Automatic Context-Based Policy Generation from Usage- and Misusage-Diagrams

Master's Thesis of

Thomas Lieb

at the Department of Informatics
Institute for Program Structures and Data Organization (IPD)

Reviewer:	Prof. Dr. Ralf H. Reussner
Second reviewer:	Prof. Dr.-Ing. Anne Koziolek
Advisor:	M.Sc. Maximilian Walter
Second advisor:	Dr. rer. nat. Robert Heinrich

15. June 2020 – 14. December 2020

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Stuttgart, 14.12.2020

. . . . . .

(Thomas Lieb)

Abstract

In systems with a very dynamic process like Industry 4.0, contexts of all participating entities often change and a lot of data exchange happens with external organizations such as suppliers or producers which brings concern about unauthorized data access. This creates the need for access control systems able to handle such a combination of a highly dynamic system and the arising concern about the security of data. In many situations the decision for access control depends on the context information of the requester. One concept to support the systematic development of secure systems for these dynamic environments is the context meta-model. It is a model-driven security approach which applies the concepts of model-driven software development to the field of software security with the help of context-based access policies. Another problem of dynamic system is that the manual development of access policies can be time consuming and expensive. Approaches using automated policy generation have shown to reduce this effort. In this master thesis we introduce a concept which combines the context based model-driven security with automated policy generation and evaluate if it is a suitable option for the creation of access control systems and if it can reduce the effort in policy generation. The approach makes use of usage and misusage diagrams which are on a high architectural abstraction level to derive and combine access policies for data elements which are located on a lower abstraction level. The approach was evaluated using four case studies and it was shown that it is an accurate method for creating policies and an effort reduction can be achieved in some cases.

Zusammenfassung

In Systemen mit sehr dynamischen Prozessen wie in der Industrie 4.0 ändern sich häufig die Kontexte aller beteiligten Teilnehmer und es findet ein großer Datenaustausch mit externen Organisationen wie Lieferanten oder Herstellern statt, was Bedenken hinsichtlich nicht autorisierter Datenzugriffe hervorruft. Dies schafft die Notwendigkeit für Zugangskontrollsysteme, die diese Kombination aus hochdynamischem System mit gleichzeitiger Sicherheit der Daten bewältigen können. In vielen Situationen hängt die Entscheidung für die Zugriffskontrolle von den Kontextinformationen des Anforderers ab. Ein Konzept zur Unterstützung der systematischen Entwicklung sicherer Systeme für diese dynamischen Umgebungen ist das Kontext-Metamodell. Es handelt sich um einen modellgetriebenen Sicherheitsansatz, bei dem die Konzepte der modellgetriebenen Softwareentwicklung mithilfe kontextbasierter Zugriffsrichtlinien auf den Bereich der Software-Sicherheit angewendet werden. Ein weiteres Problem der dynamischen Systeme besteht darin, dass die manuelle Entwicklung von Zugriffsrichtlinien zeitaufwändig und teuer sein kann. Ansätze mit automatisierter Richtlinienerstellung haben gezeigt, dass sich dadurch der Aufwand verringeren lässt. In dieser Masterarbeit stellen wir ein Konzept vor, das die kontextbasierte modellgetriebene Sicherheit mit der automatisierten Richtlinienerstellung kombiniert und bewertet, ob dies eine geeignete Option für die Erstellung von Zugriffsrichtlinien ist und ob es den Aufwand bei der Richtlinienerstellung verringern kann. Der Ansatz verwendet Usage- und Misusage-Diagramme, die sich auf einer hohen Architekturabstraktionsebene befinden, um Zugriffsrichtlinien für Datenelemente abzuleiten und zu kombinieren, die sich auf einer niedrigeren Abstraktionsebene befinden. Der Ansatz wurde anhand von vier Fallstudien bewertet und es wurde gezeigt, dass es sich um eine genaue Methode zur Erstellung von Richtlinien handelt und in einigen Fällen eine Reduzierung des Aufwands erreicht werden kann.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

The ongoing automation of traditional manufacturing processes with the help of smart technology is known as the fourth industrial revolution, or Industry 4.0 [13]. In the global production of goods, far reaching digital networks and the fundamental restructuring of production can lead to improved product and service delivery, a boost in overall productivity, reduced labor cost and energy consumption, resulting in more cost-efficient products [6]. Massive networking leads to a large amount of data exchange which also occurs with outside organizations such as suppliers or producers, bringing concern about unauthorized access to each other's data [37]. "The protection objectives here are availability, integrity, confidentiality and legally compliant use (e.g. privacy) of the resources or data" [13]. This creates the need for access control systems able to handle this combination of a highly complex and dynamic system and the arising concern about the security of data.

