(left, eachMote,
                            reconfiguration);
                    }

                }
            }
        }

        return reconfiguration;
    }
}
```

Listing 9.1: Default adaptation strategy π_D taken from [92].

Note that, for simplicity, listing 9.1 only reflects a snippet of the actual adaptation logic and does not reflect all the technical details, e.g. we omit the monitor phase implementation. We thus concentrate on the pure adaptation logic; more precise implementation details can be looked up in [157]. The strategy checks in the analyse-phase whether there exists a wireless link for which the power settings are not optimal. If they are, in effect, not optimal, the plan-phase is invoked. In the plan-phase, the transmission power is adapted for links that are not optimally configured w.r.t. the current SNR value associated with the said link. Moreover, the distribution factor is increased for those links of a mote where the transmission power is higher.

The third strategy π_Q (see listing 9.2) represents a quality-driven strategy because the decision of adapting the system is exclusively determined w.r.t. the current quality objectives, i.e. packet loss and energy consumption. In the analyse-phase, strategy π_Q checks whether the quality objectives are violated, i.e. whether their values exceed some thresholds. If so, the plan-phase is executed. The strategy prioritises energy consumption; that is, during planning, it is first checked whether the energy consumption objective is violated and if so, appropriate adaptations are planned. Hereby, the transmission power for all wireless links with SNR values (i.e. SNR value ≥ 0) is decreased. Also, the distribution factor for motes with two links is decreased for those links that indicate higher transmission powers. If the packet loss objective is violated, the transmission power for all links with low SNR values (i.e. SNR value < 0) is increased and the distribution factor for motes with two links is increased for those links that indicate higher transmission powers.

Reward Function In this section, we discuss the reward function that we consider in the evaluation of the adaptation strategies. Recall that the quality objectives within the DeltaIoT system are to minimise packet loss and energy consumption. Thus, the quality attributes of packet loss and energy consumption determine the reward. As we will see in the evaluation results, however, the energy consumption values (measured in Coulomb) are significantly higher than the estimated packet loss (the percentage ratio of sent and

```java
public class QualityBasedStrategy extends
    ReconfigurationStrategy<QVToReconfiguration> {
    ...
    @Override
    protected boolean analyse(State source, SharedKnowledge knowledge) {
        return getPacketLoss(knowledge) >= PL_THRESHOLD ||
            getEnergyConsumption(knowledge) >= EC_THRESHOLD;
    }

    @Override
    protected QVToReconfiguration plan(State source, Set<QVToReconfiguration>
        options, SharedKnowledge knowledge) {
        DeltaIoTNetworkReconfiguration reconfiguration =
            getFirstElementOf(options);

        if (getEnergyConsumption(knowledge) >= EC_THRESHOLD) {
            // Decreases the transmission power for all links with high SNR,
                i.e. SNR >= 0.
            decreaseTransmissionPowerLocally(reconfiguration, knowledge);
            // Decreases the distribution factor for motes with two links, i.e.
                for the link with the higher transmission power.
            decreaseDistributionLocally(reconfiguration, knowledge);
            return reconfiguration;
        } else {
            // Increase the transmission power for all links with low SNR, i.e.
                SNR < 0.
            increaseTransmissionPowerLocally(reconfiguration, knowledge);
            // Increase the distribution factor for motes with two links (the
                link with higher transmission power).
            increaseDistributionLocally(reconfiguration, knowledge);
            return reconfiguration;
        }
    }
}
```

Listing 9.2: Quality-based adaptation strategy π_Q

received packets). That is, when calculating the reward for a given state by summing the packet loss and energy consumption values, the accumulated reward is completely dominated by the energy consumption (packet loss values are within the range of $[0, 1]$, so they do not contribute sufficiently to the total reward computation). Therefore, we normalise each packet loss and energy consumption value to the range $[0, 1]$ w.r.t. an upper and lower bound indicating the best possible and worst possible packet loss and energy consumption values, respectively. For packet loss we denote the upper bound β^+_{pl}

and lower bound β_{pl}^-; for energy consumption the bounds are denoted as β_{ec}^+ and β_{ec}^-. The normalisation function is defined as follows:

$$norm_q : I\!R \rightarrow [0,1], x \mapsto \frac{1}{\beta_q^+ - \beta_q^-} \cdot (\beta_q^+ - x) \tag{9.5}$$

Hereby, q refers to one of the quality attributes, i.e. pl or ec. The normalisation allows the comparison of energy consumption and packet loss. Therefore, we define the following reward function:

$$\begin{gathered} r_{\mathcal{S}} : \mathcal{S} \times \Delta \times \mathcal{S} \rightarrow [0,2], \\ (S, \delta, S') \mapsto (norm_{pl} \circ predict_{pl})(S') + (norm_{ec} \circ predict_{ec})(S') \end{gathered} \tag{9.6}$$

The function $predict_q : \mathcal{S} \rightarrow I\!R$ abstracts away the predicted packet loss (i.e. $predict_{pl}(S')$) and energy consumption values (i.e. $predict_{ec}(S')$) obtained from some state S' resulting from applying δ in state S. We use the model-checking tool Prism to predict both quality attributes (this is discussed later).

Interdependency Assumption of Architecture and Environment In section 4.3.2, we already illustrated the interdependency of architectural configuration and environment within the DeltaIoT system. More specifically, we pointed out that the transmission power (which is part of the architectural configuration) affects the SNR values (which are part of the environment). Consequently, the stochastic evolution of the environmental dynamics is not purely environmental-driven but also partially by the selected architectural configurations. Conventionally, we assume by default that the environmental dynamics are purely environmental-driven, i.e. independent of the architectural configuration. However, taking into account the previous discussion, this assumption no longer applies to the DeltaIoT system.

In section 6.3.2.1, we explained that interdependency assumptions are encoded within the *SimExp* framework by implementing a dedicated interface. For the implementation of the interdependency assumption, only the stochastic evolution of the SNR values must be considered as the other environmental variables (i.e. mote activation and wireless interference) are purely environmental-driven. As mentioned earlier, the work of Shevtsov et al. [168] already provides information about the DeltaIoT system. Moreover, they provide a complete table which maps transmission power values to SNR values for each wireless link. The table can be used to implement the interdependency assumption, i.e. by calculating the next SNR value w.r.t. the current transmission power associated with a wireless link. However, while exploring the code of the DeltaIoT simulator, we found a set of linear equations of the form

$$SNR_{ij} : \{0, 1, \ldots, 15\} \rightarrow I\!R, x \mapsto m_{ij} \cdot x + c_{ij} \tag{9.7}$$

Here, m_{ij} is some multiplier and c_{ij} is some constant dependent on the wireless link (i, j) for which the SNR is to be calculated. The equations calculate for a given link (i, j) w.r.t. the current transmission power x the new SNR value. Therefore, we instantiate each equation for each wireless link (i, j) and calculate the new SNR value of each link based on the transmission power extracted from the current architectural configuration.

Prediction of quality attributes In section 6.3.2.2, we explained how to extend the *SimExp* framework to account for other prediction tools which are not part of the simulation and analysis tool repertoire of the Palladio ecosystem. For the prediction of packet loss and energy consumption, we extended the *SimExp* framework by the model-checking tool Prism [109]. Weyns and Iftikhar [207] used Prism in the context of DeltaIoT to predict packet loss and energy consumption. We reuse their Prism artefacts [140] (i.e. property and module files) for prediction. Hereby, we use the Prism module files as parameterised templates that are complemented by values derived from a given self-adaptive system state S. More specifically, for a self-adaptive system state $S := (C, E)$, the prediction procedure extracts all relevant parameters of the architectural configuration C (i.e. the PCM model) and environmental state E and inserts them in the parameterised template of the Prism module. After completion, the module file is forwarded to the Prism tool for prediction.

9.2.1.2. Experiment Setup

For the validation, we evaluate the quality of the previously presented adaptation strategies with both, the DeltaIoT simulator and our *SimExp* method. Afterwards, we compare the rank of the strategies w.r.t. both evaluation approaches. To validate the appropriateness of our MDP-based approach, we expect the evaluation results of the strategies with our *SimExp* method to produce the same rank as the DeltaIoT simulator.

Therefore, we complemented the DeltaIoT simulator by implementing the quality-based strategy π_Q as shown in listing 9.2. The default strategy π_D was already implemented and provided by the simulator. Also, the DeltaIoT simulator allows the evaluation of non-adaptive and thus static systems w.r.t. some specified initial configuration. We configured the simulator by defining the same initial architecture as for the instantiation of the *SimExp* method before (see 9.5). Moreover, we used the same set of SNR equations (recall 9.7) already defined by the simulator. We configured the wireless interferences and mote activation probabilities as shown in Table 9.6 and Table 9.7. For transparency, we initialised a Git repository in the code base of the DeltaIoT simulator. Thus, all configurations and settings made are documented and can be understood. Moreover, within the DeltaIot simulator, there are various configurable simulation parameters. We adopted all default simulation parameters predefined by the simulator (some of them are relevant for the Prism files for quality prediction). Table 9.8 provides an overview. Only the number of trajectories was defined by us. By default, the DeltaIoT simulator performs only a single simulation of an entire trajectory. To achieve a particular statistical certainty, we have increased the number to 10 runs. One could argue that even 10 runs might not be sufficient; however, we had efficiency problems with some energy consumption predictions with Prism (up to several minutes for a single prediction). Therefore, we limited the number of runs to 10. Nonetheless, we will see later that the results are stable in terms of statistical certainty.

Name	Value	Description
Spreading factor	1.158	Number of bits encoded of a transmitted packet [92].
Reception cost	14.2	-
Reception time	4	-
Coulomb	1000.0	The unit for measuring the energy consumption.
Number of simulations	96	Number of simulations per trajectory (compare with horizon from *SimExp*).
Number of trajectories	10	Number of trajectories to simulate.

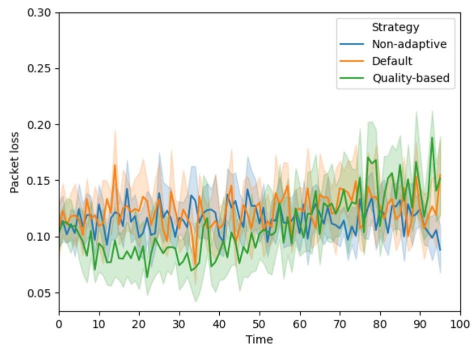
Table 9.8.: Overview of the parameter setting for the DeltaIoT case study system. For some parameters, no description could be found.

9.2.1.3. Experiment Results

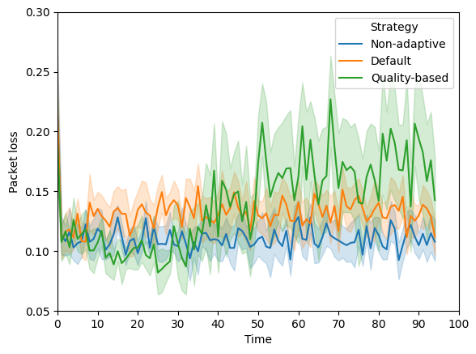
Finally, we present the results of the validation. As described in the last section, we evaluated all three strategies (i.e. non-adaptive strategy $\pi_{\delta_\emptyset}$, default strategy π_D and the quality-based strategy π_Q) with our *SimExp* method and the DeltaIoT simulator. The primary objective of each strategy is to minimise the packet loss and energy consumption of the system.

The results of the evaluation runs of *SimExp* and the DeltaIoT simulator are depicted on Figure 9.2 regarding packet loss and Figure 9.3 for energy consumption. Note that in the following all line plots indicate the mean and 95% confidence interval of the considered quality attribute. In general, the y-axes of the line graphs show the packet loss/energy consumption of each strategy; the x-axes indicate the discrete time steps in both cases.

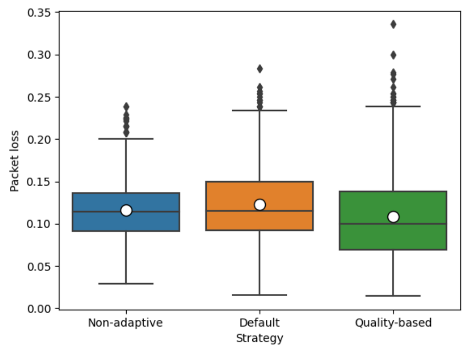
In terms of Figure 9.2, the left-hand side (i.e. Figure 9.2a and Figure 9.2c) shows the results of the DeltaIoT simulator. More specifically, it depicts the line plot and box plot of the packet loss achieved by the strategies. The right-hand side shows the line plot and box plot of the *SimExp* evaluation of the strategies. It can be seen that the evaluation results for the non-adaptive strategy $\pi_{\delta_\emptyset}$ and default strategy π_D indicate the same results, i.e. π_D causes higher packet loss than $\pi_{\delta_\emptyset}$. Also, the prediction accuracy is remarkably precise, i.e. mean value and median are not deviating significantly. This fact is visualized in the box plots more clearly. However, the quality-based strategy π_Q is slightly deviating. When considering the mean values of the *SimExp* results, we observe the strategy ordering $\pi_{\delta_\emptyset} < \pi_D < \pi_Q$ w.r.t. packet loss. In contrast, the ordering of the DeltaIoT results are given as follows: $\pi_Q < \pi_{\delta_\emptyset} < \pi_D$. Nonetheless, the mean values of the packet loss of both, DeltaIoT and *SimExp* are fairly close to each other, i.e. approximately in a range of $[0.1, 0.15]$. Therefore, it can be argued that all strategies achieve comparable packet loss results such that the packet loss quality objective is not a crucial factor when considering the overall quality of each strategy (i.e. taking into account the energy consumption as well).



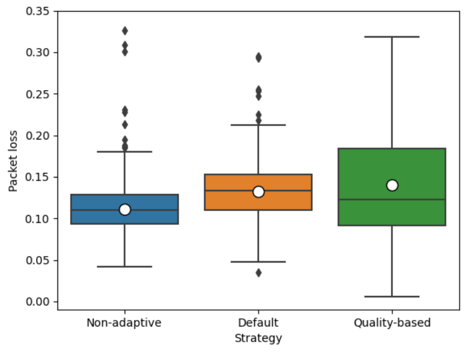
(a) DeltaIoT results for packet loss



(b) *SimExp* results for packet loss



(c) DeltaIoT results for packet loss

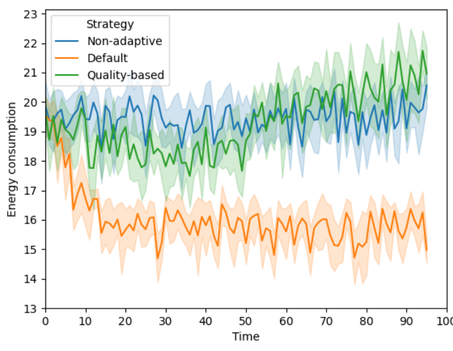


(d) *SimExp* results for packet loss

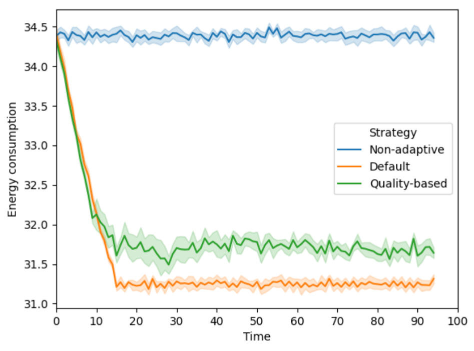
Figure 9.2.: Comparison of DeltaIoT and *SimExp* results considering the line plots and box plots of the packet loss. In addition to the median value of the box plots, the white dots indicate the mean values.

The energy consumption results (see Figure 9.3) are not as accurate as the packet loss results. In terms of prediction accuracy, the pure energy consumption predictions are quite deviating. The prediction range of the DeltaIoT simulator is approximately between $[14, 23]$; the prediction range of the *SimExp* results is approximately between $[31, 34.5]$. However, this deviation results from the prediction inaccuracy of Prism and is not a result of *SimExp* (we will come back to this later). Nonetheless, the rank of the strategies in terms of energy consumption is preserved in both evaluations, i.e. $\pi_{\delta_\emptyset}$ causes the highest mean energy consumption followed by π_Q and π_D which performs best.

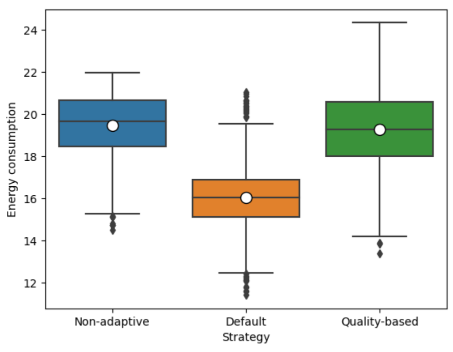
Recall that the quality-based strategy π_Q depends on some packet loss and energy consumption thresholds on which decisions are made (see Listing 9.2). Generally, the thresholds are determined w.r.t. the quality requirements. For the experiments, we configured the packet loss threshold to be $\alpha = 0.1$. Regarding energy consumption, we have chosen two distinct thresholds. As discussed, the energy consumption predictions of DeltaIoT and *SimExp* are deviating to an extent that makes it not possible to use the same threshold; this would result in completely different results. Therefore, we have selected two separate thresholds ($\beta = 18$ for DeltaIoT and $\beta = 32$ for *SimExp*) w.r.t. the energy consumption prediction ranges such that the results are comparable. We explain the precise determination of the thresholds later in this section. The thresholds are summarised in table Table 9.9.



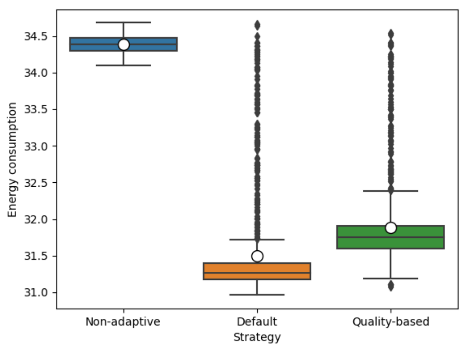
(a) DeltaIoT results for energy consumption



(b) *SimExp* results for energy consumption



(c) DeltaIoT results for energy consumption



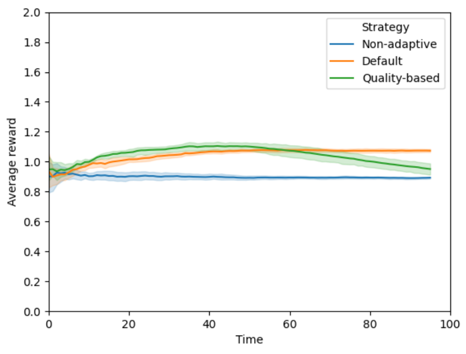
(d) *SimExp* results for energy consumption

Figure 9.3.: Comparison of DeltaIoT and *SimExp* results considering the line plots and box plots of the energy consumption. In addition to the median value of the box plots, the white dots indicate the mean values.

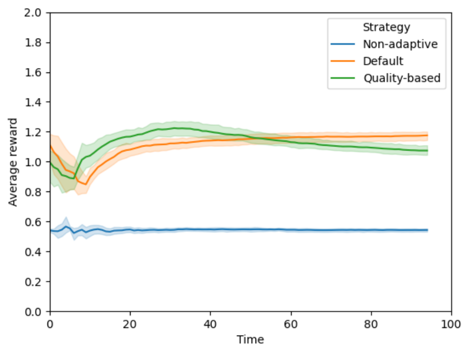
DeltaIoT					
Packet loss			**Energy consumption**		
$\alpha^+ = 0.2$	$\alpha = 0.1$	$\alpha^- = 0.025$	$\beta^+ = 26$	$\beta = 18$	$\beta^- = 10$
SimExp					
Packet loss			**Energy con.**		
$\alpha^+ = 0.2$	$\alpha = 0.1$	$\alpha^- = 0.025$	$\beta^+ = 34.5$	$\beta = 32$	$\beta^- = 30.5$

Table 9.9.: Overview of the distinct thresholds and bounds used in the quality-based strategy and reward function. A threshold value superscripted with + or − denotes the upper and lower bounds of the quality attribute in question.

To compare the results of the reward function for both simulations, we applied the reward function to the DeltaIoT simulator as well. More specifically, we calculated the reward for each time instance where a packet loss and energy consumption pair have been generated. We repeated that process for each strategy. The results of both reward functions are shown in Figure 9.4.



(a) Line plot of the accumulated reward of DeltaIoT



(b) Line plot of the accumulated reward of *SimExp*

Figure 9.4.: Comparison of the accumulated rewards of DeltaIoT and *SimExp*.

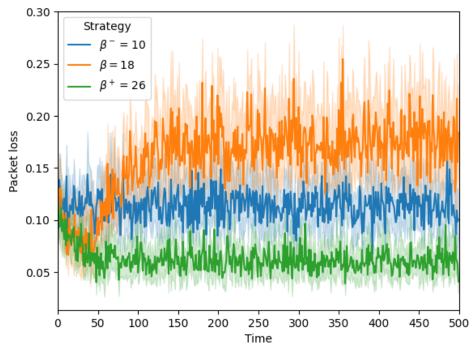
$\mathbb{E}_\pi[X_{G_0}]$		
$\pi_{\delta_\emptyset} = 51.6$	$\pi_Q = 100.7$	$\pi_D = 109.6$

Table 9.10.: Overview of the expected rewards of each strategy for the DeltaIoT case study.

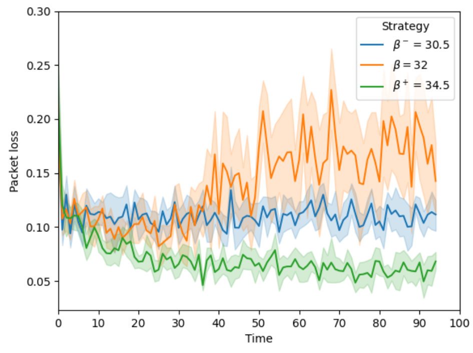
The figure depicts the average accumulated rewards. Note that Table 9.9 depicts the upper and lower bounds used to calculate the reward for each strategy. Again, we have chosen different upper and lower bounds for energy consumption because the predictions in DeltaIoT and *SimExp* differ too much. For this purpose, we evaluated the rewards of each simulated trajectory by considering the following function: $accum_\pi(N) = \frac{1}{N}\sum_{i=1}^{N} r_i$. The function accumulates the rewards of a strategy π to a given time instance N and calculates the mean value. Figure 9.4 depicts the averaged accumulation function for the considered time steps (i.e. $0, 1, 2, \ldots, 95$) of 10 runs. It can be seen that both functions indicate the same behaviour. Moreover, in terms of the strategies, the order $accum_{\pi_{\delta_\emptyset}}(96) < accum_{\pi_Q}(96) < accum_{\pi_D}(96)$ is in both results the same. Complementary to the results from Figure 9.4, we depict the estimated reward (i.e. $\mathbb{E}_\pi[X_{G_0}]$ from equation (6.10)) in Table 9.10. Note that we could not estimate the expected reward in the case of DeltaIoT because we could only extract the quality values (i.e. packet loss and energy consumption).

Although the averaged accumulated rewards of strategies $\pi_{\delta_\emptyset}$ and π_D converge to a fixed reward value for both, DeltaIoT and *SimExp*, this is not the case for strategy π_Q. Furthermore, when considering again the packet loss and energy consumption of DeltaIoT, it can be seen that the curves of both quality objectives tend to rise instead of converging towards a fixed value (in contrast to the other strategies). Therefore, we repeated the DeltaIoT simulation experiment by increasing the simulation horizon (i.e. the number of simulations which were initially set to 96) to 500 runs. The results are depicted by the orange curve on the left-hand side of Figure 9.5.

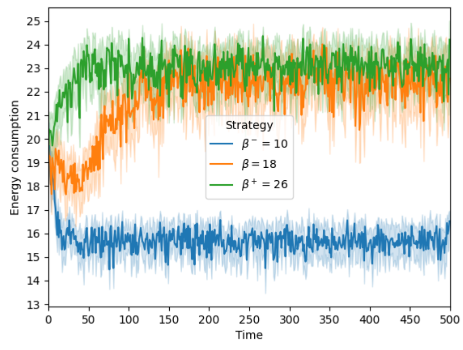
If we ignore for one moment the other curves depicted and simply focusing the orange curve (again, indicating the results of the quality-based strategy generated by the DeltaIoT simulator for 500 simulation runs), we can see that the packet loss and energy consumption



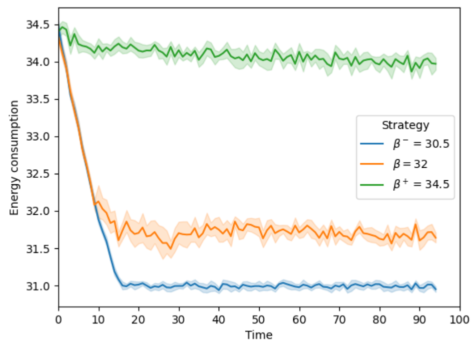
(a) DeltaIoT results for packet loss



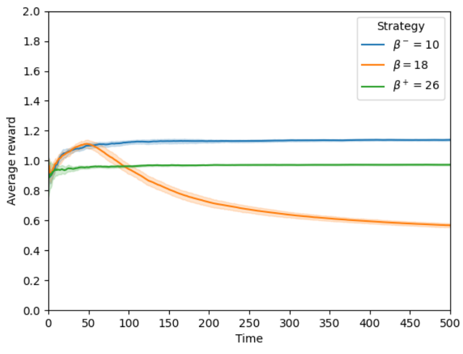
(b) *SimExp* results for packet loss



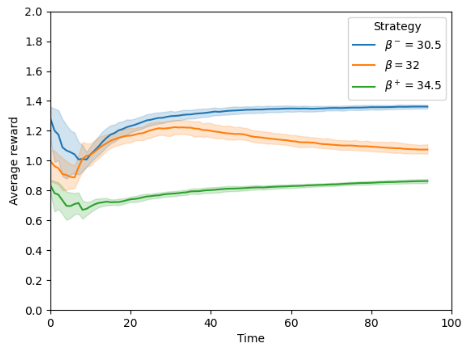
(c) DeltaIoT results for energy consumption



(d) *SimExp* results for energy consumption



(e) The accumulated reward of DeltaIoT



(f) The accumulated reward of *SimExp*

Figure 9.5.: Comparison of DeltaIoT and *SimExp* packet loss, energy consumption and accumulated rewards results of strategy π_Q taking into account the bounds β^-, β and β^+.

converge to a completely different value as suggested for simulation horizon of 96. The right-hand side of Figure 9.5 shows the quality-based strategy (also the orange curve) generated by *SimExp* for 96 simulations per trajectory (the results are, in effect, copied from the figures before). At this point, it can be seen that the packet loss (which seems to diverge a little when comparing DeltaIoT with *SimExp* for 96 runs w.r.t. strategy π_Q) now becomes accurate again, as the packet loss of strategy π_Q for the DeltaIoT simulator clearly converges to the highest packet loss value of all strategies. On the other hand, however, the energy consumption of the DeltaIoT results converges towards a value ≈ 22.5 which is again the highest energy consumption value of all three strategies (see Figure 9.3c). This is in contradiction with our results where strategy π_Q causes the second-highest energy consumption.

One reason why the results in DeltaIoT and *SimExp* differ for the strategy π_Q is that the strategy makes its decisions based on the predicted energy consumption. That is, if the energy consumption predictions are diverging, also the behaviour of the adaptation logic is likely to diverge. In addition, energy consumption is the primary quality attribute, i.e. it is first checked whether the required energy consumption level is met, and if not, appropriate countermeasures are taken; only if the energy consumption objective is satisfied, the packet loss is checked. Although the evaluation results of the strategies $\pi_{\delta_\emptyset}$ and π_D (when compared to the evaluation results of DeltaIoT) are quite precise, and we already observed prediction deviations in terms of energy consumption for Prism and the DeltaIoT simulator, we conducted further experiments to show the appropriateness of our MDP-based *SimExp* method.

Therefore, we adjusted the thresholds of the quality-based strategy by considering upper and lower bounds. The upper and lower bounds correspond to the thresholds at which the strategy π_Q reaches the maximum and minimum energy consumption; that is, the same maximum and minimum energy consumption is observed for each threshold above and below the thresholds, respectively. Hereby, we investigated the bounds in both cases (i.e. for DeltaIoT and *SimExp*) by experimental testing. In terms of *SimExp* the lower bound corresponds to $\beta^- = 30.5$ and the upper bound $\beta^+ = 34.5$; for the DeltaIoT simulator the lower bound is $\beta^- = 10$ and $\beta^+ = 26$. The thresholds and bounds are summarised in Table 9.9. Note again that for any bound lower than β^- or higher than β^+, the DeltaIoT simulator indicates the same adaptation behaviour because β^- and β^+ refer to lower and upper energy consumption bounds the system can never achieve by adaptation.

On this basis, we determined the thresholds of the strategy π_Q for the initial validation. Since the ranges of energy consumption differ, we had to choose a threshold β that makes both strategies π_Q comparable when evaluated in the context of DeltaIoT and *SimExp*. Therefore, we chose the mean value of the respective upper and lower bounds.

The rationale for evaluating strategy π_Q for lower and upper bounds is that we can expect a similar adaptation behaviour. For the upper bound case, for example, the energy consumption is always satisfactory. Consequently, fewer adaptations are required. The same applies to packet loss. Since the initial configuration already provides for high transmission powers (i.e. each mote is configured with the highest possible transmission power) and no further adaptations to reduce the transmission power are expected, the packet loss is also likely to be below the predefined threshold (recall that high transmission powers reduce the probability of packet loss). For the lower bound case, however, we expect the very opposite behaviour. In this case, the energy consumption is not satisfied at any time. Therefore, strategy π_Q adapts the system as much as possible to satisfy the energy consumption objective w.r.t. its implemented adaptation logic. Therefore, we expect low energy consumption but increased packet loss because minimising energy consumption is always at the cost of packet loss.