A concept created to support the systematic development of secure systems is model-driven security [34]. It applies the concepts of model-driven software development to the field of software security. Compared to manual assessment with inspections, which are time consuming and error-prone, the automation of techniques during the realization of secure software systems could guide security analysts towards a more complete inspection of their software design [43].

One approach combining model-driven security with access control is proposed by Bolz et al. [5]. Their approach introduces a context meta-model, which allows the representation of context-based confidentiality properties for a dataflow analysis on the architectural level.

The problem that manual development and maintenance of access policies for these highly dynamic systems is time consuming and expensive [18][19] still remains. Automatic generation of access policies can reduce this effort [46][10], while simultaneously allowing the modeling of more dynamic and contextual systems [44].

In their paper, Fernandez and Hawkins [15] propose the use of use cases as a convenient way of creating access control policies. According to them this has multiple benefits. Firstly, it does not violate the principle of defining authorization at the highest possible level, at which their semantics are still explicit [17]. Secondly, use cases are needed anyway during the development of new systems and for the creation of new architectures during the restructuring of legacy systems. They assume that the security administration of such a system should be much easier compared to current systems.

In this thesis we want to combine the concept of utilizing usage descriptions of system for the definition of access policies with the concept of automated policy generation in order to derive context-based access policies from usage and misusage diagrams.

1.2 Contribution of the Thesis

In this master thesis we introduce a concept which combines the context based model-driven security with automated policy generation. The approach uses usage and misusage diagrams which are on a high architectural abstraction to derive and combine access policies for data elements which are located on a lower abstraction level. Since multiple use cases affect the same data elements with different policies, we also introduce a concept with which the derivied policies on the data element level can be combined and reduced. With this concept, we try to minimize the number of policies while keeping the system policy the same. The derived policies can also contradict each other, either through an error in the configuration or unplanned interaction in the design, which can be checked and validated with the introduced approach.

1.3 Outline of the Thesis

The following briefly describes the structure of the thesis. In Chapter 2, the running example is introduced which is set in the context of this thesis and will be used in later chapters to illustrate how certain concepts work. Chapter 3 describes the foundation of the approach which is needed for the rest of this thesis. In Chapter 4, the state of the art of the used concepts is presented. It summarizes them and describes their relevance while also showing how they differ to our work. Chapter 5 and 6 describe how the automated derivation and combination of policies works and how the different parts of the solution are designed. In Chapter 7, the architecture and implementation of the created software is explained. In Chapter 8, the thesis is evaluated using different case studies and their results. Finally, in Chapter 11, the master thesis is summarized and an outlook regarding future work is given.

2 Running Example

In this section, a running example is described. It introduces an example environment which is going to be used throughout this paper to illustrate the concepts of the thesis, to show how it can be modelled with the tools used and to describe how the designed approach is going to work. The running example describes a company which produces goods and has different security aspects which need to be considered. Figure 2.1 illustrates the running example.