We simulated strategy π_Q for both, DeltaIoT and *SimExp*, with different bounds. The results are depicted on Figure 9.5 where the results for π_Q with upper bound β^+ refer to the green curves and the results for π_Q with lower bound β^- refer to the blue curves. The results show that strategy π_Q behaves for both bounds as expected. When we compare the DeltaIoT results with our *SimExp* results, we also notice that the curves for packet loss and energy consumption have the same shape; except the energy consumption curve for mean bound β. Consequently, the average accumulated rewards are deviating from the strategy π_Q with mean bound β. Nonetheless, we showed that the evaluation results of *SimExp* for strategy π_Q are accurate when considering bounds (or thresholds) which dictate a certain adaptation behaviour. This adaptation behaviour could be observed in the DeltaIoT simulator and also by our *SimExp* method. Therefore, we conclude that *SimExp* still gives accurate results w.r.t. the strategy π_Q (although the results for the mean bound β differed). In addition, we consider the evaluation deviation for the mean bound β as a result of the deviating energy consumption of Prism and the DeltaIoT simulator we observed before.

Complementary to the discussed results of this section, in appendix A the average architectural configurations to which each strategy converges after a specific length of sampled states per trajectory. Here, we compare the predicted configuration of *SimExp* with the actual configurations that one observed after applying the DeltaIoT simulator. We consider it remarkable that the prediction results regarding the δ_D strategy are very precise and hardly differ from each other. The same applies to the strategy π_Q with upper and lower bounds where the results are also fairly close. Only for the strategy π_Q with mean bound β there are deviations in the configurations for the very reasons that we discussed in this section before. However, the configurations do not deviate significantly either.

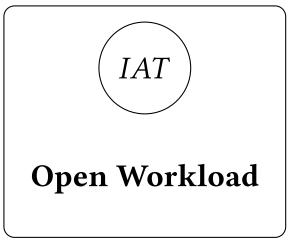
9.2.2. Load Balancing

After validating the *SimExp* method for the DeltaIoT case study system, we consider the load balancer case study system in this section. Moreover, in the context of the case study, we use SimuLizar as a baseline for *SimExp*. The section is organised as for the DeltaIoT case study before. We start to explain how the *SimExp* method is instantiated for the load balancer system. Then, we outline the experiment setup and discuss the results afterwards. The validation artefacts and results can be found at [157].

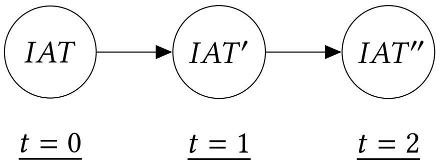
9.2.2.1. Instantiation of *SimExp*

In the following, we outline the instantiation of the *SimExp* method regarding the load balancer system by following the same structure as in the last section.

Initial Architecture Model The load balancer system has already been discussed in section 1.5.1. Therefore, we do not discuss the details of the system (and its software architecture) again but refer to said section. Recall that the load balancing system includes two application servers that are preceded by a load balancing component to balance the incoming load.



(a) Overview of the template variable in the plate model notation.



(b) The DBN describing the environmental dynamics unrolled for three time steps.

Figure 9.6.: Overview of the only environmental variable of the load balancer system: The inter-arrival time.

The load balancing component is controlled by a distribution factor which determines the percentage of load distributed to the respective application servers. For example, if the distribution factor is set to 0.7, 70 percent is distributed to application server 1 and the remaining 30 percent to application server 2. The distribution factor is adapted by a self-adaptive system to deal with varying loads. Initially, however, we assume that the distribution factor is set to 1.0, i.e. all incoming load is initially forwarded to application server 1.

Adaptations We consider two adaptations, namely *Outsource* and *Scale in*. Both adaptations are concerned with adapting the distribution factor w.r.t. some step size. More specifically, outsource refers to the case where the distribution factor is adapted in a sense such that incoming load is distributed to both servers more strongly. For example, if the step size is set to 0.1 and suppose the system is in the initial architecture configuration, outsource results in adapting the distribution factor to 0.9. Afterwards, it is possible to outsource again until the maximum possible distribution, i.e. 0.5, is reached.

Scale in, on the contrary, reverses outsource adaptations w.r.t. step size (to reduce the number of resources used and thus the costs). This means if the distribution factor is set to 0.5, scale in results in a distribution factor of 0.6. Furthermore, scale in can be repeated until the distribution factor is set to 1.0 again.

Environment Model The environment model and its corresponding environmental variables of the load balancing system are trivial. In principle, the environment includes only a single variable, namely the inter-arrival time. The inter-arrival time determines the time between two user arrivals (see section 2.3.1.1). Whenever the inter-arrival time decreases, the incoming load of the system increases because of the rising number of incoming users per time unit. The environment model and its modelled environmental dynamics (described by our *EnvDyn* metamodel) are depicted on Figure 9.6.

Note that since we use PCM as ADL to describe software architectures, the *IAT* template variable is instantiated to the open workload object (which is the container object for describing the inter-arrival rates of users) of the PCM model. For the load balancing system, there is only a single usage scenario defined with a single open workload object. Thus, the template *IAT* is instantiated once. We discuss the concrete probability distributions of the environment model in section 9.2.2.2.

Adaptation Strategies We consider three adaptation strategies in the context of the load balancer case study system. As a first adaptation strategy, we again consider the non-adaptive strategy $\pi_{\delta_\emptyset}$ (recall from section 9.2.1) which reflects the behaviour of a static software system.

As a second adaptation strategy, we consider an adaptation strategy that outsources the system when the response time exceeds a certain threshold ε^+ or scales in when the response time falls below a second threshold ε^-. The step size for adapting the distribution factor of this strategy is 0.1; therefore, we refer to the strategy as $\pi_{0.1}$ in the following.

As a third adaptation strategy (denoted $\pi_{0.2}$), we consider the same adaptation logic as discussed for strategy $\pi_{0.1}$ but consider a different step size, i.e. 0.2. Therefore, strategy $\pi_{0.2}$ reflects a design decision within the adaptation strategy family where only the step size is varied.

For both strategies, we used the fixed thresholds $\varepsilon^+ = 2.0$ and $\varepsilon^- = 0.3$ where the thresholds are specified in seconds.

Reward Function Again, we consider a reward function that reflects the extent to which the quality objectives are satisfied. In terms of the load balancing system, the primary quality objective is performance or more specifically the response time of the system. Therefore, the reward function simply returns a positive reward (i.e. +1) if the current response time is below ε^+ and a negative reward (i.e. -1) otherwise.

$$r_{\mathcal{S}} : \mathcal{S} \times \Delta \times \mathcal{S} \rightarrow \{+1, -1\}, (S, \delta, S') \mapsto (-1)^{\mathbb{1}_{predict_{rt}(S') > \varepsilon^+}} \tag{9.8}$$

Again, we use the function $predict_{rt}$ to abstract (in this case) the response time prediction for state S'; we discuss the used prediction tool later.

Interdependency Assumption of Architecture and Environment We assumed no interdependency between the system or architectural configuration and the environment, i.e. the default assumption is made (recall independence assumption (6.9)). In other words, we assume that the stochastic dynamics of the self-adaptive system are purely environmental-driven.

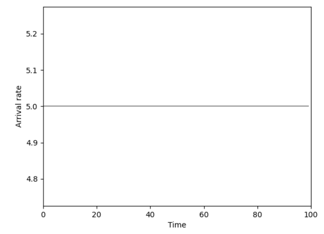
Prediction of quality attributes For the prediction of the quality attributes (i.e. the response time of the system), we use SimuLizar. It is important to understand the difference in how we use SimuLizar within our *SimExp* method itself and for comparison. Within *SimExp*, we use the simulation capabilities of SimuLizar for static systems. That is, we predict the response time of the current architectural configuration and environmental state. In this case, the environmental state consists of a single environmental variable (i.e. the inter-arrival time *IAT*) holding a single inter-arrival time value. The inter-arrival time is synchronised with the usage model of the current PCM model (describing the architectural configuration) such that SimuLizar is applied to predict the response time. Thus, we obtain a response time prediction by considering the said PCM model as a static system and the inter-arrival time as a fixed and non-varying value. This is repeated for any state sampled by *SimExp* (based on the sampled environment state or inter-arrival time and the adaptation applied by the strategy). In contrast, we compare the results of *SimExp* with the simulation results of SimuLizar by considering its self-adaptive system simulation capabilities.

9.2.2.2. Experiment Setup

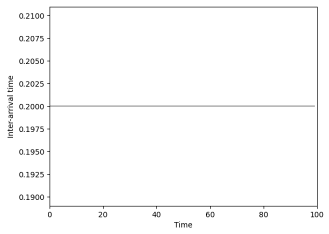
Just like the DeltaIoT system, we compare the evaluation results of SimuLizar with the evaluation results of *SimExp* w.r.t. the three adaptation strategies π_{δ_0}, $\pi_{0.1}$ and $\pi_{0.2}$ in the context of the load balancer case study system. Therefore, we implemented the strategies for both simulators. Also, to validate the appropriateness of our MDP-based approach, we expect the evaluation results of the strategies with our MDP-based *SimExp* method to produce the same rank as SimuLizar.

However, SimuLizar is effectively used to evaluate adaptation strategies for a collection of predefined scenarios. Such scenarios are usually modelled by an approach called *Usage Evolution* [31]. Usage evolution allows the modelling of varying performance-relevant factors over time. Such a factor refers to the arrival rate, i.e. the number of users arrivals within a time unit. An example usage evolution of the arrival rate is depicted on Figure 9.7g which models a scenario where the arrival rate increases to a maximum and decreases afterwards again, i.e. a peak load scenario with time range $[0, 100]$. Usage evolution is used in conjunction with Palladio in that during the simulation of SimuLizar, the inter-arrival time of PCM's usage model is periodically synchronised with usage evolution. The inter-arrival time *iat* is calculated by dividing 1 with the current arrival rate *ar*, i.e. $iat = 1/ar$.

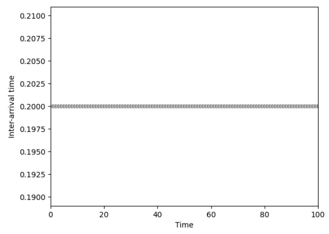
This is, however, in contrast with *SimExp* which evaluates adaptation strategies in terms of numerous sampled trajectories of the environmental dynamics. A naive approach for comparing with SimuLizar could be to sample a representative number of trajectories which are transformed to an equivalent set of usage evolutions for evaluating adaptation strategies in both contexts. However, this is impractical and would require an exhaustive effort. Conversely, we can define a set of usage evolutions and transform them into individual DBNs where each DBN represents a deterministic trajectory. For example, consider again the usage evolution describing the arrival rates of a peak load scenario depicted on Figure 9.7g. In the first step, we can transform the arrival rates such that we obtain the respective inter-arrival times, see Figure 9.7h. The inter-arrival times are then discretised at equidistant intervals, say $0, 1, 2, 3, \dots, 100$. After discretisation, we obtain a discrete function $f : \{0, 1, \dots, 100\} \longrightarrow \mathbb{R}$, as depicted on Figure 9.7i. From function f, we generate a DBN with static distribution $P(X_{IAT})$ and dynamic distribution $P(X_{IAT_{t+1}} \mid X_{IAT_t})$. Let $x, x' \in Val(X_{IAT})$, the action of static distribution $P(X_{IAT})$ on x is $x \mapsto \mathbb{1}_{f(0)=x}$ and the



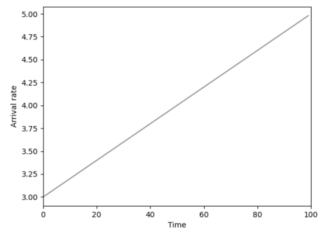
(a) Usage evolution with constant arrival rates.



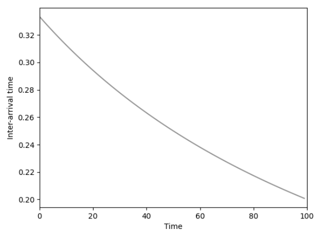
(b) Inter-arrival time at constant evolution.



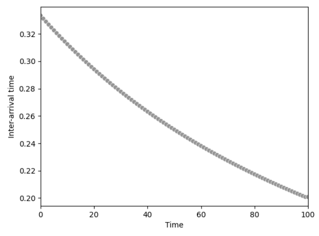
(c) Discretized inter-arrival time at constant evolution.



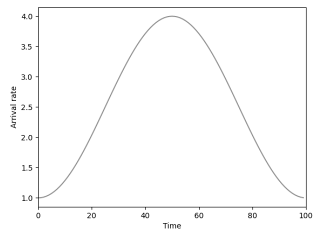
(d) Usage evolution with linear arrival rates.



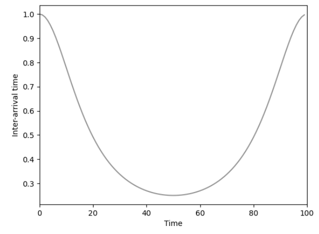
(e) Inter-arrival time at linear evolution.



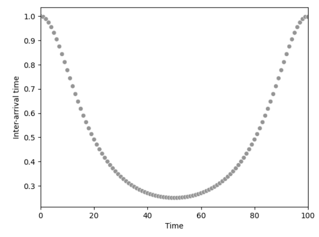
(f) Discretized inter-arrival time at linear evolution.



(g) Usage evolution with arrival rate peak.



(h) Inter-arrival time with peak.



(i) Discretized inter-arrival time with peak.

Figure 9.7.: Comparison of the usage evolutions in SimuLizar and *SimExp*.

action of dynamic distribution $P(X_{IAT_{t+1}} \mid X_{IAT_t})$ on x and x' is $(x, x') \mapsto \mathbb{1}_{f(t+1)=x'} \cdot \mathbb{1}_{f(t)=x}$. We repeated this transformation for two further usage evolutions. The usage evolutions and their discretised versions for *SimExp* are depicted on Figure 9.7.

The constant usage evolution on Figure 9.7a is selected in such a way that the initial architectural configuration is not able to deal with the arrival rate in the long run such that adaptation is required. The second usage evolution indicates a continuously increasing arrival rate which demands adaptation as well. For each strategy, we can now evaluate how each strategy performs in the three usage evolutions for SimuLizar and *SimExp* where we expect the same strategy rank.

Finally, Table 9.11 provides an overview of the experiment parameters. As already mentioned, we defined the usage evolution scenarios over the time range $[0, 100]$; thus, the simulation time of SimuLizar is adjusted to 100 and the number of trajectories in terms of *SimExp*. Because the simulation of self-adaptive systems in SimuLizar is nondeterministic, we repeated the experiments ten times to maintain statistical certainty (we see in the next section that there are no significant deviations). In terms of *SimExp*, the only

Name	Value	Description
ε^+	2.0	Upper threshold of the response time (specified in seconds) which must not be exceeded.
ε^-	0.3	Lower threshold of the response time (specified in seconds) which must not be fallen short of.
Simulation time	100	Simulated time units of SimuLizar.
Repetitions	10	Number of experiment repetitions for SimuLizar.
Number of simulations	100	Number of simulations per trajectory (compare with horizon from *SimExp*).

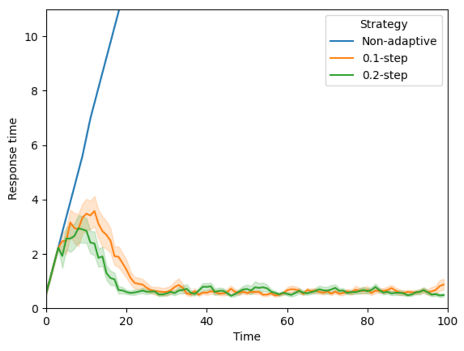
Table 9.11.: Overview of the parameter setting for the load balancer case study system.

non-deterministic component refers to the environmental dynamics. However, since we generated deterministic DBNs, the entire sampling process becomes deterministic such that no further repetitions are required.

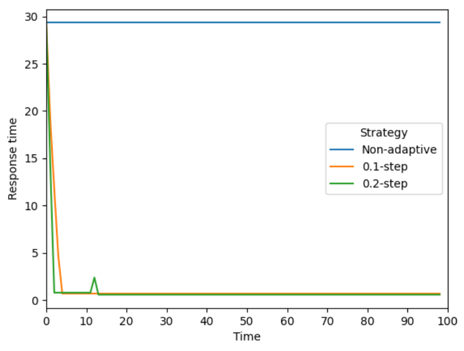
9.2.2.3. Experiment Results

Finally, we present the results of the validation. As described in the last section, we evaluated all three strategies (i.e. non-adaptive strategy $\pi_{\delta_\emptyset}$, $\pi_{0.1}$ and $\pi_{0.2}$) with our *SimExp* method and SimuLizar. The primary objective of each strategy is to keep the system responsive in the presence of varying workloads (or arrival rates). Again, all line plots indicate the mean and 95% confidence interval.

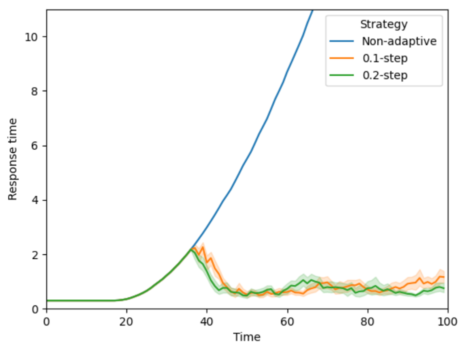
Figure 9.8 depicts the evaluation results of all strategies for SimuLizar and *SimExp* where the y-axis shows the response time of each strategy and the x-axis depicts the simulation time. At this point, it is important to note that the prediction results of SimuLizar and *SimExp* are not directly comparable because SimuLizar conducts a continuous simulation over the time range $[0, 100]$ and *SimExp* samples 100 discrete states in which each state is individually simulated. For instance, consider the results of strategy $\pi_{\delta_\emptyset}$ for SimuLizar Figure 9.8a and *SimExp* Figure 9.8b with constant usage evolution (in each case the blue curve). Since strategy $\pi_{\delta_\emptyset}$ does not apply any adaptation, the system remains in its initial architectural configuration. The initial configuration, however, is not able to deal with the constant arrival rate (or inter-arrival times) such that the response time increases linearly over time. This situation is perfectly reflected in terms of SimuLizar and its continuous simulation (see Figure 9.8a). In contrast, for strategy $\pi_{\delta_\emptyset}$ *SimExp* samples 100 states which constitute (in this case) the deterministic dynamics of the self-adaptive system where for each state the architectural configuration (i.e. the initial configuration) and the environmental states (the constant inter-arrival time) are the same. However, each state is simulated individually by SimuLizar such that we obtain for each simulation the same linearly increasing response times as SimuLizar produced in Figure 9.8a. Internally,



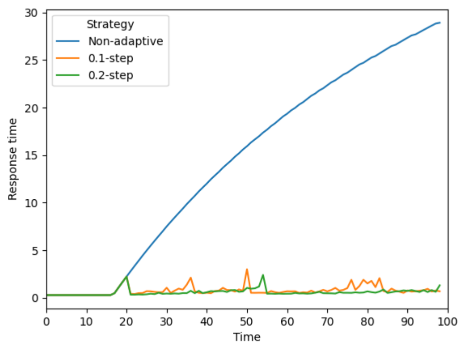
(a) Simulizar results with constant evolution.



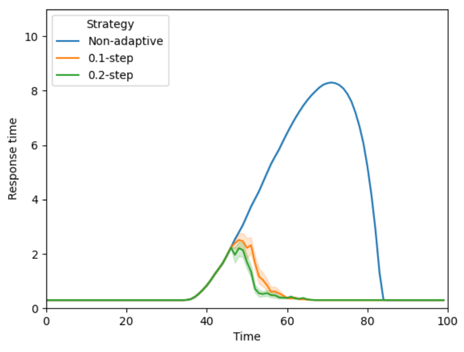
(b) *SimExp* results with constant evolution.



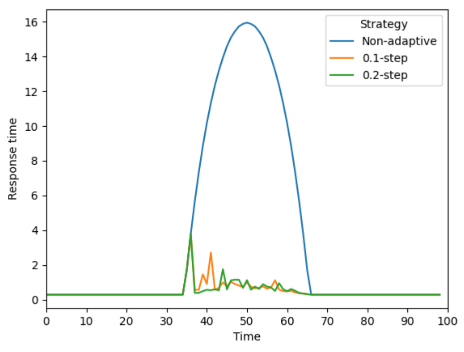
(c) Simulizar results with linear evolution.



(d) *SimExp* results with linear evolution.



(e) Simulizar results with peak.



(f) *SimExp* results with peak.

Figure 9.8.: Comparison of SimuLizar and SimExp results. The left-hand side enumerates all SimuLizar results w.r.t. the individual usage evolutions and the right-hand side the *SimExp* results, respectively.

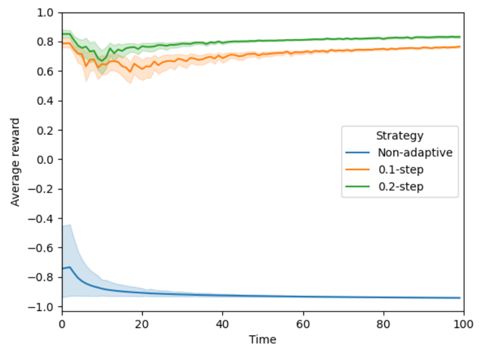
SimExp reduces the response times to a single value, i.e. it calculates the average response time which is approximately ≈ 29.33 (see Figure 9.8b). This behaviour applies to all *SimExp* results and explains why the corresponding result curves tend to be discontinuous in contrast to the SimuLizar curves. Although the prediction results of SimuLizar and *SimExp* are not directly comparable, we still expect the prediction results of *SimExp* to show the same characteristics as the prediction results of SimuLizar, e.g. strategies that show high response times in SimuLizar should also show high response times in *SimExp* and vice versa. From this, we can conclude that our *SimExp* framework allows the evaluation of adaptation strategies (w.r.t. some usage scenarios) in the same way as SimuLizar. If we look at Figure 9.8 we can see that the results of *SimExp* reflect the same as the results of SimuLizar. More specifically, for each usage evolution scenario, strategy $\pi_{\delta_\emptyset}$ indicates high response times; the strategies $\pi_{0.1}$ and $\pi_{0.2}$ performs better in which $\pi_{0.2}$ achieves the best results.

The strategy rank is especially observable in the averaged accumulated rewards as shown in Figure 9.9. We applied the same reward function that we used for *SimExp* to the SimuLizar results (see left side of Figure 9.9); the rewards of *SimExp* results are depicted on the right side. Hereby, we again accumulated and averaged the generated rewards, i.e. $accum_\pi(N) = \frac{1}{N}\sum_{i=1}^{N} r_i$. It can be seen that for $N = 100$ the ranks of strategies are the same. More specifically, for all three usage evolution scenarios, SimuLizar evaluates the strategy rank: $accum_{\pi_{\delta_\emptyset}}(100) < accum_{\pi_{0.1}}(100) < accum_{\pi_{0.2}}(100)$. The very same rank is observed in the *SimExp* results which are in line with our expectations.

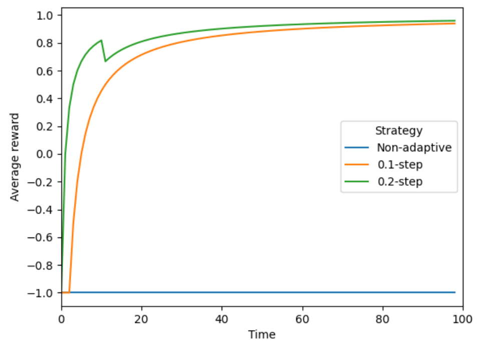
Although the averaged accumulated rewards are fairly accurate, we observe that some characteristics of the predicted response time trajectories of SimuLizar are differently predicted in *SimExp*. Regarding strategy $\pi_{\delta_\emptyset}$, for example, we observe in the *SimExp* results (although capturing the high response times associated with $\pi_{\delta_\emptyset}$) quite a different response time behaviour which is a result of the discrete sampling process to which *SimExp* adheres. More specifically, the continuous simulation of SimuLizar maintains performance-relevant properties (e.g. resource contention) over the entire simulation time range. *SimExp*, however, applies a simulation for every state such that the simulation context (and thus performance-relevant properties) is lost after each simulation. The extent to which the loss of information generally affects the evaluation of adaptation strategies cannot be fully clarified within this validation. However, it did not influence the validation use case currently under consideration but should be further investigated in future work.

9.3. Reliability Analysis of AI-enabled Systems

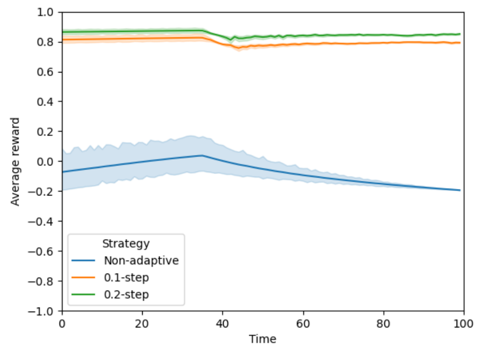
In this section, we present the results of validation goal 3. As already discussed in section 9.1.3.2, there are no approaches comparable to ours. Although there are model-based tools for analysing reliability attributes of software systems, none of these applies to the scenarios we are interested in, i.e. reliability analysis of AI-enabled systems taking into account the predictive uncertainty of AI components. That is to say, we cannot use any tool as a baseline or ground truth for comparison. Therefore, the main objective is to validate the plausibility of our reliability prediction approach for AI-enabled software systems. As our holistic approach is based on already validated components, we validate several plausibility assertions to show the validity. We initially focus on static software systems and expand the validation in the next section for self-adaptive systems. First, however, we must introduce the case study system of the Udacity self-driving car challenge. All validation artefacts (i.e. results and models) can be looked up under [156].



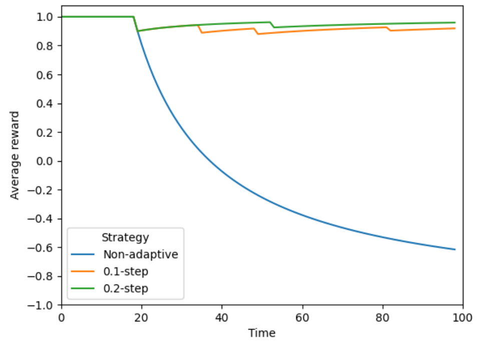
(a) Simulizar accumulated rewards for constant evolution



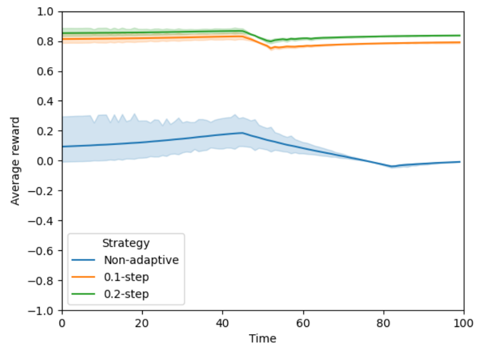
(b) SimExp results for constant evolution



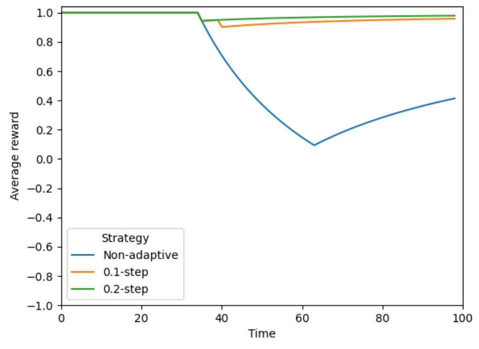
(c) Simulizar results for linear evolution



(d) SimExp results for linear evolution



(e) Simulizar results for peaked evolution



(f) SimExp results for peaked evolution

Figure 9.9.: Comparison of the accumulated rewards of SimuLizar and *SimExp*.

9.3.1. Udacity Self-Driving Car Challenge

We conduct the validation in the context of the Udacity self-driving car challenge [192]. The Udacity challenge is also known from the DNN (deep neural network) testing tool DeepTest [186] where it was used as a case study system to detect erroneous predictions from a selection of DNNs for steering angle prediction. In the subsequent two sections, we give a brief overview of the background of the challenge and introduce the considered DNNs afterwards.

9.3.1.1. Background

The Udacity self-driving car challenge is about using deep learning methods to take on certain control tasks. In this case, the DNNs are considered to predict the steering angle of a self-driving car based on pure image data. For this purpose, training data have been collected consisting of centred traffic images labelled with steering angles. A steering angle is considered as a value $\alpha \in [-25, 25]$; that is, the steering angle is a bounded value which is finally scaled by value $1/25$ such that the result is within the range $[-1, 1]$ [186].

Several teams were competing in the challenge to develop DNNs (trained by the provided training data) that are to predict the steering angle as well as possible for a final test dataset. The trained DNNs were evaluated by the RMSE (root-mean-squared-error; see section 2.7.1) metric. More specifically, the ranks of the DNNs were established by measuring the deviation of the predicted steering angles to the actual steering angles of the test data by applying RMSE. Based on the rank, the best DNNs were investigated. In the next section, we describe the DNNs that we considered during validation.