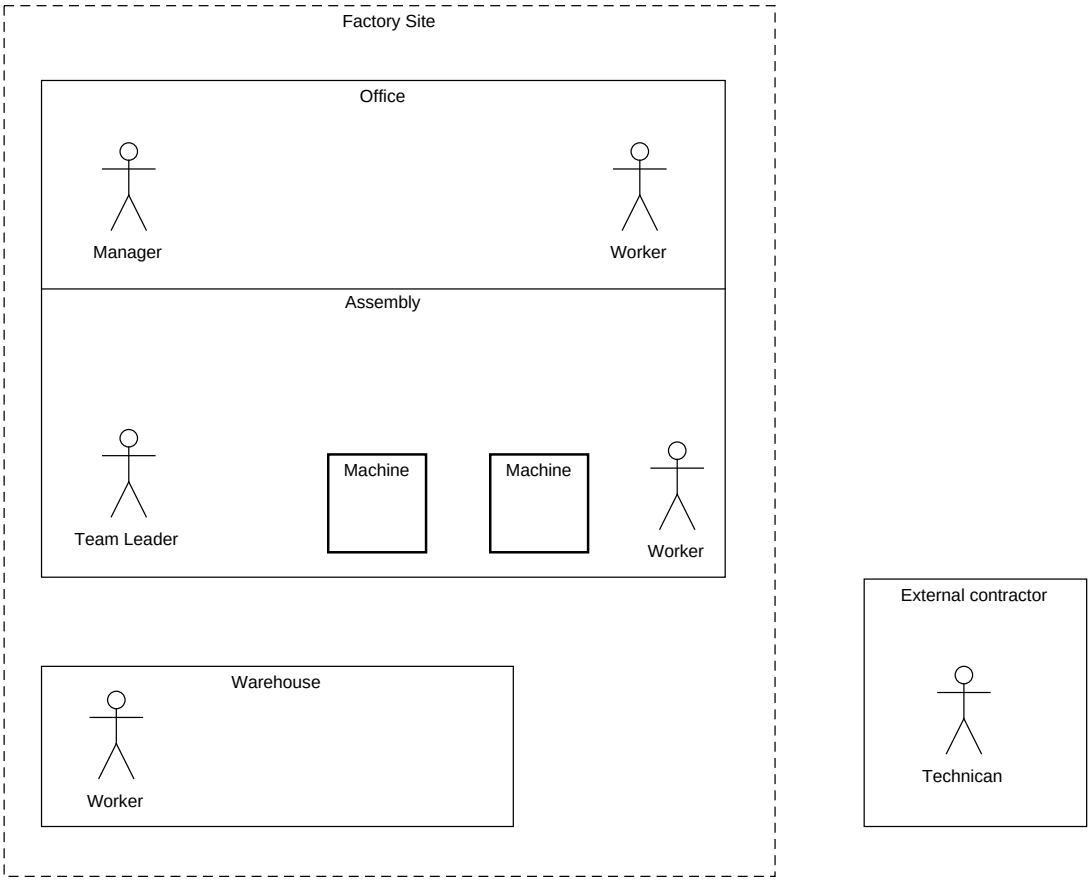


Figure 2.1: Running example illustration

The company's factory site has one main building, which is split into two areas: the office and the assembly, where the machines producing the goods are located. The warehouse is located in a second, smaller building. Workers inside the assembly are able to check the status of all machines, schedule new tasks or abort running operations. An employee

inside the warehouse should be able to check the status of all machines and their scheduled tasks, so he is able to prepare the neccessary parts for the upcoming tasks and free space for the incoming produced goods. Workers can have shifts assigned to them in which they work. Workers can change possible shifts themselves in advance up to a certain date, but only the manager or the team leader can change shifts for other employees. The company hired an external contractor to maintain their machines and fix them in case of failure. The technician employed by the contractor should not be able to access the machines of the company during normal operation. However, in case a failure should arise, the technician should be able to access the machine remotely and run different diagnostics on it, so he is able to prepare his equipment before visiting the company or in best case fix the issue from his desk. If the technician needs to fix the problem on-site, he needs to have access to the factory site. Here, only the manager or the team leader should have the rights to grant this access. The described use cases are listed in table 2.1.