9.3.1.2. Considered Deep Neural Networks for Steering Angle Prediction

In the context of the Udacity challenge, ten DNNs have been submitted and ranked [192]. From these ten models, we selected two for validation, namely *Chauffeur* [42] and *Rambo* [146]. We decided to use Chauffeur and Rambo as their trained models are publicly available, well documented and easy to reproduce (i.e. in terms of evaluation w.r.t. the provided test data).

Chauffeur The Chauffeur model [42] for steering angle prediction achieved in the Udacity challenge the third rank [192] of all DNNs. It is based on two sub-models where the first extracts a set of features from the input image and the second sub-model is responsible for the actual prediction of the steering angle. More specifically, the first sub-model corresponds to a CNN (convolutional neural network) extracting 100 features from the input image. The extracted features are forwarded to the second sub-model that corresponds to an LSTM (long-short term memory) a subgroup of RNNs (recurrent neural networks). The LSTM does not directly predict the steering angle based on the given features but predicts the steering angle based on the concatenation of the features extracted from 100 images. In the following, we denote the Chauffeur prediction model as b_C.

Rambo The Rambo prediction model [146] achieved the second-best rank within the Udacity challenge [192]. It consists of three CNNs in which two CNNs are inspired by a DNN architecture of NVIDIA [29] and the third is inspired by comma.ai's steering model [51]. The Rambo model takes three consecutive images and computes two consecutive differenced images which serve as input. The result of each CNN is merged into a final layer. In the following, we denote the Chauffeur prediction model as b_R.

N-Version Recall from section 7.3 that we implemented the n-version programming pattern as an architectural template. For steering angle prediction, the n-version approach is essentially used to forward the input image to N redundant and distinct prediction models that generate a prediction for the image. The predictions are finally merged or combined by a voter or other type of decision procedure. In this case, the steering angles are combined by calculating the average of all predictions. Moreover, we consider two redundant steering angle prediction models, namely Chauffeur and Rambo. In the following, we denote the n-version prediction model as b_{NV}.

Perfect and Worst Steering Models Finally, we supplement the prediction models b_C, b_R and b_{NV} by two additional steering models (or black-boxes) b^+ and b^-. As introduced in section 9.1.1, the black-box models b^+ and b^- correspond to models where either the correct output is always produced for every input (i.e. b^+) or the wrong output is always produced for every input (i.e. b^-). In this case, the black-box models correspond to steering angle prediction models following the aforementioned semantics.

9.3.2. A Generic Software Architecture for Self-Driving Cars

So far, we introduced the Udacity self-driving car case study system and the various steering angle prediction models that we consider within the validation of goal 3. A steering angle prediction model only provides a small fraction of the functionality which constitutes a self-driving car. Thus, there are other functionalities (e.g. braking and accelerating) relating to other software components that make up the software system as a whole. The case study does not provide any information about the software architecture of the self-driving car because the focus is merely on the prediction of steering angles. However, we need a software architecture based on which we create the corresponding PCM model for which we want to show the validity of our reliability prediction approach. In the context of the Udacity challenge, there are no assumptions made about the software architecture in which a steering angle prediction model is included. Therefore, we assume w.l.o.g. a software architecture of a self-driving car w.r.t. proposed architectures from literature [18, 94, 183]. Note that we do not sacrifice generality by assuming a particular architecture as the plausibility assertions (9.1)-(9.3) are defined over a fixed architecture model C_b where only the used black-box model b is different. That is to say, if we analyse the architecture models individually with distinct AI components, only the AI components themselves have an impact on the reliability of the entire system. Therefore, the design of the architectural model is of no relevance as long as they remain the same for each AI component b. More generally, if plausibility assertions (9.1)-(9.3) hold for a particular software architecture of an architecture family, the assertions hold for the entire architecture family.

Since we are no experts in the field of engineering software architectures for self-driving cars, we searched the literature for existing architectural designs. Hereby, we created a PCM model for describing a fairly generic software architecture of a self-driving car based on [18, 94, 183]. The generic architecture model is by no means to be considered complete

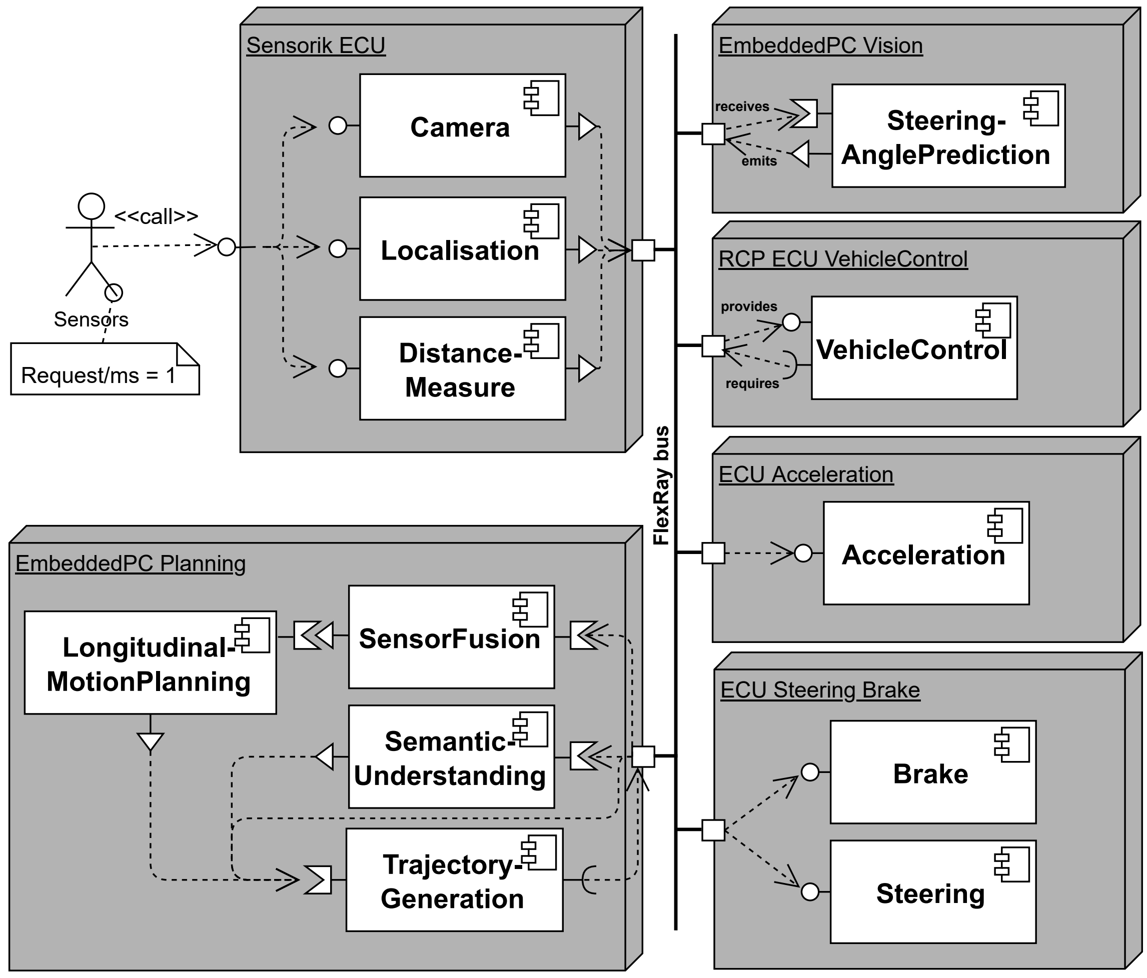


Figure 9.10.: An excerpt of the generic software architecture of a self-driving car assumed in validation based on [18, 94, 183].

but rather serves as a sufficiently abstract and simplified representation of a real software architecture for a self-driving car. In the following, we briefly discuss the architecture model and its generic components.

Figure 9.10 depicts the generic architecture. We assume an event-based communication between the individual software components, e.g. by using ROS (robot operating system) [142], which is already used in the context of autonomous driving [3]. Excluded from event-based communication are the `VehicleControl`, `Acceleration`, `Brake` and `Steering` components, as the `VehicleControl` is connected exclusively to the `TrajectoryGeneration` component, i.e. the generated trajectory is passed directly to the `VehicleControl` component and not received from other components. Similarly, the `VehicleControl` is directly connected with the components `Acceleration`, `Brake` and `Steering`. For clarification, we slightly abuse the graphical notations by not explicitly modelling every event channel from one component to another (e.g. as for the HRI example system in Figure 1.3). That is, not all event channels between components are directly visible in the figure but can be looked up in the PCM model directly (see [156]).

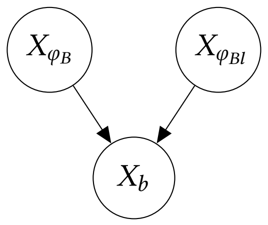


Figure 9.11.: The sensitivity model of the steering angle prediction models.

The system is constantly taking sensor values (i.e. by a fixed time rate) to perceive the environment. More specifically, the components `Camera`, `Localisation` and `DistanceMeasure` indicate that respective sensor values are taken. The sensor values are then emitted in the form of messages, which are received by the components for which there exists an event channel for the reception. For example, the `SteeringAnglePrediction` component receives the recorded images of the `Camera` component, predicts the steering angle and emits the prediction as a message again. Note that the `SteeringAnglePrediction` component represents one of the steering angle prediction models, e.g. Rambo b_R or Chauffeur b_C; depending on the currently deployed AI component. Moreover, the components `SensorFusion` and `SemanticUnderstanding` receive the recorded image data together with the sensed localisation and distance data. The `SensorFusion` abstracts away all procedures that take the sensor data from several sources (in this case image, localisation and distance data) to produce features or new data of higher quality [18]. Similarly, the `SemanticUnderstanding` component receives sensor data and is considered to generate semantically richer data. For example, the image data can be enriched by detecting objects or segmenting regions of interest that are relevant to the final trajectory planning [18]. The `LongitudinalMotionPlanning` component is responsible for determining the braking or acceleration parameters [94]. The `TrajectoryGeneration` component receives the according messages (i.e. the longitudinal motion parameters, semantically enriched data and predicted steering angle) to compute a collision-free trajectory. The trajectory is finally translated into control signals that are passed on to the `VehicleControl` component which controls the respective actuators (i.e. `Acceleration`, `Brake` and `Steering`).

Finally, note that we assumed *FlexRay* as communication protocol [94]. Moreover, we modelled the resource environment consisting of six resource containers and the deployment of the component as specified by Jo et al. [94].

9.3.3. Sensitivity Model and Analysis

In this section, we introduce the sensitivity model that we consider for validation. Furthermore, we present a sensitivity analysis algorithm that we developed based on the work of Tian et al. [186].

The general structure of the sensitivity model is depicted on Figure 9.11. For validation, we consider two uncertainties: image brightness X_{φ_B} and blurring $X_{\varphi_{Bl}}$ where $Val(X_{\varphi_B}) := \{Low, Normal, Strong\}$ indicates three brightness levels and $Val(X_{\varphi_{Bl}}) :=$

$\{Blurry, NotBlurry\}$ indicates two possible events. As usual, the random variable X_b is binary, i.e. $Val(X_b) := \{Success, Fail\}$. We decided to consider the uncertainties as they were used (among others) by DeepTest which already verified erroneous behaviour of the steering angle prediction models when observing said uncertainties in the context of the Udacity self-driving car challenge [186]. Clearly, image brightness and blurring are generally no good uncertainties or properties in the context of autonomous driving to analyse reliability attributes. Instead, one would rather focus on stronger properties, e.g. neuron coverage. Nonetheless, for showing the validity of our approach, the effectively chosen uncertainties (and their quality) are of no relevance since only their measurable effect on the AI model is of importance. For example, if we measure varying prediction confidence of an AI component for some uncertainties and observe the variation in the reliability predictions as well, we can conclude the validity (at least regarding assertion (9.1)) of our approach independent of the concretely chosen uncertainties.

Regarding the given structure of the sensitivity model, we need to determine the sensitivity distribution (i.e. $P(X_b \mid X_{\varphi_B}, X_{\varphi_{Bl}})$) which we derive by applying a sensitivity analysis. For the sensitivity analysis, we consider an algorithm depicted on algorithm 9.1. Clearly, we could have used one of the sensitivity analysis approaches described in section 7.1.2; however, we decided to develop our own analysis as we would have to adopt the sensitivity analysis approaches anyway to obtain a sensitivity model with the required structure. Moreover, we can reuse insights from the DeepTest tool to derive the sensitivity values (as we will see later).

The algorithm inputs five parameters.

- D_{Sens}: The parameter refers to the sensitivity dataset D_{Sens} based on which the sensitivity model is determined. Hereby, D_{Sens} does not simply consist of labelled data, but each data is additionally labelled with the uncertainty values which are observed in the given data. More specifically, an element $d := (x, y, \varphi_B, \varphi_{Bl}) \in D_{Sens}$ of the sensitivity dataset includes (in addition to a given image $x \in \mathcal{X}$ and correct label or output $y \in \mathcal{Y}$ of the input and output space of black-box b) two uncertainty values or labels φ_B and φ_{Bl}. For example, for a given image x and the corresponding label y, there are two additional uncertainty labels, e.g. $\varphi_B = Low$ and $\varphi_{Bl} = Blurry$, which means that the image x is blurred and also has low brightness. The sensitivity dataset D_{Sens} consists, in effect, of synthetic image data and originates from a test dataset D_{Test} used to evaluate the accuracy of steering angle prediction model b. D_{Sens} is generated by D_{Test} by applying image transformations that account for the distinct uncertainties under consideration. In the next section, we discuss in more detail how dataset D_{Sens} is generated.
- b: Corresponds to the steering angle prediction model b which is to be analysed.
- $RMSE_{orig}(b)$: The parameter refers to the original RMSE value associated with steering angle prediction model b and is originally determined by D_{Test}.
- $RMSE_{avg}$: Corresponds to the average RMSE of all steering angle prediction models, i.e. $RMSE_{avg} := \frac{1}{N} \sum_{b \in \{b_1, \dots, b_N\}} RMSE_{orig}(b)$.

Algorithm 9.1: Sensitivity analysis algorithm

```
Input: Dataset $D_{Sens}$,
Steering angle prediction model $b$,
The original RMSE $RMSE_{orig}(b)$,
The averaged RMSE $RMSE_{avg}$,
The set of uncertainties $\Phi$
Output: Sensitivity distribution $P_{\theta_b}(X_b \mid X_{\varphi_B}, X_{\varphi_{Bl}})$
1  $\lambda \in I\!N$ // The $RMSE_{avg}$ scaling factor
2  $S(\varphi_B, \varphi_{Bl}) \longleftarrow initialiseS()$ // an empty list, for all $(\varphi_B, \varphi_{Bl}) \in \Phi$
3  $\theta_b(\varphi_B, \varphi_{Bl}) \longleftarrow initialiseParams()$ // $\forall(\varphi_B, \varphi_{Bl}) \in \Phi : \theta_b(\varphi_B, \varphi_{Bl}) = 0$
4  foreach $(x, y, \varphi_B, \varphi_{Bl}) \in D_{Sens}$ do
5      $p \longleftarrow (y, b(x))$
6      $append(S(\varphi_B, \varphi_{Bl}), p)$ // Appends $p$ to $S(\varphi_B, \varphi_{Bl})$
7  end
8  foreach $(\varphi_B, \varphi_{Bl}) \in \Phi$ do
9      $RMSE_{local}(b) \longleftarrow \frac{\lambda \cdot RMSE_{avg}}{2}$
10     if $S(\varphi_B, \varphi_{Bl}) \neq \emptyset$ then
11         $\boldsymbol{y}, \hat{\boldsymbol{y}} \longleftarrow S(\varphi_B, \varphi_{Bl})$ // fetch all predictions and respective label
12         $RMSE_{local}(b) \longleftarrow \sqrt{\frac{1}{|S(\varphi_B,\varphi_{Bl})|} \sum_{i=1}^{|S(\varphi_B,\varphi_{Bl})|} (\boldsymbol{y}_i - \hat{\boldsymbol{y}}_i)^2}$ // calculates the local
            RMSE
13     end
14     if $\varphi_B = Normal \wedge \varphi_{Bl} = NotBlurry$ then
15         $RMSE_{local}(b) \longleftarrow RMSE_{orig}(b)$
16     end
17     $\theta_b(\varphi_B, \varphi_{Bl}) \longleftarrow \mathbb{1}_{RMSE_{local}(b) \leq \lambda \cdot RMSE_{avg}} \cdot \left(1 - \frac{RMSE_{local}(b)}{\lambda \cdot RMSE_{avg}}\right)$
18 end
19 return $P_{\theta_b}(X_b \mid X_{\varphi_B}, X_{\varphi_{Bl}})$
```

- Φ: The set of uncertainties based on which the sensitivity model is analysed; in our case $\Phi := Val(X_{\varphi_B}) \times Val(X_{\varphi_{Bl}})$.

The output of the algorithm corresponds to the estimated sensitivity model, i.e. $P_{\theta_b}(X_b \mid X_{\varphi_B}, X_{\varphi_{Bl}})$.

Basically, the algorithm holds two variables (i.e. $S(\varphi_B, \varphi_{Bl})$ and $\theta_b(\varphi_B, \varphi_{Bl})$) and one constant λ. The constant λ corresponds to a natural number which describes a scaling factor used to determine the success probability (we explain the meaning of λ later).

The algorithm initially iterates over all elements of D_{Sens}. For each image x the corresponding prediction is made by applying b on x, i.e. $b(x)$. The correct output y and prediction $b(x)$ are appended to $S(\varphi_B, \varphi_{Bl})$. Recall that any $d \in D_{Sens}$ is labelled with the corresponding uncertainties which are observable in image x. That is, the prediction of b is made w.r.t. to the labeled uncertainties φ_B and φ_{Bl}. Thus, the prediction $b(x)$ and correct output or label y must be associated with φ_B and φ_{Bl} which is done by appending the tuple $(y, b(x))$ to $S(\varphi_B, \varphi_{Bl})$.

The second loop of the algorithm iterates over all uncertainties of Φ (i.e. $Val(X_{\varphi_B}) \times Val(X_{\varphi_{Bl}})$). Within the loop, variable $RMSE_{local}(b)$ is declared and initialized. In principle, the variable holds the RMSE which is calculated based on $S(\varphi_B, \varphi_{Bl})$ for a given uncertainty pair φ_B,φ_{Bl}. In other words, it holds a value which is calculated by applying RMSE to all prediction and label pairs that are associated with particular uncertainty pair φ_B,φ_{Bl}. We consider $RMSE_{local}(b)$ as local RMSE because it is not computed based on the entire dataset D_{Sens} (as opposed to $RMSE_{orig}(b)$ which is calculated w.r.t. D_{Test}) but only a subset of D_{Sens}. Initially, $RMSE_{local}(b)$ is set to $(\lambda \cdot RMSE_{avg})/2$ (the reason for the initialisation is explained shortly). Afterwards, the first if-clause checks whether $S(\varphi_B, \varphi_{Bl})$ maintains values for uncertainty pair φ_B,φ_{Bl}. If so, the local RMSE is calculated w.r.t. the values of uncertainty pair φ_B,φ_{Bl} stored in $S(\varphi_B, \varphi_{Bl})$. The second if-clause checks whether the uncertainty pair corresponds to the default values. Recall that dataset D_{Sens} is generated by D_{Test}, i.e. for all elements of D_{Test} some image transformation is applied to generate synthetic images that include increased brightness or blurring. The data elements of D_{Test}, however, are considered to indicate no uncertainties; that is, no increased image brightness and no blurring are observed in the images. Thus, we cannot calculate the local RMSE because $S(Normal, NotBlurry) = \emptyset$. Instead, we set $RMSE_{local}(b) := RMSE_{orig}(b)$ since $RMSE_{orig}(b)$ is computed w.r.t. D_{Test}, i.e. based on images where $\varphi_B = Normal \wedge \varphi_{Bl} = NotBlurry$ holds true.

Next, the success probability of making a correct prediction w.r.t. φ_B,φ_{Bl} must be determined. Since X_b is a binary random variable, it is sufficient to estimate either the event of encountering a successful or an erroneous prediction. We estimate the success probability $\theta_b(\varphi_B, \varphi_{Bl}) = Pr(X_b = Success \mid X_{\varphi_B} = \varphi_B, X_{\varphi_{Bl}} = \varphi_{Bl})$ by taking $RMSE_{local}(b)$ into account:

$$\theta_b(\varphi_B, \varphi_{Bl}) = \mathbb{1}_{RMSE_{local}(b) \leq \lambda \cdot RMSE_{avg}} \cdot \left(1 - \frac{RMSE_{local}(b)}{\lambda \cdot RMSE_{avg}}\right) \tag{9.9}$$

The first term of the product of (9.9) makes sure that the local RMSE is smaller or equal to the average RMSE of all considered steering angle prediction models $RMSE_{avg}$ scaled by constant λ. In other words, if for a given uncertainty pair the RMSE is strongly deviating (i.e. the uncertainties have a great impact on the prediction results), the $RMSE_{local}(b)$ increases accordingly but must not exceed the threshold $\lambda \cdot RMSE_{avg}$. Otherwise, $\mathbb{1}_{RMSE_{local}(b) \leq \lambda \cdot RMSE_{avg}}$ evaluates to 0 and thus the success probability becomes 0 as well. However, if $RMSE_{local}(b) \leq \lambda \cdot RMSE_{avg}$ holds, the success probability is calculated according to the second term of the product of (9.9), i.e. $1 - (RMSE_{local}(b)/\lambda \cdot RMSE_{avg}) \in [0, 1]$. This procedure is inspired by a metamorphic relation introduced in DeepTest [186] which measures the deviation of predicted and actual steering angles.

If no condition of the if-clauses of algorithm 9.1 evaluates to true, $RMSE_{local}(b) = (\lambda \cdot RMSE_{avg})/2$ applies. If we consider the if-clauses more precisely, it can be seen that there still might be the case $S(\varphi_B, \varphi_{Bl}) = \emptyset$. Another reason (besides $\varphi_B = Normal \wedge \varphi_{Bl} = NotBlurry$) why $S(\varphi_B, \varphi_{Bl}) = \emptyset$ might still apply relates simply to the fact that D_{Sens} contains no images where a respective uncertainty tuple $(\varphi_B, \varphi_{Bl})$ is observed. In this case, no statement about the success probability can be made, i.e. the probability of observing success or failure is by chance. More specifically, if we substitute $RMSE_{local}(b) = (\lambda \cdot RMSE_{avg})/2$ in (9.9), it evaluates to 0.5.

Finally, based on $\theta_b(\varphi_B, \varphi_{Bl})$ the sensitivity distribution $P_{\theta_b}(X_b \mid X_{\varphi_B}, X_{\varphi_{Bl}})$ is determined and returned in which θ_b indicates that the parametric setting of the distribution refers to $\theta_b(\varphi_B, \varphi_{Bl})$.

9.3.4. Generating Synthetic Data

In the last section, we discussed the sensitivity model and analysis. For the sensitivity analysis, we need a sensitivity dataset D_{Sens} which does not only include input image and steering angle label pairs but also further labels regarding the uncertainties observable in the input image. For this purpose, we generate D_{Sens} by the test dataset D_{Test} which is primarily used to evaluate the accuracy (i.e. RMSE) of a steering angle prediction model b. More specifically, we apply image transformations on each input image of D_{Test}. For the generation of synthetic image data, we reuse image transformations of DeepTest [186] that generate synthetic image data to detect erroneous behaviour in AI models. The synthetic image data generation algorithm is depicted on algorithm 9.2 The algorithm

Algorithm 9.2: Synthetic image data generation algorithm

```
   Input: Test dataset $D_{Test}$
   Output: Sensitivity dataset $D_{Sens}$
1  $D_{Sens} \leftarrow \emptyset$
2  foreach $(x, y) \in D_{Test}$ do
3      $\varphi_B \leftarrow Normal$
4      $\varphi_{Bl} \leftarrow NotBlurry$
5      $r \leftarrow random(1, 3)$ // selects randomly a natural number from the range
6      if $r = 1$ then
7          $x, \varphi_{Bl} \leftarrow t_{Bl}(x)$
8      end
9      if $r = 2$ then
10         $x, \varphi_B \leftarrow t_B(x)$
11     end
12     if $r = 3$ then
13         $x, \varphi_{Bl} \leftarrow t_{Bl}(x)$
14         $x, \varphi_B \leftarrow t_B(x)$
15     end
16     $D_{Sens} \leftarrow D_{Sens} \cup (x, y, \varphi_B, \varphi_{Bl})$
17 end
18 return $D_{Sens}$
```

starts to loop over all input images and steering angle label pairs. For each iteration,

Name	Value	Description
λ	20	Scaling factor of $RMSE_{avg}$.
D_{Test}	-	The CH2_001 test dataset [191] which encompasses 5614 images and corresponding steering angle labels.
$acc(b)$	see (9.10)	We use an RMSE-based accuracy measure which is calculated based on D_{Test}.

Table 9.12.: Overview of the parameter setting for the Udacity self-driving car case study system.

the brightness and blurring uncertainty values are initially set to their default values, i.e. *Normal* and *NotBlurry*. Afterwards, a natural number r is randomly selected from the set $\{1, 2, 3\}$ to distinguish between three cases. In the first case (i.e. $r = 1$), only the image transformation to insert the blur t_{Bl} is applied to x, returning a synthetic image (including the blur) and the corresponding uncertainty label (i.e. *Blurry*). The image transformation t_{Bl} is reused by DeepTest, where several different blur filters are available: Averaging, Gaussian, median and bilateral. In the transformation t_{Bl} we randomly select one of the blur filters. In the second case (i.e. $r = 2$), only the image transformation t_B to adjust the brightness of the image is applied. Again, we reuse the image transformation of DeepTest where a constant value $\beta \in \{10, 20, 30, \ldots, 100\}$ is either added (to increase brightness) or subtracted (to decrease brightness) from each pixel in the image. Moreover, whether β is added or subtracted is again randomly chosen. t_B returns the synthetic image with the adjusted brightness and uncertainty label (i.e. *Strong* or *Low*). In the third case (i.e. $r = 3$), both transformations (i.e. t_{Bl} and t_B) are applied. Depending on the randomly selected cases, a new synthetic image is created which is finally added to the set D_{Sens}.

9.3.5. Experiment Setup

In this section, we briefly discuss the experimental setup including the preliminary analyses and experimental parameters. A summary can be found in Table 9.12.

Recall from section 9.3.1.2 that we consider for validation five steering angle prediction models, namely Chauffeur b_C, Rambo b_R, n-version b_{NV} (including b_C and b_R as two versions), a perfect steering angle prediction model b^+ and the worst possible model b^- for predicting steering angles. For each black-box b, we determined the accuracy $acc(b)$. Recall that the accuracy measure is relevant for the plausibility assertions. Concretely, we use an RMSE-based accuracy measure:

$$acc(b) = 1 - RMSE_{orig}(b) = 1 - \sqrt{\frac{1}{N}\sum_{i=1}^{N}(b(x_i) - y_i)^2}, N = |D_{Test}| \tag{9.10}$$

We calculate the RMSE (recall from section 2.17) based on the test dataset D_{Test}. For D_{Test}, we considered the test dataset CH2_001 [191] which encompasses 5614 images. The dataset was used in the Udacity self-driving car challenge as the final test dataset to evaluate the distinct steering angle prediction models.

For each steering angle prediction model (except b^- and b^+), we analysed the sensitivity as described in section 9.3.3 based on a generated sensitivity dataset D_{Sens} (as described in 9.3.4). The individual sensitivity models are depicted on Table 9.13. For each sensitivity model generated by the sensitivity analysis, we created a corresponding *EnvDyn* model which holds the sensitivity values and is used during reliability prediction. Note that for reliability prediction with our approach, the probabilities of the uncertainties themselves must also be determined, i.e. $P(X_{\varphi_B})$ and $P(X_{\varphi_{Bl}})$. Therefore, we have assumed (w.l.o.g.) a distribution defined over the uncertainties that we have consistently used for each reliability prediction.

Moreover, we created a PCM model describing the software architecture of a self-driving car. Recall that w.l.o.g. we assumed a generic software architecture for self-driving cars based on proposed architectures from literature (see 9.3.2).

For each steering angle prediction model, we created an uncertainty-refined failure model to connect each b with its respective sensitivity model. Furthermore, each failure model references the same software-induced failure type (which is defined within the `Steering-AnglePrediction` component) reflecting the refined failure type modelling the failure potential of each AI component. Based on the uncertainty-refined failure model, we predicted the reliability (or rather probability of success) of our generic software architecture of a self-driving car by considering each black-box b individually.

As a final remark, steering angle prediction models b^f which are annotated by f refer to models where the filtering pattern from section 7.1.1.2 is applied (or the corresponding architectural template) and $M^f_{C_b}$ the architecture model, respectively. If the filtering pattern is applied, the resulting architecture model M_C is modified as well. We stated, however, that we consider only a single architectural configuration; more specifically, the generic software architecture from section 9.3.2 for which we created the corresponding PCM model M_C. As a matter of fact, the same applies for $M_{C_{b_{NV}}}$ which describes the architecture model where the n-version pattern (or architectural template) is applied. This contradicts our plausibility assertions in which we stated that we consider a single architecture model M_{C_b} where only the AI component b is interchangeable. The reason for considering a fixed architectural model M_{C_b}, where only b is variable, is that only b needs to have an impact on the reliability of the system, so we can relate reliability attributes of b directly to reliability predictions and for comparison (e.g. $M_{C_{b_R}}$ with $M_{C_{b_C}}$). That is, when applying the filter to architecture model M_{C_b}, we obtain a new architecture model $M^f_{C_b}$ which are not directly comparable, e.g. the filter component itself might indicate a certain failure probability which impacts to some extent the overall reliability of the system. For validation purposes, however, we configured all reliability-related elements inserted into the architectural model (after either the filtering or the n-version pattern was applied) to not affect the reliability of the system, but only the prediction accuracy or predictive uncertainty of b itself, i.e. the sensitivity model. This means that each applied pattern or architectural template acts exclusively as an architectural countermeasure (as described on 7.1.3.2) and has no other consequences in terms of reliability. Therefore, the architecture models $M_{C_{b_{NV}}}$, $M^f_{C_{b_R}}$ and $M^f_{C_{b_C}}$ are admittedly different in terms of their architectural configuration but are still directly comparable with all other models.

9.3.6. Experiment Results

In this section, we present the experiment results. Recall that the main intention is to show that the plausibility assertions (9.1)-(9.3) are satisfied. Moreover, we show similarity between the sensitivity model $P(X_b \mid X_{\varphi_B}, X_{\varphi_{Bl}})$ and the reliability predictions $P(X_{Sys} \mid X_U, X_{\varphi_B}, X_{\varphi_{Bl}})$. In other words, we show that the predicted success/failure probabilities indicate the same behaviour or characteristics as the sensitivity model for different uncertainty values φ_B and φ_{Bl}.

First, however, consider Table 9.13 which depicts the sensitivity models of all steering angle prediction models. As mentioned in the last section, we obtained the sensitivity models and their respective probabilities by applying the sensitivity analysis from section 9.3.3. For b^+, we defined the success probability to be always 1 to account for the perfect prediction property. Accordingly, we define the success probabilities of b^- to be always 0. From the sensitivity model, it appears that Chauffeur and Rambo indicate lower probabilities of success (and thus higher probabilities of failure) for uncertainty values that deviate from what we consider normal (i.e. images that are not blurred $NotBlurry$ and under normal brightness conditions $Normal$). Moreover, we calculated the RMSE of each model based on the original test dataset D_{Test} (i.e. $RMSE_{orig}(b)$) and based on dataset D_{Sens} (i.e. $RMSE_{Sens}(b)$) which contains the generated synthetic image data enriched by varying image brightness and blur. For Chauffeur b_C, we calculated $RMSE_{orig}(b_C) = 0.05768$, which deviates slightly (but negligibly) from the originally documented RMSE (0.05816) of the Udacity challenge. Similarly, Rambo's calculated RMSE ($RMSE_{orig}(b_R) = 0.05682$) also differs negligibly from the published results where the RMSE is reported as 0.05787. However, the rank of the two models remains the same, i.e. Rambo performs better than Chauffeur. The final rank of all models can be found in [192]. The third and fourth columns depict the success and failure probabilities of each b. Note that $RMSE_{orig}(b^-) = \infty$ accounts for the fact that b^- produces the worst possible predictions. Moreover, by considering the accuracy $acc(b)$ of each steering angle prediction model b (recall that $acc(b) = 1 - RMSE_{orig}(b)$), we observe the following order:

$$b^- < b_C < b_R < b_{NV} < b^+ \tag{9.11}$$

By recalling assertion (9.1), we must observe

$$rel(M_{C_{b^-}}) < rel(M_{C_{b_C}}) < rel(M_{C_{b_R}}) < rel(M_{C_{b_{NV}}}) < rel(M_{C_{b^+}}) \tag{9.12}$$

in the reliability predictions of our approach. Therefore, consider Table 9.14.

The table shows all steering angle prediction models which are ordered from top to bottom according to (9.11). The third column of the table shows the predicted success probabilities

b	$RMSE_{orig}$	$RMSE_{Sens}$	Φ	**Succ.**	**Fail.**
b^-	∞	∞	$(Blurry, Low)$	0	1
			$(Blurry, Normal)$	0	1
			$(Blurry, Strong)$	0	1
			$(NotBlurry, Low)$	0	1
			$(NotBlurry, Normal)$	0	1
			$(NotBlurry, Strong)$	0	1
b_C	0.05768	0.07065	$(Blurry, Low)$	0.9374	0.0626
			$(Blurry, Normal)$	0.9347	0.0653
			$(Blurry, Strong)$	0.9248	0.0752
			$(NotBlurry, Low)$	0.933	0.067
			$(NotBlurry, Normal)$	0.9448	0.0552
			$(NotBlurry, Strong)$	0.9305	0.0695
b_R	0.05682	0.06144	$(Blurry, Low)$	0.9462	0.0538
			$(Blurry, Normal)$	0.9365	0.0635
			$(Blurry, Strong)$	0.9435	0.0565
			$(NotBlurry, Low)$	0.9411	0.0589
			$(NotBlurry, Normal)$	0.9456	0.0544
			$(NotBlurry, Strong)$	0.9434	0.0566
b_{NV}	0.04221	0.04631	$(Blurry, Low)$	0.9561	0.0439
			$(Blurry, Normal)$	0.9542	0.0458
			$(Blurry, Strong)$	0.9523	0.0477
			$(NotBlurry, Low)$	0.9562	0.0438
			$(NotBlurry, Normal)$	0.9596	0.0404
			$(NotBlurry, Strong)$	0.9543	0.0457
b^+	0	0	$(Blurry, Low)$	1	0
			$(Blurry, Normal)$	1	0
			$(Blurry, Strong)$	1	0
			$(NotBlurry, Low)$	1	0
			$(NotBlurry, Normal)$	1	0
			$(NotBlurry, Strong)$	1	0

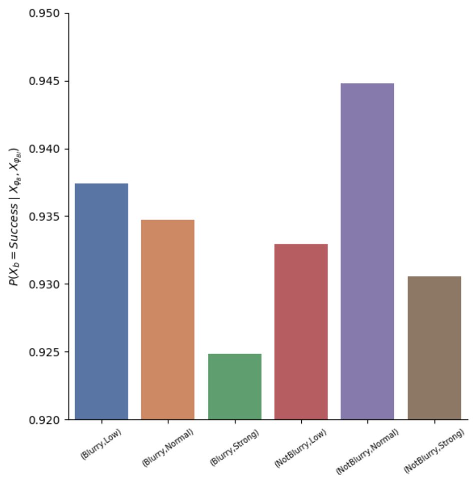
Table 9.13.: An overview of the steering angle prediction models, their sensitivity models and RMSE values.

that we obtained after applying our reliability approach for each b to architecture model M_{C_b}. It can be seen that the same order applies as required in (9.12). Thereby, we conclude that our reliability prediction approach maintains plausibility assertion (9.1).

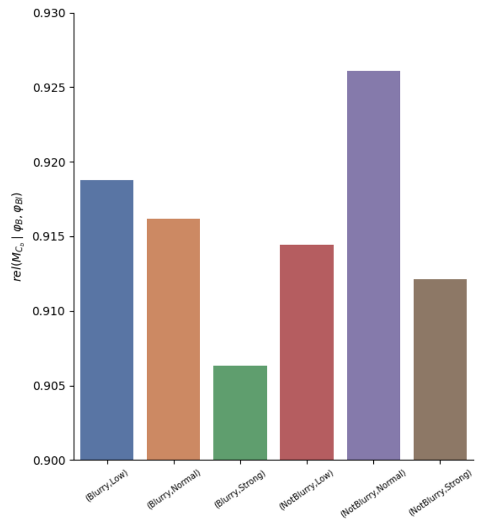
After we validated that our reliability prediction approach preserves the overall success probability of the whole system (i.e. $rel(M_{C_b})$) w.r.t. the accuracy of b, we now investigate how the characteristics of b regarding $(\varphi_B, \varphi_{Bl})$ are preserved. Therefore, consider Figure 9.12, Figure 9.13, Figure 9.14 and Figure 9.15 which relate the individual success

Model	$acc(b)$	$rel(M_{C_b})$	$Pr(X_b = Success)$
b^-	$-\infty$	0	0
b_C	0.94232	0.9170	0.9356
b_R	0.94318	0.9242	0.9429
b_{NV}	0.95779	0.9372	0.9561
b^+	1	0.9798	1

Table 9.14.: Comparing the accuracy of all steering angle prediction models with the overall success probability of our reliability prediction approach and the overall success probability of individual sensitivity models (i.e. $Pr(X_b = Success)$).



(a) Success probabilities of b_C.

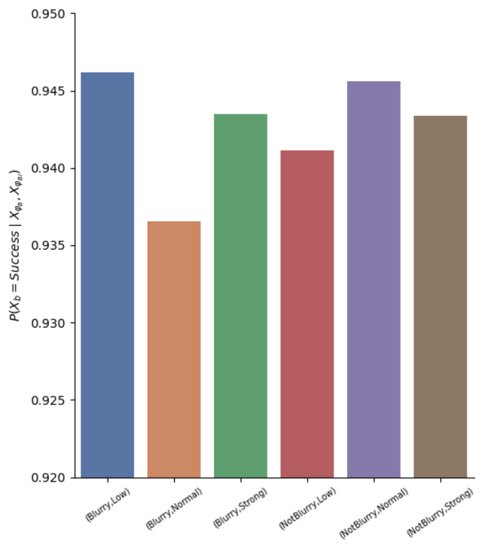


(b) Prediction results $rel(M_{C_{b_C}} \mid \varphi_B, \varphi_{Bl})$.

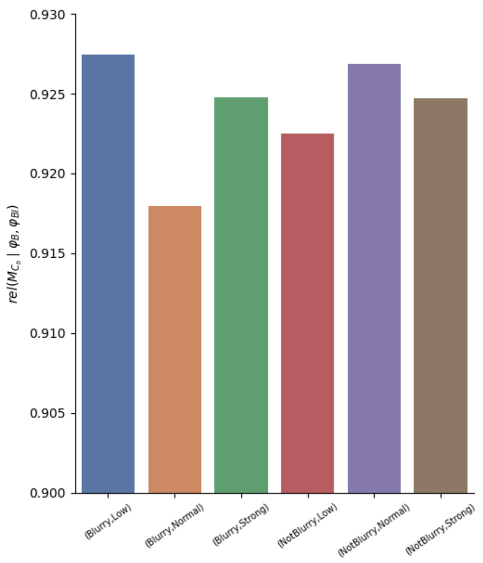
Figure 9.12.: Comparing the sensitivity model of Chauffeur (i.e. $P(X_{b_C} = Success \mid X_{\varphi_B}, X_{\varphi_{Bl}})$) with the prediction results of our reliability prediction approach (i.e. $rel(M_{C_{b_C}} \mid \varphi_B, \varphi_{Bl})$).

probabilities of the sensitivity models of b^+, b_R, b_C and b_{NV} with the predicted success probabilities $rel(M_{C_b} \mid \varphi_B, \varphi_{Bl})$ for all φ_B, φ_{Bl} and fixed architecture model M_{C_b}.

Note that b^- is not depicted because the success probabilities of the sensitivity model and the predicted success probabilities of the system are 0. On the other hand, Figure 9.15 depicts the success probabilities of the sensitivity model of perfect model b^+ (and the predicted success probabilities of the system including b^+). It can be seen that both $P(X_{b^+} = Success \mid X_{\varphi_B}, X_{\varphi_{Bl}})$ and $rel(M_{C_{b^+}} \mid \varphi_B, \varphi_{Bl})$ indicate equal probabilities which is a result of the fact that the success probability of b^+ is for all pairs $(\varphi_B, \varphi_{Bl})$ equal to 1. It is important to note that the figures do not depict probability distributions but merely the individual success probabilities for any uncertainty pair, e.g. $Pr(X_{b_C} = Success \mid X_{\varphi_B} = Low, X_{\varphi_{Bl}} = Blurry)$ and $Pr(X_{Sys} = Success \mid X_U = U, X_{\varphi_B} = Low, X_{\varphi_{Bl}} = Blurry)$ for b_C. However, each bar of a single plot is associated with a probability distribution defined over X_b and X_{Sys}, respectively. For example, $Pr(X_{b_C} = Success \mid X_{\varphi_B} = Low, X_{\varphi_{Bl}} = Blurry) = 0.9374$ and thus $Pr(X_{b_C} = Fail \mid X_{\varphi_B} = Low, X_{\varphi_{Bl}} = Blurry) = 1 - 0.9374$; similarly, $Pr(X_{Sys} = Success \mid X_U = U, X_{\varphi_B} = Low, X_{\varphi_{Bl}} = Blurry) = rel(M_{C_{b_C}} \mid Low, Blurry) =$

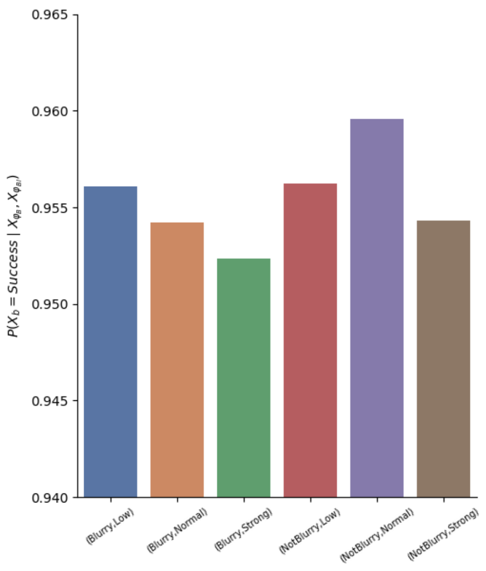


(a) Success probabilities of b_R.

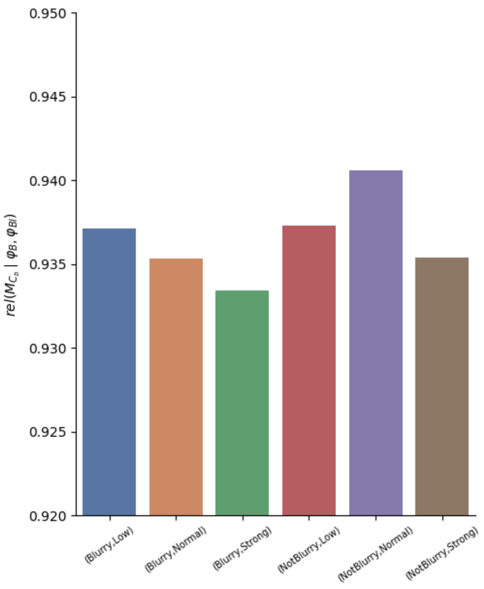


(b) Prediction results $rel(M_{C_{b_R}} \mid \varphi_B, \varphi_{BI})$.

Figure 9.13.: Comparing the sensitivity model of Rambo (i.e. $P(X_{b_R} = Success \mid X_{\varphi_B}, X_{\varphi_{BI}})$) with the prediction results of our reliability prediction approach (i.e. $rel(M_{C_{b_R}} \mid \varphi_B, \varphi_{BI})$).



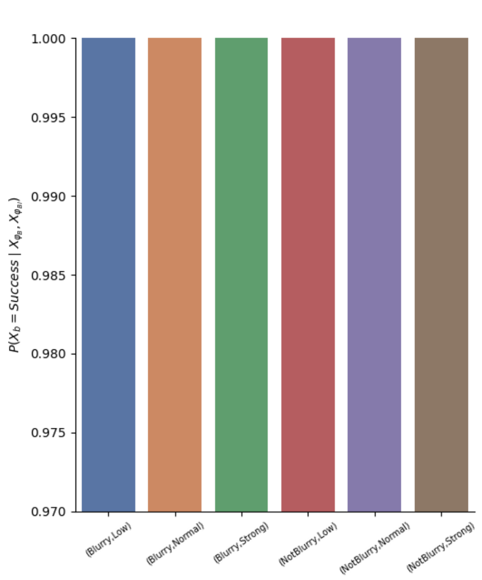
(a) Success probabilities of b_{NV}.



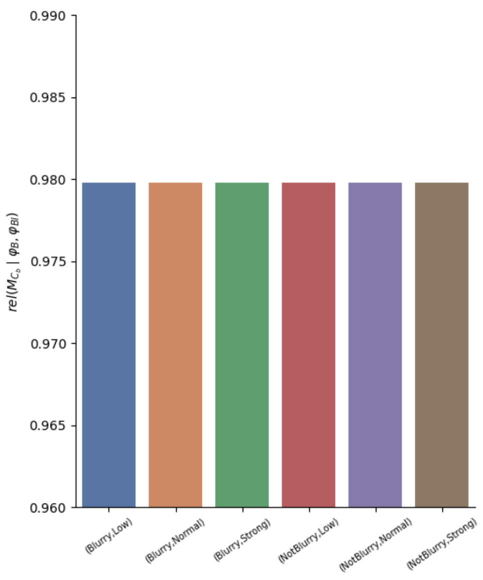
(b) Prediction results $rel(M_{C_{b_{NV}}} \mid \varphi_B, \varphi_{BI})$.

Figure 9.14.: Comparing the sensitivity model of the n-version model (i.e. $P(X_{b_{NV}} = Success \mid X_{\varphi_B}, X_{\varphi_{BI}})$) with the prediction results of our reliability prediction approach (i.e. $rel(M_{C_{b_{NV}}} \mid \varphi_B, \varphi_{BI})$).

0.9188 and $Pr(X_{Sys} = Fail \mid X_U = U, X_{\varphi_B} = Low, X_{\varphi_{BI}} = Blurry) = 1 - 0.9188$. From the figures, it is quite obvious that the probabilities of success show the same behaviour for each uncertainty pair or more specifically are proportional to $rel(M_{C_b} \mid \varphi_B, \varphi_{BI})$. Thus, the same must be true for the probabilities of failure. Complementary to the figures,



(a) Success probabilities of b^+.



(b) Prediction results $rel(M_{C_{b^+}} \mid \varphi_B, \varphi_{Bl})$.

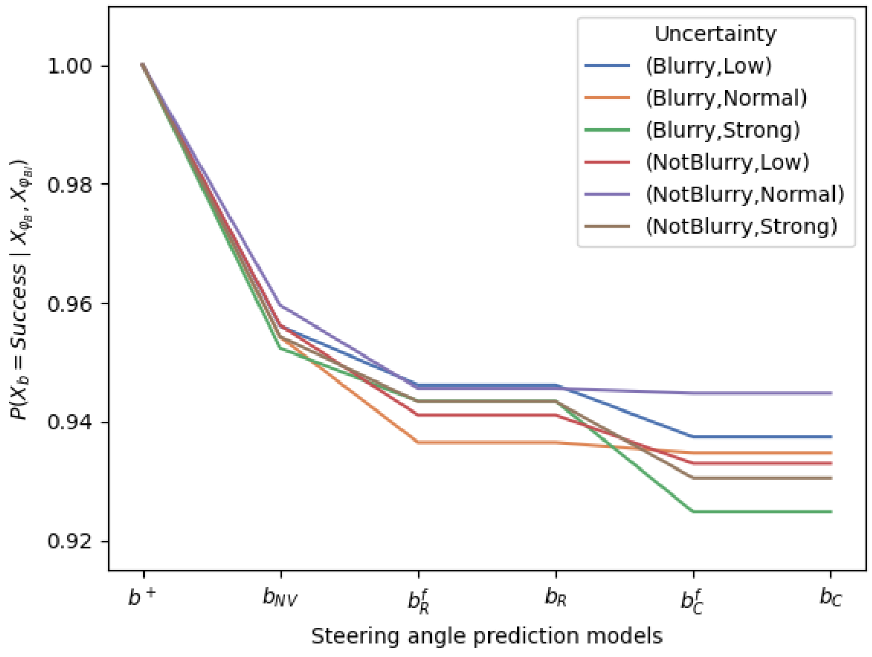
Figure 9.15.: Comparing the sensitivity model of the perfect steering angle prediction model (i.e. $P(X_{b^+} = Success \mid X_{\varphi_B}, X_{\varphi_{Bl}})$) with the prediction results of our reliability prediction approach (i.e. $rel(M_{C_{b^+}} \mid \varphi_B, \varphi_{Bl})$).

Table 9.15 contains the concrete values of the success probabilities, the probabilities of the uncertainties pairs themselves, and the Bhattacharyya distance D_B.

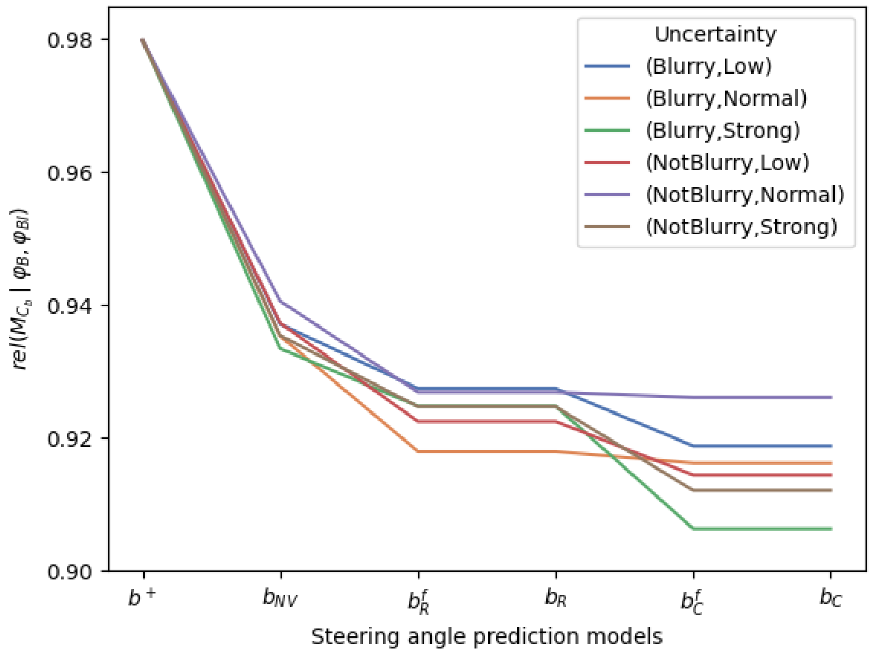
Note that the table does not consider b^- since the success probabilities are all zero. Moreover, for simplification, we do not depict all entries of b^+ because the success probabilities (i.e. $P(X_b \mid X_\Phi)$ and $rel(M_{C_b} \mid \Phi)$) and Bhattacharyya distances are the same for all uncertainties. We calculated the Bhattacharyya distance between $P(X_b \mid X_{\varphi_B} = \varphi_B, X_{\varphi_{Bl}} = \varphi_{Bl})$ and $P(X_{Sys} \mid X_U = U, X_{\varphi_B} = \varphi_B, X_{\varphi_{Bl}} = \varphi_{Bl})$ for all $(\varphi_B, \varphi_{Bl})$. As can be seen in Table 9.15, the Bhattacharyya distance is fairly small for all b which shows the similarity of the respective distributions.

Finally, we validate plausibility assertions (9.2) and (9.3). For the assertions, we have to compare the sets Φ_{b_i,b_j} pairwise for all considered black-boxes b. However, the pairwise comparison is tedious and difficult to assess even for a few steering angle prediction models as we have to compare all combinations (b_i, b_j). Instead, we exploit a property that can be observed in all steering angle prediction models. If we carefully review the table Table 9.15, we can see that the individual success probabilities (i.e. the fourth column) are partially ordered between the steering angle prediction models (ordered from top to bottom according to (9.11)), e.g. $Pr(X_{b^-} = Success \mid X_{\varphi_B} = Low, X_{\varphi_{Bl}} = Blurry) < Pr(X_{b_C} = Success \mid X_{\varphi_B} = Low, X_{\varphi_{Bl}} = Blurry) = Pr(X_{b_C^f} = Success \mid X_{\varphi_B} = Low, X_{\varphi_{Bl}} = Blurry) < \ldots$ Figure 9.16 depicts this monotonic property.

More precisely, Figure 9.16a shows monotonically decreasing success probabilities (y-axis) of each sensitivity model (x-axis), with the steering angle prediction models sorted according to (9.11) but in descending order. Regarding plausibility assertion (9.2) and (9.3),



(a) Success probabilities of the sensitivity models $P(X_b = Success \mid X_{\varphi_B}, X_{\varphi_{BI}})$.



(b) Prediction results $rel(M_{C_b} \mid \varphi_B, \varphi_{BI})$.

Figure 9.16.: Comparing the success probabilities of the individual sensitivity models with the predicted success probability of our approach.

b	Φ	$P(X_\Phi)$	$P(X_b \mid X_\Phi)$	$rel(M_{C_b} \mid \Phi)$	D_B
b_C	$(Blurry, Low)$	0.09	0.9374	0.9188	0.0006
	$(Blurry, Normal)$	0.12	0.9347	0.9162	0.0006
	$(Blurry, Strong)$	0.09	0.9248	0.9063	0.0005
	$(NotBlurry, Low)$	0.21	0.933	0.9144	0.0006
	$(NotBlurry, Normal)$	0.28	0.9448	0.9260	0.0007
	$(NotBlurry, Strong)$	0.21	0.9305	0.9121	0.0005
b_C^f	$(Blurry, Low)$	0	0.9374	0.9188	0.0006
	$(Blurry, Normal)$	0	0.9347	0.9162	0.0006
	$(Blurry, Strong)$	0	0.9248	0.9063	0.0005
	$(NotBlurry, Low)$	0.3	0.933	0.9144	0.0006
	$(NotBlurry, Normal)$	0.4	0.9448	0.9260	0.0007
	$(NotBlurry, Strong)$	0.3	0.9305	0.9121	0.0005
b_R	$(Blurry, Low)$	0.09	0.9462	0.9274	0.0007
	$(Blurry, Normal)$	0.12	0.9365	0.9179	0.0006
	$(Blurry, Strong)$	0.09	0.9435	0.9247	0.0007
	$(NotBlurry, Low)$	0.21	0.9411	0.9224	0.0006
	$(NotBlurry, Normal)$	0.28	0.9456	0.9268	0.0007
	$(NotBlurry, Strong)$	0.21	0.9434	0.9247	0.0007
b_R^f	$(Blurry, Low)$	0	0.9462	0.9274	0.0007
	$(Blurry, Normal)$	0	0.9365	0.9179	0.0006
	$(Blurry, Strong)$	0	0.9435	0.9247	0.0007
	$(NotBlurry, Low)$	0.3	0.9411	0.9224	0.0006
	$(NotBlurry, Normal)$	0.4	0.9456	0.9268	0.0007
	$(NotBlurry, Strong)$	0.3	0.9434	0.9247	0.0007
b_{NV}	$(Blurry, Low)$	0.09	0.9561	0.9371	0.0008
	$(Blurry, Normal)$	0.12	0.9542	0.9353	0.0008
	$(Blurry, Strong)$	0.09	0.9523	0.9334	0.0008
	$(NotBlurry, Low)$	0.21	0.9562	0.9372	0.0008
	$(NotBlurry, Normal)$	0.28	0.9596	0.9405	0.0009
	$(NotBlurry, Strong)$	0.21	0.9543	0.9354	0.0008
b^+	-	-	1	0.9798	0.01

Table 9.15.: Comparison of the similarity of the success probabilities of the sensitivity models with the success probability predicted by our reliability prediction approach.

this means that the success probabilities of our reliability analysis must indicate the very same monotonic property. Therefore, consider Figure 9.16b which shows the predicted success probabilities $rel(M_{C_b} \mid \varphi_B, \varphi_{Bl})$ (y-axis) for each steering angle prediction model (x-axis) and uncertainty pair $(\varphi_B, \varphi_{Bl})$. If we compare Figure 9.16a and Figure 9.16b, we can observe exactly this property. Note that we omitted b^- in Figure 9.16 as the success probability in both cases is 0 for all pairs $(\varphi_B, \varphi_{Bl})$. More specifically, the line plot of

Figure 9.16b is simply shifted in the negative direction of the y-axis (because the reliability predictions are slightly smaller than the success probabilities of the sensitivity models) but indicates the very same behaviour. Therefore, we conclude that plausibility assertions (9.2) and (9.3) must hold.

Note that we have deliberately not considered or discussed the case where the filtering pattern was applied, e.g. $M^f_{C_{b_R}}$ and $M^f_{C_{b_C}}$ where an additional filter is inserted to contain the effect of uncertainties. The main reason for this is that the previous discussion referred exclusively to the success probability of the sensitivity models and the models predicted by our approach. However, recall that the filtering pattern corresponds to an uncertainty-specific architectural countermeasure which has no direct impact on the success probability of the sensitivity model itself but rather on the impact of uncertainties (see section 7.1.3.2). This can also be observed in Figure 9.16b where the sensitivity models of b^f_C and b^f_R are identical to the sensitivity models of b_C and b_R. However, when looking at the probabilities of the uncertainties (i.e. the third column), it can be seen that they are different compared to the others. In this case, we modelled the effect of the filter component to be deterministic. More precisely, we assumed that the filter always eliminates the blur from the image, i.e. $f(Blurry) = f(NotBlurry) = NotBlurry$. We made this strict improvement assumption because we can now expect $rel(M_{C_{b_C}}) < rel(M^f_{C_{b_C}})$ and $rel(M_{C_{b_R}}) < rel(M^f_{C_{b_R}})$. Note from equation (7.4) that $rel(M_{C_b}) = \sum_{(\varphi_B,\varphi_{Bl})\in\Phi} Pr(X_{\varphi_B} = \varphi_B, X_{\varphi_{Bl}} = \varphi_{Bl}) \cdot rel(M_{C_b} \mid \varphi_B, \varphi_{Bl})$. However, since the application of the filter eliminates image blur, the corresponding probabilities of uncertainty tuples which contain *Blurry* have zero probability and do not affect the success probability of b. This effect is reflected when predicting the corresponding success probabilities of the entire system:

$$rel(M_{C_{b_C}}) = 0.9170 < 0.9183 = rel(M^f_{C_{b_C}})$$
$$rel(M_{C_{b_R}}) = 0.9242 < 0.9249 = rel(M^f_{C_{b_R}})$$

Thus, systems that contain the filter pattern are more reliable than systems that do not, which is in line with our expectations.