ID	Use case description	Actor
1	The workers in the assembly should have full access to the machines	Worker
2	The workers in the warehouse should be able to see the current and the scheduled tasks	Worker
3	The workers in the office should be able to see the current and the scheduled tasks	Worker
4	The start of a new task should only be allowed if the person is in the assembly	All
5	An employee should be able to adjust his upcoming shifts up to a certain date	All
6	Only employees with a leading role should be able to adjust or assign shifts to employees	Manager, Team Leader
7	Only the manager should have access to the financial data of the company	Manager
8	The external technician should not have access to the machines during normal operation	Technician
9	In case of an error the technician should have access to the diagnostic function	Technician
10	The technician should never be able to start or stop tasks on the machine	Technician
11	The external worker needs to be granted temporary access to the factory site	Manager, Team Leader

Table 2.1: Textual description of allowed and forbidden usage of the example system

In the use case descriptions each actor has different contexts for which they should be allowed to execute the described behaviour or, for the negative case, should denied. A context for a worker is, for example, the role in the company, the currently assigned

shift, the position or location inside the company site, the type of task and if a remote or on-site access is needed. With our approach, the contexts which are assigned to the use cases descriptions should be derived to the data element they affect. Each task in the use case can affect multiple data elements. The derived access policies are specified with contexts from the usage diagram. Since a data element can be affected by multiple use cases and multiple allowed and forbidden context sets, the derived policies are combined and reduced to a minimal set and simultaneously validated for possible conflicts.

3 Foundation

This chapter describes the foundation of the thesis. A context based approach for access control is used which is based on Data-Driven Software Architecture. They use the Palladio Component Model as a modeling framework which is a model-driven software development approach. Additionally, a brief overview of access control strategies is given.

3.1 Model-Driven Software Development

Model-Based Software Development (MBSD) is an approach for which models are used as secondary artifacts for documentation and communication purposes. This use of models has several disadvantages [41]. Changes in the code have to be transferred to the model manually or the model and the code become inconsistent. Depending on the degree of the model's accuracy, these adaptions can be complex and time-consuming tasks.

Model-Driven Software Development (MDSD) removes these disadvantages by using models as primary artifacts, which means that they are treated equally to the code. By finding domain-specific abstractions and making them accessible through formal modeling, they have great potential for automatic code generation. This can lead to an increase in the productivity of the software developing process and in the quality and maintainability of the software system. To describe the rules on which these models are built, models of these models are needed: the so-called meta-models[41].

3.2 Palladio Component Model

Palladio is a tool-supported simulator for software architecture. It can be used for the prediction of several quality properties of software. It was initially developed by the Karlsruhe Institute of Technology (KIT), the FZI Research Center for Information Technology and the University of Paderborn. The Palladio Component Model (PCM) is a detailed meta-model for component-based software architecture which is based on the Eclipse Modeling Framework (EMF) [3].

Figure 3.1 shows a PCM instance with the different models contained in the PCM and the roles which are responsible for them. In the following sections, the different parts of the PCM are briefly explained.

3.2.1 Roles

In model-driven software development and in Palladio various roles work on different parts of the architecture where they contribute their specific knowledge.

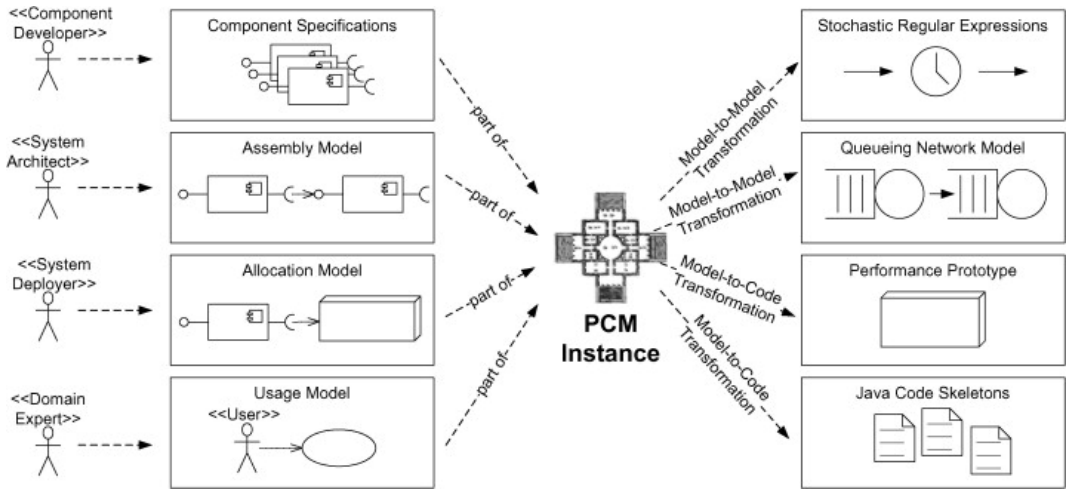


Figure 3.1: Palladio component model [3, Fig. 1]