9.4. Evaluating Self-Adaptive Systems to Safeguard AI Components

In this section, we focus on validation goal 4. More specifically, we expand the validation of the last section (i.e. reliability prediction of static AI-enabled systems) to self-adaptive systems in which an AI component is to be safeguarded. For the validation, we focus on two case study systems, namely the Udacity self-driving car challenge and the HRI system of section 1.5.3. All results and validation artefacts can be found in reference [157].

9.4.1. Udacity Self-Driving Car Challenge

This section is similar in structure to the previous sections where *SimExp* was validated. That is, we outline how the *SimExp* method is instantiated in the context of the Udacity case study system and discuss the experiment setup and results afterwards. First, however, we discuss how we expand the Udacity case study system to a self-adaption scenario.

9.4.1.1. Self-Adaptive Filtering of Input Images

In section 9.3, we introduced the Udacity self-driving car challenge where AI models (or more specifically deep learning) have been employed to predict steering angles for a self-driving car. In this section, we expand the scenario by considering the same generic software architecture from section 9.3.2 but assume that the filtering pattern is applied to contain the effect of image blurring. Although safety is an important aspect of self-driving cars, the performance of the system, or the ability to process in real-time, must be ensured at the same time [116]. Therefore, the filter component is dynamically activated in situations where an increased image blur is observed, e.g. by leveraging image blur detection methods [189, 132, 117].

We consider the steering angle prediction models b^-, b_C, b_R and b^+. Note that we exclude b_{NV} as it represents no AI component which must be safeguarded but an already applied architectural pattern to enhance the reliability of the system. Clearly, one may combine the n-version and filtering pattern to obtain an even more reliable system; however, we do not consider this scenario as it has no relevance for the validation of goal 4 but only leads to more complexity.

The motivation for dynamically activating and deactivating the filter component is to balance the reliability and performance attributes of the system. Thus, one would expect that both attributes are considered in the reward function. The consideration of performance as an additional influencing factor, however, does not allow the validation of plausibility assertion (9.4) where the assertion is exclusively based on reliability attributes. Therefore, we only consider reliability as a single quality attribute.

9.4.1.2. Instantiation of *SimExp*

For the instantiation of *SimExp*, we could reuse various artefacts which have already been generated in section 9.3.

Initial Architecture Model We reuse the same generic software architecture of a self-driving car as presented in section 9.3.2. As discussed, we assume that the filtering pattern is applied. Hereby, we configure the initial architecture such that the filter component is initially deactivated.

Adaptations We consider two adaptations, namely the activation and deactivation of the filter component. We implemented both adaptations as model transformations which can be selected and applied by an adaptation strategy.

Environment Model Recall that the environmental dynamics of a self-adaptive system consists of a static part (describing the initial distribution of the environment) and a dynamics part (describing the temporal extension of the static environment) which in conjunction constitutes a DBN. The environment $\mathcal{E}$ of the Udacity case study system encompasses two environmental variables namely, image blur φ_{Bl} and brightness variations φ_{Bl}. That is, an environmental state is formed by two random variables: $E := (X_{\varphi_{Bl}} X_{\varphi_{Bl}})$. In the context of the sensitivity model from section 9.3, we already modelled the static environment, i.e. the distributions $P(X_{\varphi_{Bl}})$ and $P(X_{\varphi_B})$. Therefore, only the stochastic evolution (i.e. the dynamic part) must be modelled. Just as for $P(X_{\varphi_{Bl}})$ and $P(X_{\varphi_B})$, we assumed w.l.o.g. the dynamic distributions which can be looked up in reference [157].

Adaptation Strategies We consider three adaptation strategies, namely the non-adaptive strategy $\pi_{\delta_\emptyset}$, the randomised filtering strategy π_{Ran} and the image blur mitigation strategy π_{Mit}. The strategy $\pi_{\delta_\emptyset}$ again simulates the behaviour of a static system where no adaptation is triggered at all. The two remaining strategies are used to validate the steering angle prediction model by dynamically activating the filter component. The randomised filter strategy π_{Ran} activates or deactivates the filter component randomly and independently of the current environmental state; the image blur reduction strategy π_{Mit} activates the filter component whenever increased image blur is observed and deactivates it otherwise.

It should be noted that the strategies are not thoroughly engineered. It is arguably not sufficient for self-driving cars to just observe the degree of blurring and activate a filtering component when needed. However, it is not within the scope of this thesis to research suitable adaptation strategies specifically for autonomous systems, nor are we experts in this field. However, regardless of the complexity of the strategies, our *SimExp* method must preserve the plausibility assertion (9.4). Recall that the assertion requires that for any strategy π which is individually applied to safeguard an AI black-box b from a set of AI black-box models $\{b_1, b_2, \dots\}$ with $b_1 < b_2 < \dots$, the same ordering must apply for adaptation strategy where only b is modified, i.e. $\pi[b_1] < \pi[b_2] < \dots$. For the assertion, however, the complexity of the adaptation strategy is of no relevance. That is to say, if we can validate that assertion (9.4) holds for the adaptation strategies under consideration, it must immediately follow that it also holds for any strategy of arbitrary complexity. Moreover, we have already validated the appropriateness of our *SimExp* method for more complex adaptation strategies in section 9.2.

Reward Function As mentioned before, we only account for the reliability of the system to determine the reward because the consideration of additional quality attributes might distort the order of evaluated strategies. More specifically, the reward function returns

Strategy	$\mathbb{E}_{\pi}[X_{G_0}]$				$\sum_i r_i$			
	b^-	b_C	b_R	b^+	b^-	b_C	b_R	b^+
$\pi_{\delta_\emptyset}[b]$	0.0	84.5	85.2	89.3	0.0	13647	13719	14555
$\pi_{Ran}[b]$	0.0	80.5	79.2	85.2	0.0	13664	13733	14555
$\pi_{Mit}[b]$	0.0	81.8	82.4	86.3	0.0	13663	13733	14555

Table 9.16.: Overview of the expected and total rewards of each strategy for the Udacity self-driving car case study.

simply the predicted success probability of the system by using our reliability prediction approach for AI-enabled systems.

$$r_{\mathcal{S}} : \mathcal{S} \times \Delta \times \mathcal{S} \to [0,1], (S, \delta, S') \mapsto rel(M_{C_b} \mid \varphi_B, \varphi_{Bl}) \tag{9.13}$$

Note that $S' := (C_b, E)$ with $E := (X_{\varphi_{Bl}}, X_{\varphi_B})$.

Interdependency Assumption of Architecture and Environment We assumed no interdependency between the system or architectural configuration and the environment, i.e. the dynamics of the self-adaptive system is purely environmental-driven (recall independence assumption (6.9) on page 144). This seems reasonable because no architectural configuration changes the way how the environment (i.e. brightness and blurring effects) is stochastically evolving.

Prediction of quality attributes As mentioned before, we predict the probability of success for any sampled state S. Therefore, we use our reliability prediction approach for AI-enabled systems, i.e. $rel(M_{C_b} \mid \varphi_B, \varphi_{Bl})$.

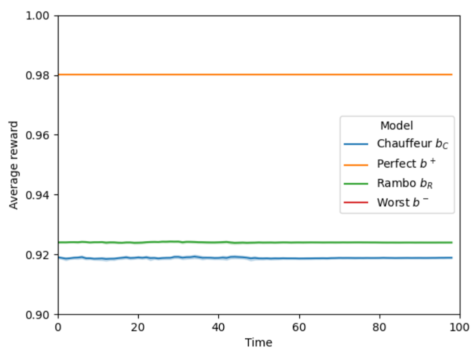
9.4.1.3. Experiment Setup

We evaluated each strategy w.r.t. any steering angle prediction model b^-, b_C, b_R and b^+. Hereby, we sampled for each strategy 150 trajectories of length 100 making a total of 15000 sampled states. In the next section, we discuss the results of the validation.

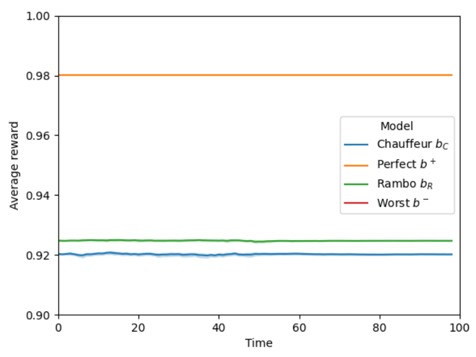
9.4.1.4. Experiment Results

Figure 9.17 depicts the accumulated and averaged rewards generated by each strategy. It can be seen that the accumulated and averaged rewards of any strategy are fairly stable and converge towards a fixed reward. Complementing Figure 9.17, Table 9.16 lists the expected and total rewards for each strategy.

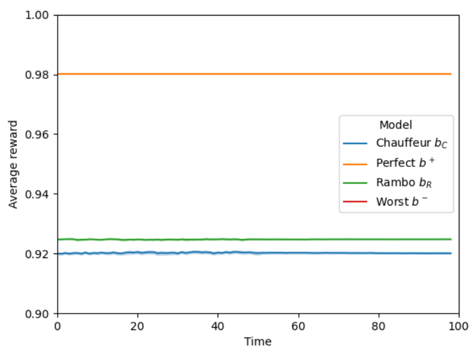
Note that each strategy which safeguards $\pi[b^-]$ is not visible on the figures because they generate zero-valued rewards (which seems reasonable because the success probability



(a) Non-adaptive strategy $\pi_{\delta_\emptyset}[b]$.



(b) Randomised filter strategy $\pi_{Ran}[b]$.



(c) Image blur mitigation strategy $\pi_{Mit}[b]$.

Figure 9.17.: *SimExp* results of each strategy individually evaluated for b^-, b_C, b_R and b^+.

of b^- is always 0). From section 9.3.6, we know that the accuracy of the steering angle prediction models corresponds to $b^- < b_C < b_R < b^+$. That is, we expect for each strategy the following order w.r.t. the expected reward $\mathbb{E}_\pi[X_{G_0}]$:

$$\pi[b^-] < \pi[b_C] < \pi[b_R] < \pi[b^+] \tag{9.14}$$

While in Figure 9.17 and almost all expected rewards shown in Table 9.16 reflect the very same order, we can observe a slight deviation of strategy $\pi_{Ran}[b]$. More specifically the expected reward of $\pi_{Ran}[b_C]$ is higher than $\pi_{Ran}[b_R]$, i.e. $\pi_{Ran}[b_C] > \pi_{Ran}[b_R]$. On the contrary, however, if we look at the total rewards generated by $\pi_{Ran}[b_C]$ and $\pi_{Ran}[b_R]$, we can observe the expected order. Therefore, we suspect that the expected rewards of the experiment did not converge properly (which can also be a result of the similar success probabilities of b_C and b_R). If the convergence assumption holds, we must merely sample more states, calculate the expected reward again and should observe the proper order. Therefore, we sampled for $\pi_{Ran}[b_C]$ and $\pi_{Ran}[b_R]$ 20000 states and calculated the expected reward again. Hereby, we obtained $\mathbb{E}_{\pi_{Ran}[b_C]}[X_{G_0}] = 79.3769$ (with total reward 18219.2) and $\mathbb{E}_{\pi_{Ran}[b_R]}[X_{G_0}] = 80.6528$ (with total reward 18310.4) which confirms our suspicion.

Finally, the results show that dynamically activating and deactivating a filter component to remove blurring does not have a major impact on the reliability of the system. In fact, this is not surprising as we already observed a small improvement in reliability when using the filtering pattern. Looking at the results of the generated rewards without considering the performance effects, it is quite obvious that using a filter component does not seem to be a good decision as it does not add significant value to the overall reliability of the system, but rather degrades the performance. Nonetheless, the results show that plausibility assertion (9.4) is preserved by our *SimExp* method which is in line with our expectations.

9.4.2. Human-Robot-Interaction

In this section, we continue with validating goal 4 by considering the HRI case study system. Whereas in the last section, we regarded adaptation strategies where only the filter component has been dynamically activated, we expand the adaptation logic in this section by considering a second type of adaptation, namely switching the AI component. In this case, we have two AI components which can be dynamically exchanged by a self-adaptive system. Recall that the adaptation problem of the HRI system is to strike a balance between the performance and reliability attributes of the system. Therefore, the first AI component (hereinafter referred to as b_D) tends to be less reliable in prediction but less computationally expensive; in turn, the second AI component (hereinafter referred to as b_{Rob}) is more robust but computationally expensive. However, this must be taken into account by a self-adaptive system to balance both attributes as well as possible. Although the validation primarily concentrates on plausibility assertion (9.4), we complement the validation results by considering a second reward function which also takes into account the performance of the system (and not only the reliability as in the last section). Thus, we demonstrate how *SimExp* allows software engineers to make trade-off decisions.

9.4.2.1. Instantiation of *SimExp*

In this section, we outline how the *SimExp* method is instantiated.

Initial Architecture Model We modelled the software architecture of the HRI system (as presented in section 1.5.3) as PCM model. Initially, the filter component is deactivated. Moreover, we consider b_D as the default AI model which is initially instantiated in the system.

Adaptations As already mentioned, the first adaptation that we consider is activating or deactivating a filter component which reduces noise artefacts (just as in the last section). The second adaptation corresponds to switching the currently deployed AI component with a different one. As explained earlier, we consider b_D as the default AI model that can be dynamically exchanged by the more robust model but computationally expensive b_{Rob} and vice versa.

Environment Model Recall that in terms of the HRI case study system, we have two uncertainties, namely brightness φ_B and sensor noise φ_{SN}. Hereby, the value space $Val(X_{\varphi_B})$ is equally discretised as in the Udacity case study; the value space of $X_{\varphi_{SN}}$, however, is discretised by considering three sensor noise levels $Val(X_{\varphi_{SN}}) := \{Low, Medium, High\}$. The structure of the sensitivity model for both AI components is depicted on Figure 7.10. Just as in the Udacity case study, we assumed the distributions $P(X_{\varphi_B})$ and $P(X_{\varphi_{SN}})$ for the static environment and the dynamic environment (i.e. $P(X'_{\varphi_B} \mid X_{\varphi_B})$ and $P(X'_{\varphi_{SN}} \mid X_{\varphi_{SN}})$) as well. Again, it is important to note that we can assume w.l.o.g. any kind of distribution which governs the environmental dynamics as long as they are rigorously used for the evaluation of all strategies. The concrete distributions can be looked up in [157].

Adaptation Strategies We consider three adaptation strategies, namely the by now well-known non-adaptive adaptation strategy $\pi_{\delta_\emptyset}$, a randomised strategy π_{Ran} and finally a reliability-prioritised strategy π_{Rel}. As before, $\pi_{\delta_\emptyset}$ simulates the behaviour of a static software system.

The adaptation logic of the reliability-prioritised strategy π_{Rel} is depicted on Listing 9.3. For the sake of clarification, we simplified the strategy and also show only the analyse- and plan-phase of the adaptation strategy. In principle, analyse-phase checks whether the brightness and sensor noise levels are increased or deviating from what is considered to have no significant effect on the prediction of the AI component. If so, the plan-phase is initiated. If the brightness and sensor noise are not suspicious, but the system response time exceeds a fixed threshold ε, the plan-phase is also initiated. Otherwise, the strategy terminates and no adaptation is planned. Since the π_{Rel} strategy prioritises reliability, the planning phase first checks whether the reason for the adaptation is due to potentially malicious input values (e.g. increased sensor noise or varying image brightness) and selects appropriate adaptations. Otherwise, performance-improving countermeasures are taken that mainly reverse reliability-improving adaptations which are computationally expensive and degrade the response time of the system.

In contrast, strategy π_{Ran} randomly selects an applicable adaptation at each time, i.e. activating the filter component is not possible, if the filter is already activated.

Reward Function We use the same reward function as presented in section 9.4.1 (see (9.13)). Recall that plausibility assertion (9.4) is formulated based on the motivation to preserve reliability properties. Therefore, we can only consider reliability predictions (or the success probability of the system) to assess whether the assertions hold. However, since the challenge of a self-adaptive system is also to take performance-related attributes into account, we consider a second reward function:

$$\begin{aligned} r_{Perf} &: \mathcal{S} \times \Delta \times \mathcal{S} \to [0, 2], \\ (S, \delta, S') &\mapsto rel(M_{C_b} \mid \varphi_B, \varphi_{Bl}) + (norm_{rt} \circ rt)(S') \end{aligned} \tag{9.15}$$

To distinguish between both reward functions, we denote r_{Rel} the pure reliability-based reward function (i.e. (9.13)) and r_{Perf} the reward function from (9.15). Note that r_{Perf}

```java
public class ReliabilityPrioritizedStrategy extends
    ReconfigurationStrategy<QVToReconfiguration> {
    ...
    @Override
    protected boolean analyse(State source, SharedKnowledge knowledge) {
        if (getImgBrightness(knowledge) != "Normal" || getSensorNoise(knowledge)
            != "Low") {
            return true;
        }
        return getResponseTime(knowledge) > THRESHOLD_RT
    }

    @Override
    protected QVToReconfiguration plan(State source, Set<QVToReconfiguration>
        options, SharedKnowledge knowledge) {
        var highSensorNoise = getSensorNoise(knowledge) != "Low";
        var brightChange = getImgBrightness(knowledge) != "Normal";
        if (highSensorNoise && brightChange && isDefaultMLModelActivated) {
            return switchToRobustMLModel(options);
        } else if (highSensorNoise && !isFilteringActivated) {
            return activateFilteringReconfiguration(options);
        } else if (brightChange) {
            return QVToReconfiguration.empty();
        }

        if (getResponseTime(knowledge) > THRESHOLD_RT) {
            if (!isDefaultMLModelActivated) {
                return switchToDefaultMLModel(options);
            } else if (isFilteringActivated) {
                return deactivateFilteringReconfiguration(options);
            } else {
                return QVToReconfiguration.empty();
            }
        }
        return QVToReconfiguration.empty();
    }
}
```

Listing 9.3: Adaptation logic of strategy π_{Rel}

accounts for performance by considering the normalised response time. We use the normalisation function $norm_{rt}$ from (9.5) which normalises predicted response times to the range $[0, 1]$ w.r.t. some upper and lower response time bounds (i.e. by considering the upper response time bound (i.e. β^+_{rt} and β^-_{rt}). We solely use r_{Rel} when checking assertion (9.4); however, by considering r_{Perf}, we will demonstrate *SimExp* supports software engineers in the decision-making process.

Φ	b_D		b_{Rob}	
	Success	**Failure**	**Success**	**Failure**
$(High, High)$	0.85	0.15	0.95	0.05
$(High, Medium)$	0.9	0.1	0.97	0.03
$(High, Low)$	0.95	0.05	0.99	0.01
$(Medium, High)$	0.9	0.1	0.98	0.02
$(Medium, Medium)$	0.95	0.05	0.99	0.01
$(Medium, Low)$	0.99	0.01	1	0
$(Low, High)$	0.85	0.15	0.95	0.05
$(Low, Medium)$	0.9	0.1	0.97	0.03
(Low, Low)	0.95	0.05	0.99	0.01

Table 9.17.: The sensitivity models of b_D and b_{Rob}.

Interdependency Assumption of Architecture and Environment We assumed no interdependency between the system or architectural configuration and the environment, i.e. the dynamics of the self-adaptive system is purely environmental-driven (recall independence assumption (6.9) on page 144).

Prediction of quality attributes For this part of the validation, we consider two quality prediction tools/approaches. First, we use our reliability prediction approach for AI-enabled systems, i.e. $rel(M_{C_b} \mid \varphi_B, \varphi_{Bl})$. Second, we use SimuLizar to predict the response time of a given state S.

9.4.2.2. Experiment Setup

In the context of the HRI system, a Mask R-CNN [80, 2] has been used as a detection component. However, the component has been trained via transfer learning, i.e. the AI model was trained on a completely unrelated training dataset containing completely different images as in the HRI context and finally retrained for a smaller set of HRI-related training data. Therefore, we had not sufficient data to perform a sensitivity analysis. Instead, we assumed sensitivity models for the AI components b_D and b_{Rob} which are depicted on Table 9.17.

The monitorable space is defined as $\Phi := Val(X_{\varphi_B}) \times Val(X_{\varphi_{SN}})$. We have picked the individual sensitivity values such that for all uncertainty values b_{Rob} indicates higher success probabilities reflecting the circumstance that b_{Rob} is more robust regarding uncertainties compared to b_D. We also configured the PCM model such that the initial architecture configuration produces an acceptable response time, i.e. less than the fixed threshold ε, which we set to 0.1. On the other hand, we have chosen the reliability-improving adaptations (i.e. activation of the filter component and switching to b_{Rob}) such that they introduce a performance overhead, i.e. the response time exceeds the required threshold ε.

Name	Value	Description
β_{rt}^{+}	0.3	The upper response time bound used for response time normalisation, i.e. $norm_{rt}$.
β_{rt}^{-}	0.1	The lower response time bound used for response time normalisation, i.e. $norm_{rt}$.
ε	0.1	The response time threshold that must not be exceeded.
Number of simulations	100	Number of simulations per trajectory (compare with horizon from *SimExp*).
Number of trajectories	150	Number of trajectories to simulate.

Table 9.18.: Overview of the parameter setting for the HRI case study system. For some parameters, no description could be found.

In this way, we artificially generate a situation where performance and reliability compete with each other but must be balanced by an adaptation strategy in the best possible way.

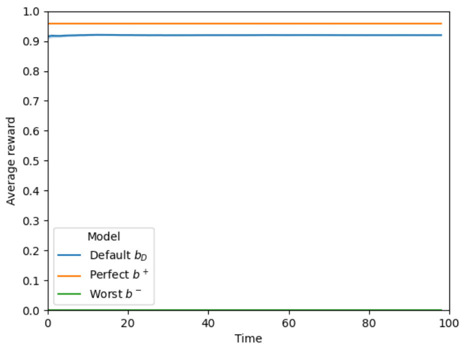
In the last section, we considered an ideal filter component that was able to deterministically remove any occurrence of image blur. For the HRI system, we assume that the filter is not able to remove all sensor noise. Therefore, we defined a probabilistic filter component that is not always able to successfully remove sensor noise. Depending on the level of sensor noise, the filter is more or less successful. More specifically, for *High* sensor noise, we assume that the filter can reduce the level to *Medium* 60% of the time and is unsuccessful to 40%; for *Medium* sensor noise, we expect the filter to reduce the sensor noise level to *Low* 70% of the time and be unsuccessful to 30% (where unsuccessful means that the sensor level remains unchanged). The probabilistic nature of the filter is reflecting the situation where sensor noise is not sufficiently reduced, or the filtered image is still likely to produce wrong predictions.

Finally, table Table 9.18 summarises the made setups for the experiment. Just as for the Udacity case study, we sampled 15000 states.

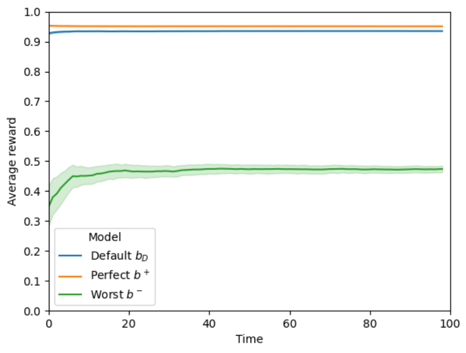
9.4.2.3. Experiment Results

As before, we calculated for each strategy the accumulated and averaged rewards of each trajectory. That is, for a strategy π, we apply $accum_{\pi}(N) := \frac{1}{N}\sum_{i=1}^{N} r_i$ to each trajectory (or rather the rewards generated by π of the trajectory). The results are depicted on Figure 9.18 as line plots which indicate the mean reward calculated by $accum_{\pi}(N)$ for each time step $t \in \{0, 1, \dots, N = 100\}$ and 95% confidence interval.

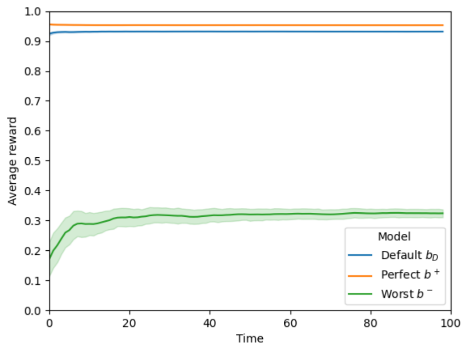
We considered three AI black-boxes which are to be safeguarded, namely b^{-}, b_D and b^{+}. Note that b_{Rob} is already considered to be a sufficiently robust (but at the cost of a high resource burden) AI model such that we focus only on b_D. Because of this high computational cost, b_{Rob} is rather to be considered in situations where the overall reliability of the system is at risk (and the filter component is not sufficient to maintain reliability



(a) Pure reliability based rewards of $\pi_{\delta_\emptyset}[b]$.



(b) Pure reliability based rewards of $\pi_{Ran}[b]$.



(c) Pure reliability based rewards of $\pi_{Rel}[b]$.

Figure 9.18.: *SimExp* results of the strategies $\pi_{\delta_\emptyset}[b]$, $\pi_{ran}[b]$, $\pi_{Rel}[b]$ evaluated w.r.t. r_{Rel}.

objectives). Moreover, when safeguarding b^+ it makes arguably no sense to switch the perfect AI model b^+ with b_{Rob}. For validation, however, only the preservation of assertion (9.4) matters and not the suitability of the adaptation logic itself.

Therefore, we have the following order of AI models: $b^- < b_D < b^+$. Based on the ordering, we expect to observe

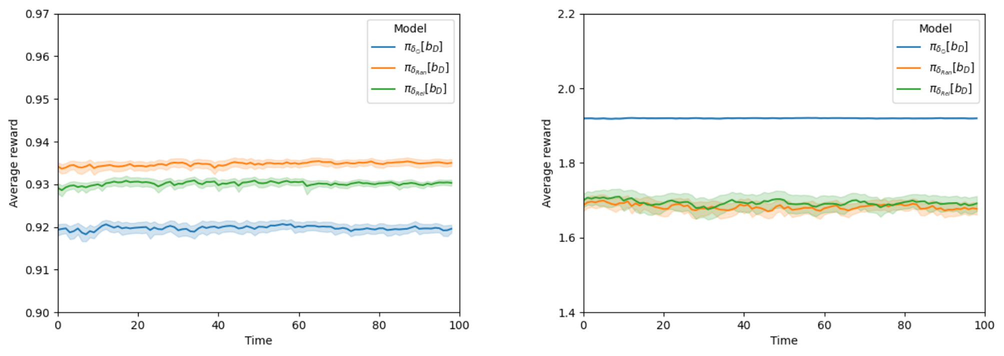
$$\pi[b^-] < \pi[b_D] < \pi[b^+] \tag{9.16}$$

w.r.t. the expected reward $\mathbb{E}_\pi[X_{G_0}]$. Looking at the results (the left side of Figure 9.18), we observe the very exact ordering. Complementary to the results of Figure 9.18 consider Table 9.19 which shows the expected and total rewards for each strategy. Just as in Figure 9.18, the expected rewards maintain the strategy ordering when considering varying AI models which conform to the expected ordering (9.16).

Finally, we evaluated the results by considering reward function r_{Perf} which also accounts for performance aspects. Let us now put ourselves in the position of a software engineer who has to choose one of the strategies for safeguarding b_D, i.e. $\pi_{\delta_\emptyset}[b_D]$, $\pi_{Ran}[b_D]$ and $\pi_{Rel}[b_D]$. Therefore, consider Figure 9.19 which shows the results of all strategies (i.e. $\pi_{\delta_\emptyset}[b_D]$, $\pi_{Ran}[b_D]$ and $\pi_{Rel}[b_D]$) applied to b_D by considering r_{Rel} (see Figure 9.19a) and r_{Perf} (see Figure 9.19b).

Strategy	$\mathbb{E}_\pi[X_{G_0}]$			$\sum_i r_i$		
	b^-	b_D	b^+	b^-	b_D	b^+
$\pi_{\delta_\emptyset}[b]$	0.0	80.1	85.1	0.0	13659	14250
$\pi_{Ran}[b]$	37.1	72.5	75.3	7038	13885	14124
$\pi_{Rel}[b]$	28.4	81.1	82.2	4813	13832	14149

Table 9.19.: Overview of the expected and total rewards of each strategy of the HRI case study.



(a) Results of $\pi_{\delta_\emptyset}[b]$, $\pi_{Ran}[b]$ and $\pi_{Rel}[b]$ based on r_{Rel}. **(b)** Results of $\pi_{\delta_\emptyset}[b]$, $\pi_{Ran}[b]$ and $\pi_{Rel}[b]$ based on r_{Perf}.

Figure 9.19.: *SimExp* results of the strategies $\pi_{\delta_\emptyset}[b]$, $\pi_{ran}[b]$, $\pi_{Rel}[b]$ w.r.t. r_{Rel} and r_{Perf}.

From the figure, it can be seen that the order of the strategies is changing when taking into account performance attributes. In summary, the results of our *SimExp* method show the highest expected reward for the strategy $\pi_{\delta_\emptyset}[b_D]$ (≈ 1.92); in comparison, $\pi_{Ran}[b_D]$ (≈ 1.68) and $\pi_{Rel}[b_D]$ (≈ 1.7) perform worse. That is, the most reasonable design decision would be to engineer the system without self-adaptation capabilities. For the reward function r_{Rel}, the strategy $\pi_{Ran}[b]$ performed best, arguably making more frequent reliability-improving adaptations due to the random component of the strategy. However, the strategy is not balancing the performance and reliability attributes ideally which results in frequent performance violations. Strategy $\pi_{Rel}[b]$, on the contrary, applies only reliability-improving adaptations if necessary which leads to better responsiveness of the system. The strategy $\pi_{\delta_\emptyset}[b_D]$, on the other hand, always remains in the initial architecture configuration, which is known to be performance-friendly, i.e. the configuration achieves the best possible response time and thus does not show any performance violations. In addition, the AI model b_D already has acceptable predictive accuracy which leads to high rewards overall. Note however that the results are dependent on the defined reward function and the behaviour of the environment. For example, if we assume a more dynamic environment and the reward function favours the reliability of the system, the overall result could be different.