- The architecture and the relationship between the individual components are designed by the *Software Architect*. He is responsible for passing on further instructions to the other roles. At the same time, he is responsible for the system model that defines the composition of the components characterized.

- Knowledge of how the user interacts with the system and which parameters are used in the control flow come from the *Domain Expert*. The modeling of this information is done in the usage model.

- The *Component Developer* is responsible for implementing and specifying the individual components. He models these in the repository model. It contains the individual components and interfaces.

- The *Software Distribution Expert* composes the system environment for the software. This is done in the execution environment model. Also, the software distribution expert instructs resources to the components. This is modeled in the allocation model.

The models that are developed and maintained by the various roles together form a Palladio model instance.

3.2.2 Models

The individual roles are responsible for certain sub-models which, as a whole, make up the complete architecture description. In the following, the five sub-models in PCM are described in more detail. For our approach, the first three are relevant, but the case studies used in the evaluation have a complete PCM model with all five parts.

3.2.2.1 Repository model

The repository model (Fig. 3.2) contains data types, interfaces and components. A component can offer its functionality to the outside via an interface. This is modeled with a *ProvidedRole*. It can also request a specific interface must be provided to it in order to use the functionality of other components. This will be modeled with a *RequiredRole*. Interfaces contain a collection of signatures, which represent an operation.

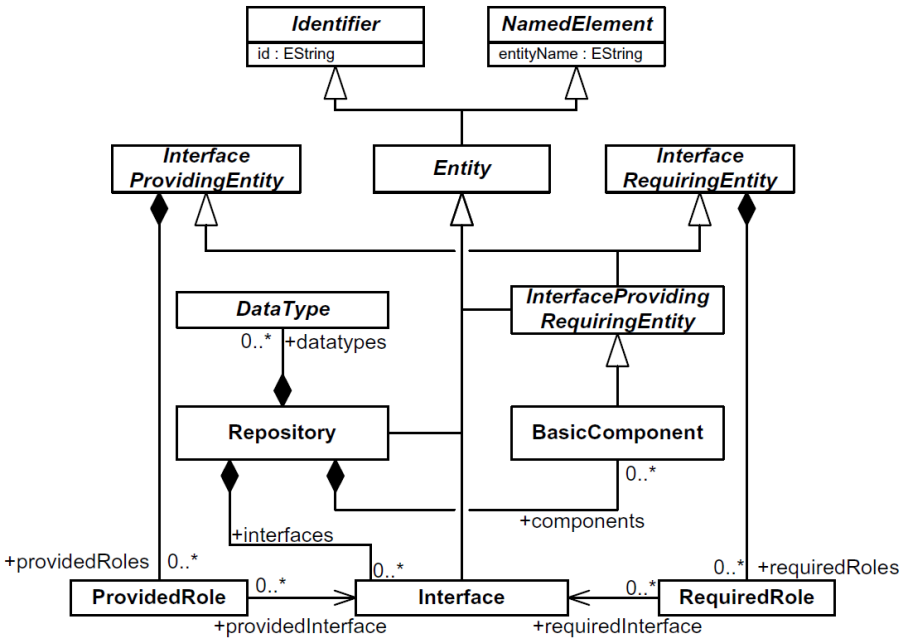


Figure 3.2: Repository model [3, Fig. 3]

With the help of a Service Effect Specification (*SEFF*) the behaviour within and between the components can be described (Fig. 3.3). A *SEFF* adds a behaviour description to a signature in the component. The behaviour can be modeled with an *InternalCallAction* and an *ExternalCallAction*. An *InternalCallAction* represents internal behaviour within the component. An *ExternalCallAction* calls an operation in another component. Branches in the behaviour can be modeled with a *BranchAction*. Either a probability or a condition can be specified which decides which branch is taken. Using a *UsageVariable*, a parameter assignment for the call can be specified. In addition, the *SEFF* resource requirements can be specified in regards to certain resources, such as *CPU*. The resource requirements are specified as abstract work packages. In a concrete execution environment, in which the resources are specified, the time values can be derived from the abstract work packages. With this behaviour specified in the *SEFF*, the model can be analyzed.