Finally, note that the expected rewards of all strategies safeguarding b_D (see the third column of Table 9.19) deviate from the averaged accumulated rewards of Figure 9.19a. In terms of plausibility assertion (9.4) this has no consequences because the assertion is defined over the same strategy π where only the safeguarded AI model b varies. Nonetheless, one would expect the same ordering when comparing the expected reward with the averaged accumulated reward. However, when considering the total reward (see the sixth column of Table 9.19), we observe the very same strategy ordering as in Figure 9.19a. Just as in section 9.4.1, we attribute this to the convergence behaviour. More specifically, while $\pi_{\delta_\emptyset}$ and π_{Rel} dictate a specific adaptation logic, the resulting sampled trajectory space often contains more characteristic trajectories. On the contrary, strategy π_{Ran} is completely based on random such that no characteristic trajectories exist but solely trajectories that follow no pattern or logic. This exploratory property of π_{Ran} leads to an increased trajectory space from which samples are generated (containing no characteristic trajectories but only random ones). However, this is accompanied by slow convergence behaviour when estimating $\mathbb{E}_{\pi_{Ran}}[X_{G_0}]$.

9.5. Discussion of Results and Research Questions

In this section, we discuss the results of the validation and answer the research questions accordingly. Moreover, we discuss the threats to validity.

9.5.1. Goal Achievement

Having carried out the validations for each validation goal, we are now discussing the results and achievement of each validation goal. As a final remark, we would like to emphasise that we consider the case study systems load balancing and HRI rather as supplementary validations. In the case of the load balancing system, this is because SimuLizar is not fully comparable to *SimExp* (as SimuLizar is scenario-based); in the case of the HRI system, we had to assume sensitivity models and could not analyse the existing AI models. Nevertheless, both case studies are considered an integral part of the validation, which reinforces the validity of our results.

9.5.1.1. Modelling the Environmental Dynamics

For validation goal 1, we aimed to validate the applicability of our *EnvDyn* metamodel. In contrast to the other validation goal, there is no dedicated section where goal 1 is validated. This is because the validation of all other goals implicitly validates goal 1.

For the validation goal, we focused on two questions: First, whether we can instantiate and apply our *EnvDyn* metamodel domain independently (i.e. question 1.1) and second, whether the essential characteristics of the operating environment are captured (i.e. question 1.2). For both questions, we considered the metrics used in the remaining validation goals as their results directly provide answers for the validation questions 1.1 and 1.2. We were able to instantiate the *EnvDyn* metamodel for all considered case study systems. That is, we modelled the operating environment of the individual self-adaptive systems considered for each case study system (each of which belongs to a different domain). Therefore, we identified the essential variables of the environment which affect the quality attributes of the system (e.g. activation probability, SNR and wireless interference in the context of the DeltaIoT case study system), modelled their initial/static distribution (i.e. the BN describing the static environment) and modelled the environmental dynamics (i.e. the temporal expansion captured by a DBN). Moreover, the sensitivity models of the AI components are also modelled with our *EnvDyn* metamodel (modelled as BN).

Based on the modelled operating environments, we conducted the validation of the goals 2, 3 and 4 as intended. As presented in the respective sections, we showed that the validation results are in line with our expectations and are considered to be valid. Therefore, we conclude that our *EnvDyn* metamodel provides the required capabilities to model and capture the essential characteristics of an operating environment. In summary, we conclude that the *EnvDyn* metamodel is applicable in the sense that the operating environments can be modelled domain-independently and that the created instances are sufficiently accurate such that they allow the evaluation of adaptation strategies (i.e. goals 2 and 4) and the prediction of reliability attributes of AI-enabled systems (i.e. 3).

9.5.1.2. Evaluating Adaptation Strategies of Self-Adaptive Systems

In terms of validation goal 2, we validated our *SimExp* method in the context of two case study systems, namely the DeltaIoT and the load balancer system. Moreover, we formulated three validation questions (i.e. questions 2.1-2.3).

Question 2.1 was concerned with whether our *SimExp* framework generates comparable evaluation results compared with a domain-specific simulator. We defined two metrics (namely metrics 2.1.1 and 2.1.1) to elaborate the question. For metric 2.1.1, we considered the DeltaIoT system where we compared the evaluation results of adaptation strategies produced by *SimExp* with the results of the domain-specific DeltaIoT simulator. The results demonstrated that the rank of the strategies was preserved by *SimExp.* Although we found deviations for a single strategy, the anomaly can be attributed to prediction deviations of the Prism model-checking tool used for energy consumption prediction. Furthermore, we have shown that our *SimExp* method provides accurate evaluation results for this strategy under certain conditions. For metric 2.1.2, we instantiated the *SimExp* method in the performance domain, i.e. the load balancing system based on the ZNN.com system. In the case study, we considered SimuLizar as a domain-specific simulator to predict the quality of adaptation strategies in terms of response time. However, SimuLizar is a scenario-based simulation tool which is only partially comparable with our *SimExp* method. Nonetheless, we defined three evaluation scenarios and three adaptation strategies for which SimuLizar could evaluate each strategy individually. We transformed the scenarios into deterministic trajectories which we used to evaluate each adaptation strategy with *SimExp.* We observed that our *SimExp* framework produced the same evaluation results as SimuLizar, i.e. the same quality rank of the strategies.

In validation question 2.2, the goal was to validate whether our *SimExp* method supports software engineers in decision-making. Hereby, we considered the same metrics as for question 2.1. Because the results of the metrics demonstrated that our *SimExp* frameworks generate the same strategy rank, we can conclude that the framework allows for the comparison of distinct strategies. Moreover, for the load balancer system we considered two adaptation strategies of the same adaptation strategy family. Because also in this case, the proper adaptation strategy rank was maintained, we can also conclude that design decisions within an adaptation strategy family can be evaluated. Similarly, we considered several variants of an adaptation strategy in the DeltaIoT system (corresponding to design decisions within a strategy family) and evaluated them accordingly, e.g. different energy consumption thresholds.

In validation question 2.3, we focused on whether *SimExp* enables the comparison of self-adaptive and static software systems. Again, we considered the same metrics. Recall that we have always considered the adaptation strategy $\pi_{\delta_\emptyset}$ in each case study system, which simulates the behaviour of a static software system. Since both, DeltaIoT and SimuLizar allow the evaluation of static systems, we could validate whether *SimExp* generates the same ranks for a set of adaptation strategies including $\pi_\emptyset$ and compare the results with DeltaIoT and SimuLizar. In doing so, we have shown that our *SimExp* method allows the comparison between self-adaptive systems and a static software system. Therefore, we conclude that *SimExp* provides means to support software engineers in deciding whether self-adaptation capabilities are necessary or whether a static system already meets the quality requirements.

Complementary to the results of both case study systems, we showed that *SimExp* preserves plausibility assertions when evaluating adaptation strategies for safeguarding AI black-box components (see validation goal 4 later). The results not only demonstrate the general validity of *SimExp* but additionally demonstrates the ability to compare distinct strategies and design decisions within strategy families. In total, we considered nine different adaptation strategies (considering all case study systems; three per case study, with strategy $\pi_\emptyset$ being the same in all case studies) to solve the adaptation problem of each case study. Taking into account the results of all metrics for each validation question, we conclude that MDPs are appropriate analytical models to evaluate adaptation strategies.

9.5.1.3. Reliability Analysis of AI-enabled Systems

Regarding validation goal 3, we considered the Udacity challenge for predicting steering angles of self-driving cars. Because our approach is based on an existing reliability approach (without changing the core prediction logic), we focused on plausibility assertions which must be preserved by our approach. We considered three validation questions, i.e. questions 3.1-3.3.

In validation question 3.1, we addressed whether sensitivity models are appropriate representations of AI models when predicting system-level reliability attributes. We tackled the question by considering two metrics, i.e. metrics 3.1.1 and 3.1.2. For the metric 3.1.1, we measured the similarity of the individual success/failure probabilities of each sensitivity model to the resulting success/failure probabilities of the reliability prediction. Since only the success/failure probabilities of the sensitivity models affect the reliability prediction, both success/failure probabilities (i.e. those of the sensitivity model and those of the reliability prediction) must have some similarity. Therefore, we measured the similarities using the Bhattacharyya distance. In all cases, we determined values close to 0, which means that the distributions are very close or similar. In addition, we considered metric 3.1.2 which is associated with plausibility assertion (9.1). Regarding plausibility assertion (9.1), the results demonstrated that for a set of steering angle prediction models which induce an accuracy order (w.r.t. some performance measure), we could observe the same order in the reliability predictions. That is, for any two AI models b and b' with $b > b'$ (i.e. b gives more accurate prediction results compared to b'), the predicted success probability of the overall system with b is higher than for the same system with b'. Given the results of both metrics, we conclude that sensitivity models are adequate representations of AI models in terms of reliability prediction.

In validation question 3.2, we focused on whether our reliability prediction approach reflects reliability attributes of AI components. Hereby, we consider metric 3.2.1 which is associated with plausibility assertions (9.1)-(9.3). Plausibility assertion (9.1) has been already validated for the previous question. For plausibility assertions (9.2) and (9.3), we similarly demonstrated that the individual conditional success probabilities of a sensitivity model (conditioned on a particular uncertainty tuple) are preserved by the individual conditional success probabilities of our reliability prediction.

Finally, in question 3.3, we validated whether our relaibility prediction approach supports software engineers in decision-making by considering metric 3.3.1. The metric is associated with the assertions (9.2) and (9.3). We have considered two architectural patterns (namely the filter and the n-version patterns), which we have applied and analysed in terms of the reliability attributes. We have taken advantage of the fact that architectural patterns (when applied as described in section 7.1) act exclusively on the sensitivity model of an AI model b. Consequently, this can be thought of as generating new sensitivity models for which the assertions (9.2) and (9.3) must still hold. The results of the validation confirm these effects and demonstrates that both assertions are preserved.

In summary, we consider the validation goal 3 as achieved. We were able to validate all plausibility assertions and answer each validation question positively. Therefore, we conclude the plausibility of our reliability prediction.

9.5.1.4. Evaluating Adaptation Strategies of Self-Adaptive Systems to Safeguard AI Black-Box Components

Finally, for validation goal 4, we expanded the validation of goal 3 to self-adaptive systems. In doing so, we have considered a single validation question (namely question 4.1) which elaborates whether our expanded *SimExp* framework allows the evaluation of adaptation strategies specifically designed to safeguard AI components. More precisely, we aim to validate whether reliability-specific attributes are sufficiently reflected in the overall evaluation result of a strategy produced by *SimExp*.

For validation question 4.1, we have considered an additional plausibility assertion (more precisely assertion (9.4)) which states in simplified terms that for any two AI models b and b' with $b > b'$ it follows that $\pi[b] > \pi[b']$, i.e. the strategy π applied to both AI models must have a higher expected reward for the safeguarded model with higher accuracy. We determined the validity of our approach regarding assertion (9.4) by considering various adaptation strategies and AI models in two case stud systems, namely the Udacity self-driving car challenge and the HRI system. For both case studies, we demonstrated that assertion (9.4) has been preserved by our approach.

The results are complementing the results of validation goal 2 and allow the conclusion that our *SimExp* method is also capable to evaluate adaptation strategies for safeguarding AI black-box components.

9.5.2. Answering the Research Questions

Based on the validation results, we are now able to answer the research questions. Recall that the research questions are structured by one main research question followed by several sub-research questions which must be answered individually to answer the main research question. Therefore, we first start to answer the sub-research question and discuss the main research question afterwards.

Research Question 1 We formulated research question **RQ1** as follows:

> **Research Question 1:** How to evaluate adaptation strategies of self-adaptive systems at design-time regarding the ability to meet quality objectives?

The first research question is mainly concerned with the question of how to evaluate adaptation strategies at design-time. We have formulated several sub-research questions whose individual answers enable us to answer the main question:

> **Research Question 1.1:** How can environmental dynamics be formalised domain-independently at design-time?

To evaluate adaptation strategies at design-time, one has to model the operating environment or environmental dynamics in which a self-adaptive system operates. Moreover, we don’t want to restrict ourselves to a certain domain but rather evaluate adaptation strategies domain-independently. This is essentially what research question **RQ1.1** is about. To tackle this issue, we introduced our *EnvDyn* metamodel based on the semantics of template-based probabilistic models which allow the instantiation of probabilistic structures domain-independently. In the validation, we showed that our metamodel could be instantiated for all domains of the four case studies. Therefore, we consider the sub-research question to be sufficiently addressed by our *EnvDyn* metamodel.

Research Question 1.2: What is an appropriate level of abstraction to represent the environmental dynamics domain independently? By appropriateness, we mean that

- adaptation strategies can be analysed at design-time with sufficient accuracy.
- environmental state spaces can be described flexibly and compactly.

The sub-research question **RQ1.2** relates to the level of abstraction that such a formal modelling language must provide, i.e. the ability to describe large state spaces and still be meaningful enough to evaluate adaptation strategies. As already discussed for **RQ1.1**, we address the problem at a high abstraction level by using template-based probabilistic models (which generalise the framework of DBNs). To describe the environmental state spaces compactly and flexibly, we considered each environmental variable as a random variable which can be related to each other, i.e. they form a network of random variables or more precisely a DBN. We assumed that the value spaces of the individual random variables are discrete which reduces the state space tremendously. The discretisation level can be controlled by a domain expert such that the complexity of the state space is controllable as well. Finally, the decomposability property of DBNs as well as the stationary and Markov assumptions that apply to DBNs, allow the compact modelling of environmental state spaces in a simple and human-understandable way. Because our *EnvDyn* metamodel adheres to the semantics of DBNs, we can leverage these characteristics to describe large state spaces compactly and flexibly. In fact, the validation results confirm our argumentation. For each case study, we were able to describe the dynamics of the environment by a compact set of discrete probability distributions. In particular, for the DeltaIoT system (which has a large environmental state space), we have demonstrated how large and complex-structured state spaces can be described and handled using our *EnvDyn* metamodel and the associated concepts. Also, for each case study, we produced valid results which are all dependent on the *EnvDyn* metamodel. Therefore, we conclude that our *EnvDyn* metamodel sufficiently addresses **RQ1.2**.

Research Question 1.3: What is an appropriate analytical model to enable design-time analyses of self-adaptive systems?

In terms of **RQ1.3**, we addressed the question by using MDPs as analytical models. More specifically, we instantiated the generic framework of MDPs in the context of self-adaptive systems. The rationale for using MDPs is twofold: first, they are prevalent models to capture the dynamics of self-adaptive systems; second, there are various approaches which build upon MDPs that can be used for adaptation strategy evaluation. Especially for the second point, we reused concepts from dynamic programming and Monte Carlo methods. More specifically, we used Monte Carlo prediction to evaluate adaption strategies based on the formal semantics of MDPs. With regard to the validation results, using MDPs as analytical models is promising. In terms of the DeltaIoT and load balancing case study systems, our MDP-based *SimExp* method produced the same ranks as their domain-specific counterpart, i.e. the DeltaIoT simulator and SimuLizar. Also, for the Udacity challenge and the HRI case study system, our *SimExp* method maintained the plausibility assertion required to consider the results valid.

Research Question 1.4: Are the predictions sufficiently accurate to yield plausible results?

In terms of accuracy, we showed that for the DeltaIoT and load balancing case study system our *SimExp* method was able to produce the same strategy ranks. Therefore, we argue that our *SimExp* method produces sufficiently accurate predictions and thus plausible results. Note that we do not aim to achieve equal or better accuracy as domain-specific simulators but maintain a certain degree of accuracy sufficient to compare and evaluate distinct adaptation strategies as well as design decisions.

However, when looking at the prediction results of the quality attributes themselves, we experienced some deviations compared to the domain-specific simulators. In the DeltaIoT case study, for example, the energy consumption predictions deviated significantly from the predictions of the DeltaIoT simulator. Nevertheless, the main characteristics of the predictions were maintained (i.e. if an architectural configuration causes high or low energy consumption, we observed this in both *SimExp* and the DeltaIoT simulator). Therefore, we conclude that at least for adaptation strategy evaluation, the results are sufficiently accurate and plausible (which also depends on the quality attribute prediction tool used).

Overall, we conclude that **RQ1** can only be partially answered. Effectively, the results indicate that our *SimExp* method enables the evaluation of adaptation strategies by considering quality objectives as a primary source to determine the overall quality of a strategy. The validation, however, revealed a couple of weaknesses in our *SimExp* method. The first problem relates to efficiency. In general, the efficiency of *SimExp* is unproblematic (since the horizon and the number of sample states per trajectory are fixed) if no convergence criteria are defined; in this case, the efficiency depends on the complexity of the state space (which is a general theoretical problem in this area). However, since our *SimExp* framework is dependent on quality attribute prediction tools to determine a reward (e.g. SimuLizar or Prism), the efficiency of *SimExp* is also dependent on the efficiency of the prediction tools. This was particularly evident in the validation of the DeltaIoT system where we used Prism as a prediction tool. Therefore, the selection of the prediction tools is crucial in terms of efficiency. The same applies to the accuracy of the prediction tools, e.g. Prism indicated deviations in predicting the energy consumption of the DeltaIoT system. Moreover, comparable inaccuracies have been observed in the context of the load balancer case study system and SimuLizar. The *SimExp* method suffers information loss due to discretisation, i.e. the continuous simulation of SimuLizar maintains performance-relevant properties while *SimExp* applies a simulation for every state such that the simulation context (and thus performance-relevant properties) is lost after each simulation.

However, we consider the *SimExp* framework not as a fully-fledged approach which is applicable for any domain but rather as a method, one can take into account when evaluating adaptation strategies for a specific domain and adaptation problem. That is, in some domains or scenarios further efforts have to be made to fully apply the *SimExp* method. For example, for the DeltaIoT case study, we have additionally created an auxiliary metamodel to adequately capture the adaptation parameter. Therefore, we argue that in some cases *SimExp* needs to be enriched by additional concepts (such as additional metamodels or state-preserving simulations) to cope with domain-specific particularities, but that the basic methodology of *SimExp* remains unaffected. Furthermore, due to the large number of possible domains where self-adaptive systems are applicable, it cannot be shown that the *SimExp* method is applicable in all scenarios. However, this was also not in the scope of this work but rather to gain insights into the general feasibility of such an approach/method. Nonetheless, the results of the validation showed that the *SimExp* method provides a framework for the evaluation of adaptation strategies.

Research Question 2 We formulated research question **RQ2** as follows:

Research Question 2: How can software systems that contain AI black-box components be evaluated in terms of meeting reliability attributes at design-time?

In the context of the research question, we investigated the following sub-research question:

Research Question 2.1: How to deal with the hidden state problem of AI black-box components?

Recall that the hidden state problem of AI black-box components refers to the inability to observe the true state of an AI component. We addressed the problem by generating a sensitivity model for each AI black-box component. The sensitivity model approximates the predictive uncertainty w.r.t. a set of uncertainties, e.g. image brightness or image blur. Based on the sensitivity model, we circumvent the problem of knowing the true state of the AI component but consider the failure potential (i.e. predictive uncertainty) which is most relevant when we deal with reliability attributes. Since the validation results are in line with our expectations (i.e. all plausibility assertions were preserved), we conclude that sensitivity models are a suitable means to represent AI components during reliability analysis and to deal with the hidden state problem.

Research Question 2.2: How to systematically consider the influence of predictive uncertainty and causally related environmental variables in the reliability prediction?

Recall that we distinguished between first-order and second-order uncertainties. First-order uncertainty refers to predictive uncertainty. Second-order uncertainties relate to monitorable environmental factors or disturbances in the input data that potentially lead to incorrect predictions and allow conclusions to be drawn about the true state of the AI component; **RQ2.2** focuses on second-order uncertainties. As for **RQ2.1**, we addressed the problem by representing AI components with sensitivity models. Sensitivity models describe how the predictive uncertainty of an AI component changes for a given set of uncertainties. This is particularly important for analysing the reliability of the overall system in which the AI model is integrated w.r.t. a set of known uncertainties. The validity of the validation results confirms the suitability of addressing **RQ2.2** using sensitivity models.

Based on the sub-research questions **RQ2.1** and **RQ2.2**, we can represent and evaluate the predictive uncertainty of AI components w.r.t. a set of uncertainties. For the reliability prediction itself, we reused the reliability prediction tool PCM-Rel [33] which predicts the success probability of a system modelled with PCM. The extended PCM-Rel by including the sensitivity model of the AI component to account for failure potentials. Because we did not change the core prediction logic of PCM-Rel, but only implemented an upstream resolving routine, we did not need to show the accuracy of our reliability prediction approach (as this would validate the accuracy of PCM-Rel itself, which has already been done in [33]). Instead, we showed the validity of our approach by checking whether a set of plausibility assertions are preserved. In the context of the Udacity case study system, we were able to validate the preservation of all assertions, so we consider **RQ2** to be sufficiently addressed.

Research Question 3 We formulated research question **RQ3** as follows:

> **Research Question 3:** How can adaptation strategies of self-adaptive systems that safeguard uncertain AI black-box components be evaluated in terms of reliability at design-time?

While research question **RQ2** was related to static software systems, **RQ3** focuses on the more general case, i.e. self-adaptive systems. Effectively, the answer of **RQ3** directly follows from the results related to **RQ1** and **RQ2**. In **RQ1**, we developed the *SimExp* method which establishes the basic framework to evaluate adaptation strategies. In **RQ2**, we provided the concepts necessary to predict reliability attributes of AI-enabled systems. Therefore, we have combined the two concepts to provide a framework for evaluating adaptation strategies of self-adaptive systems to safeguard AI black box components. We validated the approach in terms of the Udacity challenge and HRI case study system. For both case studies, we could show that plausibility assertion (9.4) was preserved which we considered as a prerequisite to consider our approach valid.

9.5.3. Threats to Validity

Finally, we complete this section by enumerating possible threats to validity. More specifically, we discuss internal and external validity.

9.5.3.1. Internal Validity

For the validation of our *SimExp* method, we estimate the threats of internal validity to be rather low. This is mainly because we compare *SimExp* with existing simulators, i.e. the DeltaIoT simulator and SimuLizar. In both cases, we produced similar results. Although we observed isolated deviations in the context of the DeltaIoT case study, we were able to demonstrate experimentally that the deviations are a result of the inaccuracy of Prism. Consequently, the inaccuracy of the evaluation tools may potentially pose a further threat to validity. However, all prediction tools that we used are extensively validated and widely accepted in their corresponding communities, i.e. see [15] for SimuLizar, [109] for Prism and [33] for PCM-Rel. Although we have found discrepancies in Prism predictions, we show experimentally that Prism (in conjunction with our *SimExp* method) gives plausible predictions under controlled conditions. Furthermore, since we are no Prism experts but have reused the Prism files from [207] (see [140] with the artefacts), we cannot rule out the possibility that these files contain implementation bugs. Another possible threat to validity refers to the inaccuracy of SimuLizar and the DeltaIoT simulator which we used as a baseline to compare our results. However, both simulators have been used in different contexts and case studies (see for example [15] for SimuLizar and [168, 194, 143] for the DeltaIoT simulator), so we consider them suitable baselines.

For validating our reliability prediction approach, we created an architecture model (PCM model) and applied sensitivity analysis to generate the sensitivity models of the AI components. Both crucially impact the validation result and are thus threats to internal validity. Regarding the architecture model, however, we already explained that as long as the architecture is used in each reliability prediction consistently, the validity of the results and plausibility assertions are not impacted. For the sensitivity analysis, we integrated the insights of Tian et al. [186] which exhaustively investigated the steering angle prediction models of the Udacity self-driving car challenge. More precisely, we considered uncertainties for which Tian et al. verified erroneous behaviour of the steering angle prediction models. Moreover, we determined the deviation of the predicted and actual steering angles based on a metamorphic relation introduced by Tian et al. Finally, at least plausibility assertions (9.2) and (9.3) are independent of the accuracy of the sensitivity model; that is, only the causalities between the success probabilities of the sensitivity models and the associated reliability analysis results are relevant.

9.5.3.2. External Validity

We have instantiated the *SimExp* method in four different domains. Therefore, we can safely exclude the possibility that the method is not generalisable.

Concerning our reliability prediction approach for AI-enabled systems, it could be argued that the approach has only been validated for two case studies with similar settings and a similar experimental setup such that generalisability is not guaranteed. If we recall the Udacity case study, one will see that the mere fact that our approach preserves the causalities between the sensitivity model and the predictions is sufficient to conclude that our approach is generalisable. The same applies to the HRI case study system. Regardless of what type of uncertainties (i.e. whether image brightness or neuron coverage or other factors) are considered in the sensitivity model, as long as the sensitivity analysis correctly captures the corresponding sensitivities, our reliability approach correctly accounts for them in the prediction results. That is, we could have used other uncertainties for which we would have analysed the sensitivity model in a completely different context, but would find the same causalities between the sensitivity model and the prediction results generated by our approach. Therefore, we conclude that our reliability prediction approach is generalisable to other application scenarios.

Part VI.

Epilogue

10. Conclusion

In this chapter, we conclude the thesis. Hereby, we give in section 10.1 a summary of the contributions, their related research questions and how they have been validated. In section 10.2, we recap and discuss the main limitations and assumptions of this thesis. Finally, in section 10.3, we discuss future work.

10.1. Summary

In summary, this thesis presented approaches to evaluate architectural safeguards of AI black-box components regarding reliability attributes. Besides classic architectural approaches such as architectural patterns (e.g. n-version programming pattern), we also considered self-adaptive systems as architectural safeguards. The central goal of this thesis was to evaluate the effect of architectural safeguards and to make informed decisions in the decision-making process, i.e. regarding the selection of an appropriate architectural safeguard w.r.t. quality requirements. We focused on software reliability as system-level property or attribute; however, we integrated our approach into the Palladio framework such that architectural safeguards could be evaluated from the perspective of several quality attributes (e.g. performance and reliability). Therefore, we presented three contributions that addressed our central goal. Additionally, we presented a fourth contribution which complements the aforementioned contributions by a classification structure to evaluate AI-enabled systems in terms of giving assurances for dependability-related system-level properties. In the following, we briefly summarise the approaches, research questions and validation results of each contribution.

Contribution 1: Domain-agnostic instantiation of probabilistic environment models. The first contribution of this thesis partially addressed research question **RQ1** by focusing on the sub-research questions **RQ1.1** and **RQ1.2**. Hereby, the main result is a formal modelling language for describing probabilistic environments, i.e. the *EnvDyn* metamodel. The metamodel serves two purposes: First, it allows the modelling of environmental variables and their effect on the predictive uncertainty of an AI model. Second, the metamodel describes concepts to model the temporal expansion of the environmental variables to describe the operating environment or the *Environmental Dynamics* of a self-adaptive system. Essentially, the semantics of the *EnvDyn* metamodel is based on Bayesian modelling. More specifically, for the first part of the metamodel, we employ BNs (Bayesian networks) to model the environmental variables and their effects on the predictive uncertainty of an AI model by a DAG (directed acyclic graph) and a set of probability distributions. For the temporal expansion, we use DBNs (dynamic Bayesian networks) which extend BNs by an inductive description capturing the temporal evolution of the random variables of the original BN. The modelling capabilities form the building blocks for the second and third contributions because the modelled environments play an essential role in predicting reliability attributes of AI-enabled software systems (second contribution) and later as a generalised variant for self-adaptive software systems (third contribution). Finally, since AI is applicable in various domains, there are also many domain-specific environmental variables which affect the predictive uncertainty of an AI model. Consequently, the *EnvDyn* metamodel must allow the instantiation of environmental variables domain-independently. While arguing from a theoretical perspective that reusing the formal semantics of *Template-based Probabilistic Models* leads to domain independence, we support this claim by instantiating the *EnvDyn* metamodel in four different domains (i.e. the four case study systems).

We validated the applicability of the *EnvDyn* metamodel by considering four case study systems. However, we do not explicitly validate the applicability but rather implicitly by validating the second and third contributions. Note that the second and third contribution highly depends on the *EnvDyn* metamodel in that instances are used to make reliability prediction for AI-enabled systems and to determine the quality of an adaptation strategy w.r.t. several quality objectives. Since we were able to successfully validate both contributions, we implicitly validated the applicability of our *EnvDyn* metamodel. In addition to the pure theoretical argument of achieving domain independence by reusing the formal semantics of template-based probabilistic models, the instantiation of the *EnvDyn* metamodel in four different domains supports that claim.

Contribution 2: Reliability prediction of AI-enabled systems at design-time. The second contribution of this thesis addressed research question **RQ2** and its related sub-questions. The result of the contribution is a reliability prediction approach for AI-enabled systems. The approach is based on the existing reliability prediction approach of Brosch [33]. We abstracted an AI black-box component by a sensitivity model which captures the predictive uncertainty of the AI model and the environmental variables affecting the predictive uncertainty. Hereby, we reuse the modelling capabilities provided by our *EnvDyn* metamodel from the first contribution. We apply an upstream sensitivity analysis to obtain the probability distributions that describe the effect of the environmental variables on the predictive uncertainty of the AI model. The resulting sensitivity model is integrated into our extended reliability prediction approach to systematically consider the failure potentials or the predictive uncertainty of the AI model. We reused the formal modelling language AT (architectural templates) to describe architectural patterns such as the n-version programming pattern. Finally, we created an uncertainty-refined failure-type metamodel which relates architectural templates (described by ATs) with the sensitivity model of an AI component. Furthermore, the metamodel allows modelling the effect of the considered architectural safeguard (e.g. an architectural pattern) on the predictive uncertainty of the AI model (directly or indirectly). Thus, one can predict the effect of an architectural safeguard on the overall reliability of the system by considering its effect on the predictive uncertainty of the AI model.