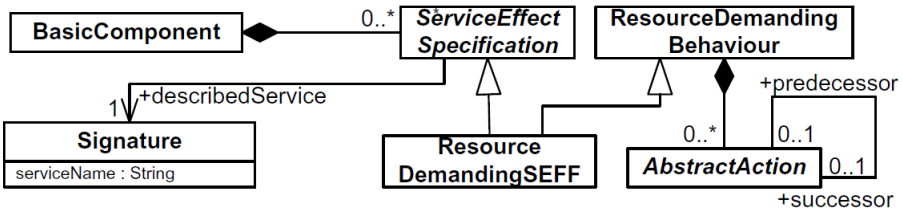


Figure 3.3: Service Effect Specification [3, Fig. 6]

3.2.2.2 System model

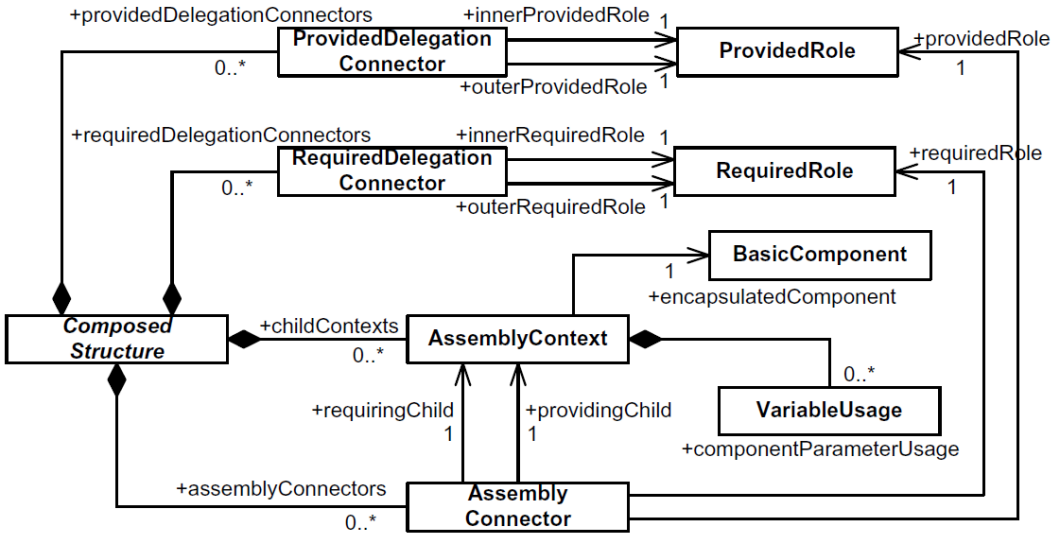


Figure 3.4: Composed structure in system model [3, Fig. 12]

The system model (Fig. 3.4) represents the software system that consists of a composition of the components defined in the repository model. Instances of components are represented in the system model with *AssemblyContext*. These can be connected with an *AssemblyConnector*. The *RequiredRole* of a component is linked to the *ProvidedRole* of a component. The system model must also provide at least one external interface so it is possible to interact with the system.

3.2.2.3 Usage model

The usage model describes the interaction of the user or external systems with the system (Fig. 3.5). With a *UsageScenario* the behaviour of a user can be specified. For a *UsageScenario* a workload can be set, which can be either an open or closed workload. A *OpenWorkload* models an infinite stream of users which arrive in specified time intervals (*ArrivalTime*) at the system and execute the scenario. A *ClosedWorkload* models a fixed number of users (*Population*) which execute the scenario, wait a specified amount of time (*ThinkTime*), and then execute it again. A scenario can call the operations which are provided by the system

via interfaces with an *EntryLevelSystemCall*. Therefore, the parameter assignments can also be specified using a *UsageVariable*.