We validated the approach by considering the Udacity case study system which considers various AI models for steering angle prediction in the context of autonomous driving. The primary goal of the validation was to show the plausibility of our approach. Because we did not change the core logic of the reliability prediction approach of Brosch [33] but rather use an update routine to account for the failure probability of an AI component (w.r.t. the sensitivity model), the accuracy of the prediction results didn't have to be validated. Instead, we validated a set of plausibility assertions which must be preserved by our holistic reliability prediction approach. The plausibility assertions account for real measured properties of the considered AI models. For example, let b and b' be two AI models where b is more accurate than b' w.r.t. some performance measure. If we assume a fixed architecture in which only the AI component is variable, then the predicted reliability of the software architecture including b must be higher compared to the prediction of the software architecture including b'. We could show that our reliability prediction approach preserves all plausibility assertions. Based on the plausibility assertions we could validate that our reliability prediction approach allows software engineers to evaluate architectural safeguards regarding reliability attributes at design-time. Moreover, the validation showed that our approach enables the comparison of architectural safeguards from the perspective of different quality attributes (e.g. reliability and performance). This assists software engineers in decision-making, as they can make informed trade-off decisions (in terms of the quality requirements of the system) at design-time.

Contribution 3: Evaluation of adaptation strategies of self-adaptive systems at design-time. The third contribution of this thesis addresses two research questions and is divided into two parts. The first (and more general) part examines how adaptation strategies of self-adaptive systems can be evaluated at design-time in general; that is, for any purpose (i.e. beyond safeguarding AI components) and in any domain. This relates to research question **RQ1** and tackles the sub-questions **RQ1.3** and **RQ1.4**. To evaluate adaptation strategies, we defined self-adaptive systems as MDPs (Markov decision processes). In MDPs, the main challenge is to find a policy function that returns for each state a suitable action such that the accumulated reward over time is maximised w.r.t. some reward function. We have equated an adaptation strategy with the concept of a policy in MDPs and integrated quality objectives (which must be addressed by the adaptation strategy) into the reward function. Moreover, we mapped the remaining concepts to equivalent concepts in the domain of self-adaptive systems and used model-based techniques to describe them at design-time, i.e. adaptations are abstracted by model transformations, architectural system configurations are described by PCM (Palladio Component Model) and the operating environment is modelled by our *EnvDyn* metamodel. We implemented the concepts in our *SimExp* framework. Internally, *SimExp* applies Monte Carlo prediction to evaluate the adaptation strategy at design-time by sampling environmental states (from the DBN capturing the operating environment modelled by the *EnvDyn* metamodel of the first contribution) and applying model transformation whenever the strategy decides (w.r.t.

the current state) whether an adaptation is applied or not. Each decision of the strategy is evaluated by the reward function where the reward function makes use of quality prediction tools to predict the impact of the selected adaptations on the quality objectives. In the end, the expected accumulated reward is estimated and relates each strategy with a value and serves as a foundation to compare strategies or assess design decisions within a strategy.

We validated the appropriateness of this part of the contribution by considering two case study systems, namely a load balancer and the DeltaIoT case study system. Both are equipped with a domain-specific simulator that allows the evaluation of adaptation strategies for the given case study. We applied our *SimExp* method in both domains, i.e. we created the corresponding models, model transformations, etc. For each case study, we considered a fixed set of adaptation strategies that were once evaluated by *SimExp* and once evaluated by the respective simulator of each case study. We compared the resulting ranks of the evaluated strategies. The results were in line with our expectations: The generated ranks of the adaptation strategies were equal to the ranks produced by the simulators. Thus, we could not only validate the appropriateness of *SimExp* but also the general possibility to evaluate and compare design decisions within an adaptation strategy and distinct strategies.

The second part of this contribution combines our reliability prediction approach from the second contribution with the *SimExp* framework to enable the evaluation of self-adaptive systems for safeguarding AI black-box components. Hereby, we integrated the reliability prediction approach into the reward function. Moreover, we discussed how to deal with large input spaces (e.g. the pixel space) during the evaluation process by focusing on more manageable spaces, i.e. the *Monitorable Space.*

For the second part of this contribution, we validated again the plausibility. Just as in the second contribution, we formulated plausibility assertions which must be maintained by the approach. For the validation, we considered two case study systems, namely the HRI and Udacity case study system. The results have shown that all plausibility assertions were maintained by our approach.

Overall, we can conclude that our *SimExp* framework allows the evaluation of adaptation strategies at design-time which greatly supports software engineers in comparing distinct strategies or design decisions within an adaptation strategy family. Especially when evaluating adaptation strategies of self-adaptive systems that are acting as architectural safeguards, software engineers cannot only evaluate the quality of a strategy from the perspective of reliability but also other quality attributes (e.g. performance). Moreover, we demonstrated how the *SimExp* framework can be used to evaluate whether an adaptive or non-adaptive solution of an architectural safeguard should be considered. Overall, the *SimExp* framework provides software engineers with a repertoire of analysis scenarios to explore multiple design options (in terms of system quality requirements) during the system design process.

Contribution 4: Classification structure to assess AI-enabled systems regarding assurances that can be given for system-level dependability properties. In the last contribution of this thesis, we elaborated a classification structure to evaluate AI-enabled systems in terms of assurances that can be given for dependability-related system-level properties. The contribution relates to research question **RQ4** and its corresponding sub-questions. We defined four classes of architectural dependability assurance into which an AI-enabled system can be classified. Each class describes the degree (e.g. fully or partially) and point in the development process (i.e. design-time, runtime or not at all) at which assurances can be given for a particular system-level property Φ_{Sys}. Based on the classes, we elaborated a classification structure consisting of various classification dimensions that classify AI systems. We identified four dimensions, namely *Abstractability* (i.e. the extent to which the system can be abstracted by models to analyse the system in terms of Φ_{Sys}), *Approximation of the System Dynamics* (e.g. the accurate description of the stochastic process that describes the dynamics the system), *Analytic Capacity* (i.e. the analytic potential of an AI component itself) and *Fail-Safe* (i.e. the ability to transition the system into a fail-safe mode).

Because the classification structure is highly subjective, a comprehensive evaluation was not possible in this thesis. The classification of multiple AI-enabled systems according to our classes and classification structure would require the involvement of domain experts due to the subjective nature of the classification structure. However, this was not realistically possible in the scope of this thesis. Instead, we applied our classification structure to AI systems from three representative domains in which AI have been commonly used, namely AI-supported assistance in automated driving, human-robot-interaction systems and aircraft collision avoidance systems. However, since we are no experts in none of the domains, we could only classify the systems according to the information provided by the scientific publications. Consequently, we could only show the applicability of our classification dimensions. A comprehensive evaluation of the classification structure in terms of other aspects (such as completeness or coverage) is planned in future work.

10.2. Central Limitations and Assumptions

Although we have already discussed in each contribution-related chapter the respective limitations and assumptions, we summarise in this section the central assumptions. Moreover, we discuss the main limitations complemented by insights we gained during validation.

Limitations: Recall that we use PCM (Palladio Component Model) as ADL (architectural description language) to describe software architectures employing models. Although we described the concepts of *SimExp* by using PCM, the *SimExp* method itself is ADL agnostic. However, this applies not to our reliability prediction approach for AI-enabled systems. More specifically, the approach builds upon the reliability prediction approach PCM-Rel of Brosch [33] which requires PCM as modelling language for describing software architectures. Consequently, our extended reliability prediction approach can only be used in combination with PCM models. The same applies to the reliability prediction logic. Recall that we did not change the prediction semantics of PCM-Rel but implemented a procedure which iteratively (w.r.t. considered uncertainties or environmental variables $\varphi_1, \dots, \varphi_n$) calculates the failure probability of an AI component, updates the failure probability in the PCM model and invokes PCM-Rel. As a result, we inherit the prediction semantics of PCM-Rel as well. For example, while the effect of a wrong prediction might rather affect other components of the software architecture which are dependent on the prediction result, PCM-Rel evaluates the failure probability directly at the AI component in the PCM model. Nonetheless, regarding the evaluation of architectural safeguards at design-time, we argue that the reliability prediction semantics of PCM-Rel are sufficient. Additionally, since PCM-Rel annotates PCM models with failure types, it is possible to specify the exact location in the software architecture (i.e. the modelled software component) that is affected by incorrect predictions of the AI component.

We applied our reliability prediction approach for AI-enabled systems in the context of autonomous driving and human-robot-interaction by considering distinct and complex types of DNNs (deep neural networks), e.g. CNNs (convolutional neural networks) or RNNs (recurrent neural networks). However, there might be more complex settings (especially in the context of autonomous driving) where AI models rely on input data from other AI models. For example, the perception phase of a cognitive system might involve upstream AI components for sensor fusion of the raw sensor data. In this case, however, the input of the AI component must rely on outputs of other AI models that are associated with AI-specific uncertainties as well. One can approach this problem by considering the involved AI components as an end-to-end approach which produces for some raw sensor data a corresponding output prediction. In this case, our reliability approach can be applied as usual in which the predictive uncertainty of the end-to-end AI component is estimated by considering various uncertainties or environmental variables. It is arguably difficult to find an appropriate set of uncertainties or environmental variables of the sensitivity model that allow reasoning about the predictive uncertainty. Although the sensitivity model doesn't have to perfectly approximate predictive uncertainty to evaluate design decisions of architectural safeguards, a poor sensitivity model would still corrupt the reliability predictions of the entire system significantly. However, as a starting point, we focused on single AI components. Nonetheless, checking the applicability of our approach for more complex AI systems can be still considered to be the subject of future work.

Finally, our *SimExp* does currently not support *Transient Effects* in the evaluation process of self-adaptive systems. Recall that transient effects relate to execution times and consumed resources of applied adaptations [178], i.e. they can be considered as costs associated with an adaptation. In the domains or case study systems we considered in this thesis (except the load balancing case study system), transient effects are rather negligible because none of the adaptations is associated with high costs. Nevertheless, since transient effects are important in some domains (e.g. performance engineering), it makes sense to include them in the reward function. As a starting point, however, we solely focused on system-level quality attributes but plan to account for transient effects in future work.

Assumptions: Since our *SimExp* method strongly builds upon MDPs (Markov decision processes), we also inherit assumptions made in MDPs. This refers to the Markov assumption, which states that the probability to transition from one state to another solely depends on the previously given state and not on the history of already past states (see equation (2.4)). The Markov assumption is essential in the *SimExp* framework when trajectories (i.e. sequences of states) are sampled w.r.t. a predefined adaptation strategy and the modelled environmental dynamics. However, as we already pointed out in chapter 4, many scientific works in the self-adaptive system community make use of MDPs to describe the stochastic dynamics of self-adaptive systems. Moreover, whenever the Markov assumption is too strong in a given context, it can always be sharpened by selecting a richer state representation (i.e. considering more variables in the state description) [105].

The second central assumption of this thesis refers to the assumption of discretising the environment or environmental dynamics into a set of discrete states. Basically, we made this assumption to reduce the state space and to tackle the state space explosion problem. However, the discretisation is at the cost of information loss. For model-based analysis, it is fairly common to use abstraction and simplification to analyse complex systems at design-time. In addition, the degree of discretisation can be controlled such that also more fine-grained states can be considered (at the cost of an increased state space).

Finally, in section 7.2, we made the MAR (missing at random) assumption (i) to justify that we can focus on environmental variables or properties instead of considering individual input-output pairs of an AI component and (ii) to deal with the hidden state problem. This enabled us to ignore potentially large and complex structured input spaces (such as the pixel space) during the evaluation of self-adaptive systems (or rather their adaptation strategies) that are specifically designed to safeguard an AI component. However, the MAR assumption must be considered with care because the assumption might not hold in some settings. Just like the Markov assumption, the MAR assumption can be sharpened [105] by identifying “good” properties that allow conclusions to be drawn about the true state of an AI component.

10.3. Future Work

In this section, we summarise several aspects of the thesis that we identified for extension in future work.

As we already discussed in section 10.2, one of the limitations of our approach is that currently no transient effects (i.e. costs associated with adaptations) are considered during the evaluation of adaptation strategies. Therefore, in future work, we plan to integrate transient effects in our *SimExp* framework. An entry point is provided by Stier [178] which considers transient effects in the analysis of self-adaptive systems in the context of the Palladio framework. However, the approach focuses on transient effects that relate to performance and energy efficiency. Nonetheless, the concepts can be used as a starting point and generalised to other domains.

In section 6, we presented our *SimExp* framework for evaluating adaptation strategies of self-adaptive systems. Currently, some required artefacts are either passed to the framework via code and not by dedicated models (the adaptation strategy and reward function implementations) while others are lacking graphical editors which would greatly reduce the modelling effort (the *EnvDyn* metamodel). Unfortunately, we could not address all in the scope of this thesis. For the sake of usability and to make the framework more amenable to the community, we plan to add more modelling features to the framework in future work. For example, we already envisioned the design of a DSL (domain-specific language) for modelling adaptation strategies [147]. In future work, we plan to implement such a DSL and integrate it into the *SimExp* framework. Moreover, another potential aspect for future work is the modelling of reward functions based on quality attributes. Currently, the reward function must be specified by a java code file which extends a particular interface provided by the *SimExp* framework. Hereby, the work of Becker [15, P.87] might serve as a starting point because it presents a metamodel for describing service level objectives (e.g. tolerance ranges, violation ranges, etc.). The concepts and ideas might be potentially reused to represent reward functions. Finally, regarding the modelling of environment models (i.e. instances of the *EnvDyn* metamodel), we plan to develop graphical editors that simplify the creation of environment models. The work of Koller and Friedman [105, P.157] provides methods for the compact description of such models, which can be regarded as a starting point.

Although we validated our approach in case study systems that include complex AI models (e.g. Udacity case study where distinct DNNs are used to predict steering angles based on image data), we plan to apply our approach to more advanced case study systems that encompass AI systems with several AI components. For example, in cognitive systems (e.g. self-driving cars) the sensor data is preprocessed in the perception phase before it is forwarded to the AI component used in other phases. However, also within the perception phase, AI models might be used for sensor fusion or representation learning (i.e. unsupervised learning). That is, the predictive uncertainty of an AI component (e.g.) used for classification or detection tasks is dependent on the result of other AI components. In this case, we must not only account for the environmental variables affecting predictive uncertainty but also model a kind of failure propagation of the individual AI components. Intuitively, we can model such a setting with our approach by treating the random variables capturing the predictive uncertainty of preceding AI components in the sensitivity model of dependent AI components. However, this is hypothetical and requires further investigation.

Also in terms of the *SimExp* framework, we plan to apply the method to further use cases of different domains. For example, Gerasimou [65] et al. provide a case study system of an unmanned underwater vehicle which is equipped with a domain-specific simulator one can use to evaluate adaptation strategies. By applying the *SimExp* methods to further case studies, we hope to identify possible vulnerabilities of the method or potential aspects that can be further automated.

Finally, regarding our classes of architectural dependability assurance and the classification structure we elaborated to classify AI systems into one of the classes, we plan a more comprehensive evaluation. Currently, we evaluated the applicability of the structure by applying it to a collection of well-known and representative AI systems. However, the classification dimensions of the structure are highly subjective such that an objective assessment is practically difficult to achieve. Moreover, we are no experts in any of the discussed domains; thus, the discussion of the classification structure is subjective. A more founded discussion would require the inclusion of domain experts. This is, however, associated with a high effort which was not possible to be conducted in the scope of this thesis but is still subject to future work. Moreover, we plan to apply the structure in systematic literature research to assess the coverage of the classification structure (i.e. whether any AI system can be classified). As a result, we hope to refine our existing dimensions (if necessary) or even identify additional dimensions (if any).

Bibliography

[1] Vahdat Abdelzad et al. "Detecting out-of-distribution inputs in deep neural networks using an early-layer output". In: *arXiv preprint arXiv:1910.10307* (2019).

[2] Waleed Abdulla. *Mask R-CNN for object detection and instance segmentation on Keras and TensorFlow*. `https://github.com/matterport/Mask_RCNN`. 2017.

[3] Michael Aeberhard et al. "Automated Driving with ROS at BMW". In: *ROSCon 2015 Hamburg*. 2015.

[4] Mohammed Alshiekh et al. "Safe reinforcement learning via shielding". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 32. 1. 2018.

[5] Saleema Amershi et al. "Software engineering for machine learning: A case study". In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE. 2019, pp. 291–300.

[6] Mehdi Amoui et al. "Adaptive action selection in autonomic software using reinforcement learning". In: *Fourth International Conference on Autonomic and Autonomous Systems (ICAS'08)*. IEEE. 2008, pp. 175–181.

[7] Adina Aniculaesei et al. "Toward a holistic software systems engineering approach for dependable autonomous systems". In: *2018 IEEE/ACM 1st International Workshop on Software Engineering for AI in Autonomous Systems (SEFAIAS)*. IEEE. 2018, pp. 23–30.

[8] Paolo Arcaini, Elvinia Riccobene, and Patrizia Scandurra. "Modeling and analyzing MAPE-K feedback loops for self-adaptation". In: *2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE. 2015, pp. 13–23.

[9] Paolo Arcaini et al. "Model-Based Testing for MAPE-K adaptation control loops". In: *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE. 2020, pp. 43–51.

[10] Ashraf Armoush. "Design patterns for safety-critical embedded systems". Aachen, Techn. Hochsch., Diss., 2010. PhD thesis. Aachen, 2010, XIV, 181 S. : graph. Darst. URL: `https://publications.rwth-aachen.de/record/51773`.

[11] Rob Ashmore, Radu Calinescu, and Colin Paterson. "Assuring the machine learning lifecycle: Desiderata, methods, and challenges". In: *ACM Computing Surveys (CSUR)* 54.5 (2021), pp. 1–39.

[12] Earl T Barr et al. "The oracle problem in software testing: A survey". In: *IEEE transactions on software engineering* 41.5 (2014), pp. 507–525.

[13] Osbert Bastani et al. “Measuring neural net robustness with constraints”. In: *Advances in neural information processing systems* 29 (2016).

[14] Dennis Bäuml. “Entwicklung zuverlässiger KI-basierter Software-Systeme in Anwesenheit von Unsicherheit”. Master’s Thesis. Karlsruhe: Karlsruhe Institute of Technology (KIT), 2022.

[15] Matthias Becker. “Engineering self-adaptive systems with simulation-based performance prediction”. PhD thesis. University of Paderborn, Germany, 2017. URL: `http://nbn-resolving.de/urn:nbn:de:hbz:466:2-28816`.

[16] Matthias Becker, Steffen Becker, and Joachim Meyer. “SimuLizar: Design-Time Modeling and Performance Analysis of Self-Adaptive Systems.” In: *Software Engineering* 213 (2013), pp. 71–84.

[17] Matthias Becker, Markus Luckey, and Steffen Becker. “Performance analysis of self-adaptive systems for requirements validation at design-time”. In: *Proceedings of the 9th international ACM Sigsoft conference on Quality of software architectures*. ACM. 2013, pp. 43–52.

[18] Sagar Behere and Martin Torngren. “A functional architecture for autonomous driving”. In: *Automotive Software Architecture (WASA), 2015 First International Workshop on*. IEEE. 2015, pp. 3–10.

[19] Richard Bellman. “Dynamic programming”. In: *Science* 153.3731 (1966), pp. 34–37.

[20] Luca Berardinelli et al. “Multidimensional context modeling applied to non-functional analysis of software”. In: *Software & Systems Modeling* (2017), pp. 1–40.

[21] Simona Bernardi and José Merseguer. “A UML profile for dependability analysis of real-time embedded systems”. In: *Proceedings of the 6th international workshop on Software and performance*. 2007, pp. 115–124.

[22] Anil Bhattacharyya. “On a measure of divergence between two statistical populations defined by their probability distributions”. In: *Bull. Calcutta Math. Soc.* 35 (1943), pp. 99–109.

[23] Alessandro Biondi et al. “A safe, secure, and predictable software architecture for deep learning in safety-critical systems”. In: *IEEE Embedded Systems Letters* 12.3 (2019), pp. 78–82.

[24] Christopher M Bishop and Nasser M Nasrabadi. *Pattern recognition and machine learning*. Vol. 4. 4. Springer, 2006.

[25] Peter Bishop, Robin Bloomfield, and Sofia Guerra. “The future of goal-based assurance cases”. In: *Proc. Workshop on Assurance Cases*. Citeseer. 2004, pp. 390–395.

[26] David M Blei, Alp Kucukelbir, and Jon D McAuliffe. “Variational inference: A review for statisticians”. In: *Journal of the American statistical Association* 112.518 (2017), pp. 859–877.

[27] RE Bloomfield et al. “Ascad—adelard safety case development manual”. In: *Adelard* 5 (1998).

[28] Rainer Böhme and Ralf Reussner. “Validation of predictions with measurements”. In: *Dependability Metrics*. Springer, 2008, pp. 14–18.

[29] Mariusz Bojarski et al. “End to end learning for self-driving cars”. In: *arXiv preprint arXiv:1604.07316* (2016).

[30] Zoran Bosnić and Igor Kononenko. “An overview of advances in reliability estimation of individual predictions in machine learning”. In: *Intelligent Data Analysis* 13.2 (2009), pp. 385–401.

[31] Gunnar Brataas, Erlend Stav, and Sebastian Lehrig. “Analysing evolution of work and load”. In: *2016 12th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA)*. IEEE. 2016, pp. 90–95.

[32] Eric Breck et al. “The ML test score: A rubric for ML production readiness and technical debt reduction”. In: *2017 IEEE International Conference on Big Data (Big Data)*. IEEE. 2017, pp. 1123–1132.

[33] Franz Brosch. *Integrated software architecture-based reliability prediction for it systems*. Vol. 9. KIT Scientific Publishing, 2012.

[34] Franz Brosch et al. “Architecture-based reliability prediction with the palladio component model”. In: *IEEE Transactions on Software Engineering* 38.6 (2011), pp. 1319–1339.

[35] Simon Burton, Lydia Gauerhof, and Christian Heinzemann. “Making the case for safety of machine learning in highly automated driving”. In: *International Conference on Computer Safety, Reliability, and Security*. Springer. 2017, pp. 5–16.

[36] Victor R Basili1 Gianluigi Caldiera and H Dieter Rombach. “The goal question metric approach”. In: *Encyclopedia of software engineering* (1994), pp. 528–532.

[37] Javier Cámara and Rogério De Lemos. “Evaluation of resilience in self-adaptive systems using probabilistic model-checking”. In: *2012 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE. 2012, pp. 53–62.

[38] Javier Cámara et al. “Adaptation impact and environment models for architecture-based self-adaptive systems”. In: *Science of Computer Programming* 127 (2016), pp. 50–75.

[39] Javier Cámara et al. “MOSAICO: offline synthesis of adaptation strategy repertoires with flexible trade-offs”. In: *Automated Software Engineering* 25.3 (2018), pp. 595–626.

[40] Matteo Camilli, Raffaela Mirandola, and Patrizia Scandurra. “Runtime Equilibrium Verification for Resilient Cyber-Physical Systems”. In: *2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. IEEE. 2021, pp. 71–80.

[41] Maria Casimiro et al. “Self-Adaptation for Machine Learning Based Systems.” In: *ECSA (Companion)*. 2021.

[42] *Chauffeur steering angle prediction model.* 2016. URL: `https://github.com/udacity/self-driving-car/tree/master/steering-models/community-models/chauffeur`.

[43] Chenyi Chen et al. "Deepdriving: Learning affordance for direct perception in autonomous driving". In: *Proceedings of the IEEE International Conference on Computer Vision.* 2015, pp. 2722–2730.

[44] Liming Chen and Algirdas Avizienis. "N-version programming: A fault-tolerance approach to reliability of software operation". In: *Proc. 8th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-8).* Vol. 1. 1978, pp. 3–9.

[45] Chih-Hong Cheng, Dhiraj Gulati, and Rongjie Yan. "Architecting dependable learning-enabled autonomous systems: A survey". In: *arXiv preprint arXiv:1902.10590* (2019).

[46] Chih-Hong Cheng, Georg Nührenberg, and Hirotoshi Yasuoka. "Runtime monitoring neuron activation patterns". In: *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE).* IEEE. 2019, pp. 300–303.

[47] Shang-Wen Cheng and David Garlan. "Stitch: A language for architecture-based self-adaptation". In: *Journal of Systems and Software* 85.12 (2012), pp. 2860–2875.

[48] Shang-Wen Cheng, David Garlan, and Bradley Schmerl. "Evaluating the effectiveness of the rainbow self-adaptive system". In: *2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems.* IEEE. 2009, pp. 132–141.

[49] Tony Clark, Cesar Gonzalez-Perez, and Brian Henderson-Sellers. "A foundation for multi-level modelling". In: *MULTI 2014–Multi-Level Modelling Workshop Proceedings.* 2014.

[50] Jeremy Cohen, Elan Rosenfeld, and Zico Kolter. "Certified adversarial robustness via randomized smoothing". In: *International Conference on Machine Learning.* PMLR. 2019, pp. 1310–1320.

[51] *comma.ai's steering model.* 2016. URL: `https://github.com/commaai/research/blob/master/train_steering_model.py`.

[52] Kai Ding, Andrey Morozov, and Klaus Janschek. "Classification of hierarchical fault-tolerant design patterns". In: *2017 IEEE 15th Intl Conf on Dependable, Autonomic and Secure Computing, 15th Intl Conf on Pervasive Intelligence and Computing, 3rd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech).* IEEE. 2017, pp. 612–619.

[53] Alexandre Donzé. "Breach, a toolbox for verification and parameter synthesis of hybrid systems". In: *International Conference on Computer Aided Verification.* Springer. 2010, pp. 167–170.

[54] Tommaso Dreossi, Alexandre Donzé, and Sanjit A Seshia. "Compositional falsification of cyber-physical systems with machine learning components". In: *Journal of Automated Reasoning* 63.4 (2019), pp. 1031–1053.

[55] Yehia Elrakaiby, Paola Spoletini, and Bashar Nuseibeh. “Optimal by Design: Model-Driven Synthesis of Adaptation Strategies for Autonomous Systems”. In: *arXiv preprint arXiv:2001.08525* (2020).

[56] Wolfgang Ertel. *Introduction to artificial intelligence*. Springer, 2018.

[57] Naeem Esfahani and Sam Malek. “Uncertainty in self-adaptive software systems”. In: *Software Engineering for Self-Adaptive Systems II*. Springer, 2013, pp. 214–238.

[58] José M Faria. “Machine learning safety: An overview”. In: *Proceedings of the 26th Safety-Critical Systems Symposium, York, UK*. 2018, pp. 6–8.

[59] Robert Feldt, Francisco Gomes de Oliveira Neto, and Richard Torkar. “Ways of applying artificial intelligence in software engineering”. In: *2018 IEEE/ACM 6th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE)*. IEEE. 2018, pp. 35–41.

[60] Antonio Filieri et al. “Self-adaptive software meets control theory: A preliminary approach supporting reliability requirements”. In: *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE. 2011, pp. 283–292.

[61] Joao M Franco et al. “Improving self-adaptation planning through software architecture-based stochastic modeling”. In: *Journal of Systems and software* 115 (2016), pp. 42–60.

[62] Lidia Fuentes-Fernández and Antonio Vallecillo-Moreno. “An introduction to UML profiles”. In: *UML and Model Engineering* 2.6-13 (2004), p. 72.

[63] Javier Garcıa and Fernando Fernández. “A comprehensive survey on safe reinforcement learning”. In: *Journal of Machine Learning Research* 16.1 (2015), pp. 1437–1480.

[64] David Garlan, Bradley Schmerl, and Shang-Wen Cheng. “Software architecture-based self-adaptation”. In: *Autonomic computing and networking*. Springer, 2009, pp. 31–55.

[65] Simos Gerasimou et al. “UNDERSEA: an exemplar for engineering self-adaptive unmanned underwater vehicles”. In: *2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE. 2017, pp. 83–89.

[66] Carlo Ghezzi et al. “Managing non-functional uncertainty via model-driven adaptivity”. In: *Software Engineering (ICSE), 2013 35th International Conference on*. IEEE. 2013, pp. 33–42.

[67] Görkem Giray. “A software engineering perspective on engineering machine learning systems: State of the art and challenges”. In: *Journal of Systems and Software* 180 (2021), p. 111031.

[68] *Goal Structuring Notation (GSN) community standard version 3*. 2011. URL: `https://scsc.uk/publications`.

[69] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. “Deep learning (adaptive computation and machine learning series)”. In: *Adaptive Computation and Machine Learning series* (2016), p. 800.

[70] Ian Goodfellow et al. “Generative adversarial nets”. In: *Advances in neural information processing systems* 27 (2014).

[71] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. “Explaining and harnessing adversarial examples”. In: *arXiv preprint arXiv:1412.6572* (2014).

[72] Vincenzo Grassi, Raffaela Mirandola, and Antonino Sabetta. “A model-driven approach to performability analysis of dynamically reconfigurable component-based systems”. In: *Proceedings of the 6th international workshop on Software and performance.* 2007, pp. 103–114.

[73] Vincenzo Grassi, Raffaela Mirandola, and Antonino Sabetta. “From design to analysis models: a kernel language for performance and reliability analysis of component-based systems”. In: *Proceedings of the 5th international workshop on Software and performance.* 2005, pp. 25–36.

[74] Shixiang Gu et al. “Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates”. In: *2017 IEEE international conference on robotics and automation (ICRA).* IEEE. 2017, pp. 3389–3396.

[75] Xiaozhe Gu and Arvind Easwaran. “Towards safe machine learning for CPS: infer uncertainty from training data”. In: *Proceedings of the 10th ACM/IEEE International Conference on Cyber-Physical Systems.* ACM. 2019, pp. 249–258.

[76] Riccardo Guidotti et al. “A survey of methods for explaining black box models”. In: *ACM computing surveys (CSUR)* 51.5 (2018), p. 93.

[77] Arpan Gujarati, Sathish Gopalakrishnan, and Karthik Pattabiraman. “New wine in an old bottle: N-version programming for machine learning components”. In: *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW).* IEEE. 2020, pp. 283–286.

[78] S. de Gyves Avila and K. Djemame. “Fuzzy Logic Based QoS Optimization Mechanism for Service Composition”. In: *2013 IEEE Seventh International Symposium on Service-Oriented System Engineering.* IEEE, Mar. 2013. DOI: `10.1109/sose.2013.28`.

[79] Muhammad Abdullah Hanif et al. “Robust machine learning systems: Reliability and security for deep neural networks”. In: *2018 IEEE 24th International Symposium on On-Line Testing And Robust System Design (IOLTS).* IEEE. 2018, pp. 257–260.

[80] Kaiming He et al. “Mask r-cnn”. In: *Proceedings of the IEEE international conference on computer vision.* 2017, pp. 2961–2969.

[81] Dan Hendrycks and Kevin Gimpel. “A baseline for detecting misclassified and out-of-distribution examples in neural networks”. In: *arXiv preprint arXiv:1610.02136* (2016).

[82] Andreas Henelius et al. “A peek into the black box: exploring classifiers by randomization”. In: *Data mining and knowledge discovery* 28.5 (2014), pp. 1503–1529.

[83] Marc Hesenius et al. “Towards a software engineering process for developing data-driven applications”. In: *2019 IEEE/ACM 7th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE).* IEEE. 2019, pp. 35–41.

[84] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[85] Boyue Caroline Hu et al. "If a Human Can See It, So Should Your System: Reliability Requirements for Machine Vision Components". In: *arXiv preprint arXiv:2202.03930* (2022).

[86] Xiaowei Huang et al. "A survey of safety and trustworthiness of deep neural networks: Verification, testing, adversarial attack and defence, and interpretability". In: *Computer Science Review* 37 (2020), p. 100270.

[87] Nikolaus Huber et al. "Model-based self-aware performance and resource management using the descartes modeling language". In: *IEEE Transactions on Software Engineering* 43.5 (2016), pp. 432–452.

[88] Nikolaus Huber et al. "Modeling run-time adaptation at the system architecture level in dynamic service-oriented environments". In: *Service Oriented Computing and Applications* 8.1 (2014), pp. 73–89.

[89] Eyke Hüllermeier and Willem Waegeman. "Aleatoric and epistemic uncertainty in machine learning: An introduction to concepts and methods". In: *Machine Learning* 110.3 (2021), pp. 457–506.

[90] "IEEE Recommended Practice on Software Reliability". In: *IEEE Std 1633-2016 (Revision of IEEE Std 1633-2008)* (2017), pp. 1–261. DOI: 10.1109/IEEESTD.2017.7827907.

[91] M Usman Iftikhar and Danny Weyns. "Activforms: Active formal models for self-adaptation". In: *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems.* 2014, pp. 125–134.

[92] Muhammad Usman Iftikhar et al. "Deltaiot: A self-adaptive internet of things exemplar". In: *2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS).* IEEE. 2017, pp. 76–82.

[93] Sae International. "Taxonomy and definitions for terms related to driving automation systems for on-road motor vehicles". In: *SAE* (2018).

[94] Kichun Jo et al. "Development of autonomous car—Part II: A case study on the implementation of an autonomous driving system based on distributed architecture". In: *IEEE Transactions on Industrial Electronics* 62.8 (2015), pp. 5119–5132.

[95] Kyle D Julian and Mykel J Kochenderfer. "Guaranteeing safety for neural network-based aircraft collision avoidance systems". In: *2019 IEEE/AIAA 38th Digital Avionics Systems Conference (DASC).* IEEE. 2019, pp. 1–10.

[96] Kyle D Julian, Mykel J Kochenderfer, and Michael P Owen. "Deep neural network compression for aircraft collision avoidance systems". In: *Journal of Guidance, Control, and Dynamics* 42.3 (2019), pp. 598–608.

[97] Kyle D Julian et al. "Policy compression for aircraft collision avoidance systems". In: *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC).* IEEE. 2016, pp. 1–10.

[98] Daniel Kang et al. “Model assertions for monitoring and improving ML models”. In: *Proceedings of Machine Learning and Systems* 2 (2020), pp. 481–496.

[99] Guy Katz et al. “Reluplex: An efficient SMT solver for verifying deep neural networks”. In: *International conference on computer aided verification*. Springer. 2017, pp. 97–117.

[100] Guy Katz et al. “The marabou framework for verification and analysis of deep neural networks”. In: *International Conference on Computer Aided Verification*. Springer. 2019, pp. 443–452.

[101] Jeffrey O Kephart and David M Chess. “The vision of autonomic computing”. In: *Computer* 36.1 (2003), pp. 41–50.

[102] Edward Kim et al. “A programmatic and semantic approach to explaining and debugging neural network based object detectors”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2020, pp. 11128–11137.

[103] Anneke G Kleppe et al. *MDA explained: the model driven architecture: practice and promise*. Addison-Wesley Professional, 2003.

[104] Fabian Kneer, Erik Kamsties, and Klaus Schmid. “Environment modeling for adaptive systems: a systematic literature review”. In: *arXiv preprint arXiv:2011.07892* (2020).

[105] Daphne Koller and Nir Friedman. *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

[106] Anne Koziolek. *Automated improvement of software architecture models for performance and other quality attributes*. Vol. 7. KIT Scientific Publishing, 2014.

[107] Sanjay Krishnan and Eugene Wu. “Palm: Machine learning explanations for iterative debugging”. In: *Proceedings of the 2Nd workshop on human-in-the-loop data analytics*. 2017, pp. 1–6.

[108] Philippe Kruchten, Patricia Lago, and Hans van Vliet. “Building up and reasoning about architectural knowledge”. In: *International conference on the quality of software architectures*. Springer. 2006, pp. 43–58.

[109] Marta Kwiatkowska, Gethin Norman, and David Parker. “Prism: Probabilistic model checking for performance and reliability analysis”. In: *ACM SIGMETRICS Performance Evaluation Review* 36.4 (2009), pp. 40–45.

[110] Philip Langer et al. “EMF Profiles: A Lightweight Extension Approach for EMF Models.” In: *J. Object Technol.* 11.1 (2012), pp. 1–29.

[111] Michael Austin Langford and Betty HC Cheng. ““Know What You Know”: Predicting Behavior for Learning-Enabled Systems When Facing Uncertainty”. In: *2021 International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE. 2021, pp. 78–89.

[112] Yann LeCun et al. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[113] Sebastian Michael Lehrig. *Efficiently conducting quality-of-service analyses by templating architectural knowledge.* Vol. 25. KIT Scientific Publishing, 2018.

[114] Rogério de Lemos and Marek Grześ. "Self-adaptive artificial intelligence". In: *2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS).* IEEE. 2019, pp. 155–156.

[115] Grace A Lewis, Ipek Ozkaya, and Xiwei Xu. "Software Architecture Challenges for ML Systems". In: *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME).* IEEE. 2021, pp. 634–638.

[116] Shih-Chieh Lin et al. "The architectural implications of autonomous driving: Constraints and acceleration". In: *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems.* 2018, pp. 751–766.

[117] Renting Liu, Zhaorong Li, and Jiaya Jia. "Image partial blur detection and classification". In: *2008 IEEE conference on computer vision and pattern recognition.* IEEE. 2008, pp. 1–8.

[118] Yaping Luo et al. "An architecture pattern for safety critical automated driving applications: Design and analysis". In: *2017 Annual IEEE International Systems Conference (SysCon).* IEEE. 2017, pp. 1–7.

[119] Fumio Machida. "On the diversity of machine learning models for system reliability". In: *2019 IEEE 24th Pacific Rim International Symposium on Dependable Computing (PRDC).* IEEE. 2019, pp. 276–27609.

[120] Anne Martens et al. "Automatically improve software architecture models for performance, reliability, and cost using evolutionary algorithms". In: *Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering.* 2010, pp. 105–116.

[121] Robert von Massow, André van Hoorn, and Wilhelm Hasselbring. "Performance simulation of runtime reconfigurable component-based software architectures". In: *European Conference on Software Architecture.* Springer. 2011, pp. 43–58.

[122] Nenad Medvidovic and Richard N Taylor. "Software architecture: foundations, theory, and practice". In: *2010 ACM/IEEE 32nd International Conference on Software Engineering.* Vol. 2. IEEE. 2010, pp. 471–472.

[123] Sean P Meyn and Richard L Tweedie. *Markov chains and stochastic stability.* Springer Science & Business Media, 2012.

[124] Tom M Mitchell. *Machine Learning.* McGraw-Hill, 1997.

[125] Volodymyr Mnih et al. "Playing atari with deep reinforcement learning". In: *arXiv preprint arXiv:1312.5602* (2013).

[126] Gabriel A Moreno et al. "Flexible and efficient decision-making for proactive latency-aware self-adaptation". In: *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 13.1 (2018), pp. 1–36.

[127] Gabriel A Moreno et al. “Proactive self-adaptation under uncertainty: a probabilistic model checking approach”. In: *Proceedings of the 2015 10th joint meeting on foundations of software engineering*. 2015, pp. 1–12.

[128] Henry Muccini and Karthik Vaidhyanathan. “Software architecture for ml-based systems: what exists and what lies ahead”. In: *2021 IEEE/ACM 1st Workshop on AI Engineering-Software Engineering for AI (WAIN)*. IEEE. 2021, pp. 121–128.

[129] Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

[130] Kevin Patrick Murphy. “Dynamic bayesian networks: representation, inference and learning”. In: (2002).

[131] Patrick Musau et al. “On Using Real-Time Reachability for the Safety Assurance of Machine Learning Controllers”. In: *2022 IEEE International Conference on Assured Autonomy (ICAA)*. IEEE. 2022, pp. 1–10.

[132] Niranjan D Narvekar and Lina J Karam. “A no-reference image blur metric based on the cumulative probability of blur detection (CPBD)”. In: *IEEE Transactions on Image Processing* 20.9 (2011), pp. 2678–2683.

[133] MDA OMG. *OMG Unified Modeling Language(OMG UML), Infrastructure, V2.1.2, 2007*. URL: `http://www.omg.org/spec/UML/2.1.2/`.

[134] Peyman Oreizy et al. “An architecture-based approach to self-adaptive software”. In: *IEEE Intelligent Systems and Their Applications* 14.3 (1999), pp. 54–62.

[135] Nicolas Papernot et al. “Practical black-box attacks against machine learning”. In: *Proceedings of the 2017 ACM on Asia conference on computer and communications security*. ACM. 2017, pp. 506–519.

[136] Kexin Pei et al. “Deepxplore: Automated whitebox testing of deep learning systems”. In: *proceedings of the 26th Symposium on Operating Systems Principles*. 2017, pp. 1–18.

[137] Ana Pereira and Carsten Thomas. “Challenges of machine learning applied to safety-critical cyber-physical systems”. In: *Machine Learning and Knowledge Extraction* 2.4 (2020), pp. 579–602.

[138] Ana Petrovska et al. “Defining adaptivity and logical architecture for engineering (smart) self-adaptive cyber–physical systems”. In: *Information and Software Technology* 147 (2022), p. 106866.

[139] Buu Phan et al. “Bayesian uncertainty quantification with synthetic data”. In: *International Conference on Computer Safety, Reliability, and Security*. Springer. 2019, pp. 378–390.

[140] *Prism artifacts*. URL: `https://people.cs.kuleuven.be/~danny.weyns/software/ActivFORMS/supplement/index.htm`.

[141] Georg Püschel et al. “Towards systematic model-based testing of self-adaptive software”. In: *Proceedings of the 5th International Conference on Adaptive and Self-Adaptive Systems and Applications (ADAPTIVE)*. Citeseer. 2013, pp. 65–70.

[142] Morgan Quigley et al. “ROS: an open-source Robot Operating System”. In: *ICRA workshop on open source software*. Vol. 3. 3.2. Kobe, Japan. 2009, p. 5.

[143] Federico Quin, Danny Weyns, and Omid Gheibi. “Reducing large adaptation spaces in self-adaptive systems using classical machine learning”. In: *Journal of Systems and Software* 190 (2022), p. 111341.

[144] Mona Rahimi et al. “Toward requirements specification for machine-learned components”. In: *2019 IEEE 27th International Requirements Engineering Conference Workshops (REW)*. IEEE. 2019, pp. 241–244.

[145] Pethuru Raj. *Architectural Patterns: Uncover essential patterns in the most indispensable realm of enterprise architecture*. Packt Publishing Limited, 2017.

[146] *Rambo steering angle prediction model*. 2016. URL: `https://github.com/udacity/self-driving-car/tree/master/steering-models/community-models/rambo`.

[147] Martina Rapp, Max Scheerer, and Ralf Reussner. “Design-Time Performability Optimization of Runtime Adaptation Strategies”. In: *Companion of the 2022 ACM/SPEC International Conference on Performance Engineering*. ICPE ’22. Bejing, China: Association for Computing Machinery, 2022, pp. 113–120. ISBN: 9781450391597. DOI: `10.1145/3491204.3527471`. URL: `https://doi.org/10.1145/3491204.3527471`.

[148] Samik Raychaudhuri. “Introduction to monte carlo simulation”. In: *2008 Winter simulation conference*. IEEE. 2008, pp. 91–100.

[149] Ralf H Reussner et al. *Modeling and simulating software architectures: The Palladio approach*. MIT Press, 2016.

[150] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. “Anchors: High-precision model-agnostic explanations”. In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 32. 1. 2018.

[151] Rick Salay and Krzysztof Czarnecki. “Improving ml safety with partial specifications”. In: *International Conference on Computer Safety, Reliability, and Security*. Springer. 2019, pp. 288–300.

[152] Rick Salay and Krzysztof Czarnecki. “Using machine learning safely in automotive software: An assessment and adaption of software process requirements in ISO 26262”. In: *arXiv preprint arXiv:1808.01614* (2018).

[153] Mazeiar Salehie and Ladan Tahvildari. “Self-adaptive software: Landscape and research challenges”. In: *ACM transactions on autonomous and adaptive systems (TAAS)* 4.2 (2009), pp. 1–42.

[154] P Santhanam, Eitan Farchi, and Victor Pankratius. “Engineering reliable deep learning systems”. In: *arXiv preprint arXiv:1910.12582* (2019).

[155] Max Scheerer. *Environmental Dynamics*. URL: `https://github.com/PalladioSimulator/Palladio-Addons-EnvironmentalDynamics`.

[156] Max Scheerer. *Reliability prediction of AI-enabled systems*. URL: `https://github.com/PalladioSimulator/Palladio-Analyzer-Dependability-ML`.

[157] Max Scheerer and Martina Rapp. *SimExp Framework - Evaluation of adaptation strategies for self-adaptive systems*. URL: `https://github.com/PalladioSimulator/Palladio-Analyzer-SimExp`.

[158] Max Scheerer, Martina Rapp, and Ralf Reussner. "Design-Time Validation of Runtime Reconfiguration Strategies: An Environmental-Driven Approach". In: *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. IEEE. 2020, pp. 75–81.

[159] Max Scheerer and Ralf Reussner. "Reliability Prediction of Self-Adaptive Systems Managing Uncertain AI Black-Box Components". In: *2021 International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE. 2021, pp. 111–117.

[160] Max Scheerer et al. "Towards classes of architectural dependability assurance for machine-learning-based systems". In: *Proceedings of the IEEE/ACM 15th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. 2020, pp. 31–37.

[161] Gesina Schwalbe and Martin Schels. "A survey on methods for the safety assurance of machine learning based systems". In: *10th European Congress on Embedded Real Time Software and Systems (ERTS 2020)*. 2020.

[162] David Sculley et al. "Hidden technical debt in machine learning systems". In: *Advances in neural information processing systems* 28 (2015).

[163] Alex Serban, Erik Poll, and Joost Visser. "Towards using probabilistic models to design software systems with inherent uncertainty". In: *European Conference on Software Architecture*. Springer. 2020, pp. 89–97.

[164] Alexandru Constantin Serban. "Designing safety critical software systems to manage inherent uncertainty". In: *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*. IEEE. 2019, pp. 246–249.

[165] Sanjit A Seshia, Dorsa Sadigh, and S Shankar Sastry. "Towards verified artificial intelligence". In: *arXiv preprint arXiv:1606.08514* (2016).

[166] Lui Sha et al. "Using simplicity to control complexity". In: *IEEE Software* 18.4 (2001), pp. 20–28.

[167] Sina Shafaei et al. "Uncertainty in machine learning: A safety perspective on autonomous driving". In: *International Conference on Computer Safety, Reliability, and Security*. Springer. 2018, pp. 458–464.

[168] Stepan Shevtsov, Danny Weyns, and Martina Maggio. "SimCA* A Control-theoretic Approach to Handle Uncertainty in Self-adaptive Systems with Guarantees". In: *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 13.4 (2019), pp. 1–34.

[169] Yong-Jun Shin, Joon-Young Bae, and Doo-Hwan Bae. "Concepts and Models of Environment of Self-Adaptive Systems: A Systematic Literature Review". In: *2021 28th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE. 2021, pp. 296–305.

[170] Hai Shu and Hongtu Zhu. "Sensitivity analysis of deep neural networks". In: *Proceedings of the AAAI Conference on Artificial Intelligence.* Vol. 33. 01. 2019, pp. 4943–4950.

[171] Ravid Shwartz-Ziv and Naftali Tishby. "Opening the black box of deep neural networks via information". In: *arXiv preprint arXiv:1703.00810* (2017).

[172] Gagandeep Singh et al. "An abstract domain for certifying neural networks". In: *Proceedings of the ACM on Programming Languages* 3.POPL (2019), pp. 1–30.

[173] Ian Sommerville. *Software Engineering, 9/E.* Pearson Education India, 2011.

[174] Matthijs TJ Spaan. "Partially observable Markov decision processes". In: *Reinforcement Learning.* Springer, 2012, pp. 387–414.

[175] Herbert Stachowiak. *Allgemeine modelltheorie.* Springer, 1973.

[176] Thomas Stahl, Markus Völter, and Krzysztof Czarnecki. *Model-driven software development: technology, engineering, management.* John Wiley & Sons, Inc., 2006.

[177] Dave Steinberg et al. *EMF: eclipse modeling framework.* Pearson Education, 2008.

[178] Christian Stier. "Adaptation-aware architecture modeling and analysis of energy efficiency for software systems". PhD thesis. Karlsruhe Institute of Technology, Germany, 2018.

[179] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. "Axiomatic attribution for deep networks". In: *International Conference on Machine Learning.* PMLR. 2017, pp. 3319–3328.

[180] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction.* Vol. 1. 1. MIT press Cambridge, 1998.

[181] Marius Take et al. "Software Design Patterns for AI-Systems". In: *Proceedings of the 11th International Workshop on Enterprise Modeling and Information Systems Architectures (EMISA 2021). Hrsg.: A. Koschmider.* 2021, p. 30.

[182] Moeka Tanabe et al. "Learning environment model at runtime for self-adaptive systems". In: *Proceedings of the Symposium on Applied Computing.* 2017, pp. 1198–1204.

[183] Ömer Şahin Taş et al. "Functional system architectures towards fully automated driving". In: *Intelligent Vehicles Symposium (IV), 2016 IEEE.* IEEE. 2016, pp. 304–309.

[184] R.N. Taylor, N. Medvidovic, and E. Dashofy. *Software Architecture: Foundations, Theory, and Practice.* Wiley, 2009. ISBN: 9780470167748. URL: `https://books.google.de/books?id=j9pdGQAACAAJ`.

[185] Jakob Thumm and Matthias Althoff. "Provably Safe Deep Reinforcement Learning for Robotic Manipulation in Human Environments". In: *arXiv preprint arXiv:2205.06311* (2022).

[186] Yuchi Tian et al. "Deeptest: Automated testing of deep-neural-network-driven autonomous cars". In: *Proceedings of the 40th international conference on software engineering.* 2018, pp. 303–314.

[187] Yuchi Tian et al. "Testing DNN image classifiers for confusion & bias errors". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 2020, pp. 1122–1134.

[188] David Timmermann et al. "A Hybrid Approach for Object Localization Combining Mask R-CNN and Halcon in an Assembly Scenario". In: *2021 IEEE 8th International Conference on Industrial Engineering and Applications (ICIEA)*. IEEE. 2021, pp. 270–276.

[189] Hanghang Tong et al. "Blur detection for digital images using wavelet transform". In: *2004 IEEE international conference on multimedia and expo (ICME)(IEEE Cat. No. 04TH8763)*. Vol. 1. IEEE. 2004, pp. 17–20.

[190] Marc Toussaint, Amos Storkey, and Stefan Harmeling. "Expectation-Maximization methods for solving (PO) MDPs and optimal control problems". In: *Inference and Learning in Dynamic Models* (2010).

[191] *Udacity self-driving car challenge - CH2_001 Dataset*. 2016. URL: `https://github.com/udacity/self-driving-car/tree/master/datasets/CH2`.

[192] *Udacity self-driving car challenge 2*. 2016. URL: `https://github.com/udacity/self-driving-car/tree/master/challenges/challenge-2`.

[193] Mark Utting and Bruno Legeard. *Practical model-based testing: a tools approach*. Elsevier, 2010.

[194] Jeroen Van Der Donckt et al. "Applying deep learning to reduce large adaptation spaces of self-adaptive systems with multiple types of goals". In: *Proceedings of the IEEE/ACM 15th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. 2020, pp. 20–30.

[195] Kush R Varshney and Homa Alemzadeh. "On the safety of machine learning: Cyber-physical systems, decision sciences, and data products". In: *Big data* 5.3 (2017), pp. 246–255.

[196] David Verstraete et al. "Deep learning enabled fault diagnosis using time-frequency image analysis of rolling element bearings". In: *Shock and Vibration* 2017 (2017).

[197] Andreas Vogelsang and Markus Borg. "Requirements engineering for machine learning: Perspectives from data scientists". In: *2019 IEEE 27th International Requirements Engineering Conference Workshops (REW)*. IEEE. 2019, pp. 245–251.

[198] Markus Völter et al. *Model-driven software development: technology, engineering, management*. John Wiley & Sons, 2013.

[199] Laura Von Rueden et al. "Informed machine learning–towards a taxonomy of explicit integration of knowledge into machine learning". In: *Learning* 18 (2019), pp. 19–20.

[200] Akifumi Wachi and Yanan Sui. "Safe reinforcement learning in constrained markov decision processes". In: *International Conference on Machine Learning*. PMLR. 2020, pp. 9797–9806.

[201] Shiqi Wang et al. "Efficient formal safety analysis of neural networks". In: *Advances in Neural Information Processing Systems*. 2018, pp. 6367–6377.

[202] Shiqi Wang et al. "Formal security analysis of neural networks using symbolic intervals". In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018, pp. 1599–1614.

[203] Hironori Washizaki et al. "Studying software engineering patterns for designing machine learning systems". In: *2019 10th International Workshop on Empirical Software Engineering in Practice (IWESEP)*. IEEE. 2019, pp. 49–495.

[204] Gereon Weiss et al. "Towards integrating undependable self-adaptive systems in safety-critical environments". In: *Proceedings of the 13th International Conference on Software Engineering for Adaptive and Self-Managing Systems*. 2018, pp. 26–32.

[205] Wiphada Wettayaprasit, Nasith Laosen, and Salinla Chevakidagarn. "Data filtering technique for neural networks forecasting". In: *Proceedings of the 7th WSEAS International Conference on Simulation, Modelling and Optimization*. World Scientific, Engineering Academy, and Society (WSEAS). 2007, pp. 225–230.

[206] Danny Weyns. *An Introduction to Self-adaptive Systems: A Contemporary Software Engineering Perspective*. John Wiley & Sons, 2020.

[207] Danny Weyns and M Usman Iftikhar. "Activforms: A model-based approach to engineer self-adaptive systems". In: *arXiv preprint arXiv:1908.11179* (2019).

[208] Danny Weyns and Usman Iftikhar. "Model-based simulation at runtime for self-adaptive systems". In: *Proceeding Models at Runtime, Würzburg 2016* (2016), pp. 1–9.

[209] Danny Weyns et al. "Applying architecture-based adaptation to automate the management of internet-of-things". In: *European Conference on Software Architecture*. Springer. 2018, pp. 49–67.

[210] *Who's responsible when an autonomous car crashes?* `http://money.cnn.com/2016/07/07/technology/tesla-liability-risk/index.html`. 2016.

[211] Hui Xu et al. "NV-DNN: towards fault-tolerant DNN systems with N-version programming". In: *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE. 2019, pp. 44–47.

[212] Haruki Yokoyama. "Machine learning system architectural pattern for improving operational stability". In: *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*. IEEE. 2019, pp. 267–274.

[213] Jie M Zhang et al. "Machine learning testing: Survey, landscapes and horizons". In: *IEEE Transactions on Software Engineering* (2020).

[214] Mengshi Zhang et al. "Deeproad: Gan-based metamorphic testing and input validation framework for autonomous driving systems". In: *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2018, pp. 132–142.

[215] Tianqi Zhao et al. "A reinforcement learning-based framework for the generation and evolution of adaptation rules". In: *2017 IEEE International Conference on Autonomic Computing (ICAC)*. IEEE. 2017, pp. 103–112.

[216] Qi Zhu et al. "Safety-assured design and adaptation of learning-enabled autonomous systems". In: *2021 26th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE. 2021, pp. 753–760.

[217] Luisa M Zintgraf et al. "Visualizing deep neural network decisions: Prediction difference analysis". In: *arXiv preprint arXiv:1702.04595* (2017).

A. Results of Architectural Configurations Predicted by *SimExp* for the DeltaIoT Case Study System

In the following, a collection of tables are depicted which show the averaged architectural configurations predicted by the *SimExp* method. In each scenario, 10 trajectories are sampled where each trajectory has some fixed length N. The architectural configurations are calculated by averaging the transmission powers and distribution factors of the Nth sampled state (i.e. the last state) by the factor of 10 (i.e. the total number of sampled trajectories).

	Default Strategy			
	DeltaIoT		***SimExp***	
Link	**Power**	**Distribution**	**Power**	**Distribution**
2 to 4	15	100	15	100
3 to 1	0	100	0	100
4 to 1	12	100	13	100
5 to 9	0	100	0	100
6 to 4	15	100	15	100
7 to 2	0	0	0	3
7 to 3	0	100	0	97
8 to 1	0	100	0	100
9 to 1	0	100	0	100
10 to 6	15	50	15	100
10 to 5	8	50	7.1	0
11 to 7	6	100	5.3	100
12 to 7	0	0	0	1
12 to 3	15	100	15	99
13 to 11	15	100	15	100
14 to 12	0	100	0	100
15 to 12	0	100	0	100

Table A.1.: Overview of the average configurations of DeltaIoT and *SimExp* to which the strategy π_D converges after sampling 96 states for 10 runs.

	Quality-based Strategy											
	DeltaIoT $\beta = 10$		***SimExp*** $\beta = 30.5$		**DeltaIoT** $\beta = 18$		***SimExp*** $\beta = 32$		**DeltaIoT** $\beta = 26$		***SimExp*** $\beta = 34.5$	
Link	**Power**	**Dist.**	**Power**	**Dist.**	**Power**	**Dist.**	**Power**	**Dist.**	**Power**	**Dist.**	**Power**	**Dist.**
2 to 4	15	100	14	100	15	100	15	100	15	100	15	100
3 to 1	0	100	0	100	0	100	0	100	15	100	14	100
4 to 1	12	100	12	100	12.1	100	13	100	15	100	13.9	100
5 to 9	0	100	0	100	0	100	0	100	15	100	13.9	100
6 to 4	15	100	14	100	15	100	15	100	15	100	15	100
7 to 2	0	0	15	0	0	96	8.1	100	15	0	15	0
7 to 3	0	100	7	100	0	4	7	0	15	100	14.1	100
8 to 1	0	100	0	100	0	100	0	100	15	100	13.9	100
9 to 1	0	100	0	100	0	100	0	100	15	100	14.1	100
10 to 6	15	10	14.6	0	15	5	14.7	0	15	0	15	0
10 to 5	7	90	7	100	7.1	95	8	100	15	100	14	100
11 to 7	5	100	5	100	5.1	100	6	100	15	100	13.9	100
12 to 7	15	0	15	0	0	100	8.1	100	15	93	14.6	92
12 to 3	15	100	14	100	15	0	15	0	15	7	15	8
13 to 11	15	100	14	100	15	100	15	100	15	100	15	100
14 to 12	0	100	0	100	0	100	0	100	15	100	14.1	100
15 to 12	0	100	0	100	0	100	0	100	15	100	14.1	100

Table A.2.: Overview of the average configurations of DeltaIoT and *SimExp* to which the strategy π_Q converges for varying bounds. In terms of *SimExp*, 96 states have been sampled for 10 runs; regarding DeltaIoT 500 states have been simulated for 10 runs.