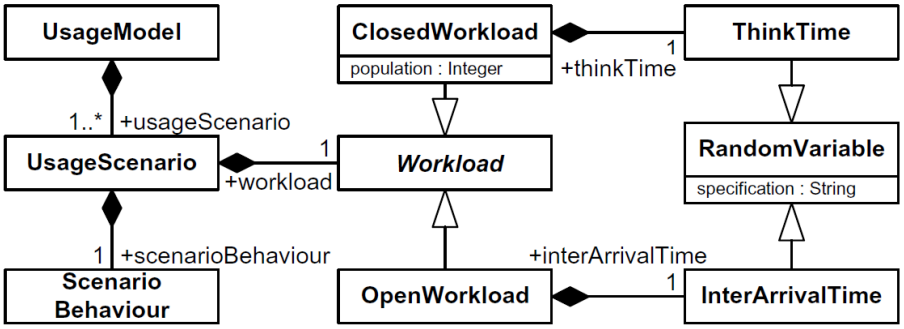


Figure 3.5: Usage model [3, Fig. 17]

3.2.2.4 Execution environment model

The execution environment model describes the used hardware and the network of the modeled software system. The elements *ResourceContainer* and *LinkingResourceContainer* are used. The *ResourceContainer* is a hardware resource, a *LinkingResource* is a channel between two *ResourceContainer*. For a *ResourceContainer*, different hardware resources can be specified. A *CPU* can be used as a processor, an *HDD* can be used as a hard disk and a *Delay* can model the delay between two actions.

3.2.2.5 Component-Allocation-Model

The allocation model describes how the individual component instances are distributed on the hardware described in the execution environment model. For this an *AllocationContext* is used.

3.2.3 Quality attributes

The Palladio bench can determine quality attributes for a system. They will be calculated from the components, their connections and the execution environment.

- Performance: Performance is one of the most important quality criteria of a system in regards to the timing constraints of the system. This applies to both time-critical and less critical systems. Performance includes the timing and resource usage of a system. This is measured using three metrics. The response time is the time between a request to a system and the received response. The throughput describes the number of requests processed per unit of time. The utilization relates to the active use of a resource and indicates the percentage it works per unit of time.
- Reliability: A system that offers a service as expected and with no side effects is called reliable. Any deviation from expected behaviour is considered an error. The overall reliability of a system is therefore expressed as the probability that the system is

working properly. Different errors can occur, for example software errors, hardware errors or network errors.

- Cost: Having many requirements usually leads to high costs. Therefore, between cost and performance or reliability must be weighed. Palladio enables the annotation of the cost of components and resources to estimate total costs and to make a trade-off with the quality attributes.

- Maintainability: Often during the development of software, there is more then one viable design possible. Palladio allows the calculation of the maintenance of the different alternatives on the system level by assigning maintenance efforts per design alternative and maintenance costs per design alternative.

While these four quality attributes are the main focus in normal software development, for the development of secure systems an additional attribute is added with the security aspect. There has been some development to bring extensions to the Palladio framework to enable it to perform analysis of this quality aspect of the system [5][40]. The contribution of our thesis will be in this area.

3.3 Data-Driven Software Architecture

Data-Driven Palladio is an extension of the PCM which integrates the concept of dataflow. This extension together with an analysis process for confidentiality create the development process of Data-Driven Software Architectures (DDSA) [40]. The Data-Driven Extension introduces data and data processing operators as first-class entities and Data-Driven Palladio supplies the meta-model, which defines how data is represented and which operations are to be performed on them. These data elements can have sets of characteristics which represent their abstract meta-data [39]. Access right mismatches are detected with the confidentiality analysis by comparing access rights assigned to data with roles assigned to the processing operations on this data. The fact that the access rights can be changed during the data processing steps is considered in this analysis. The confidentiality analysis is realized as a Prolog program and uses a Role-Based Access Control (RBAC) strategy to detect access right mismatches. The context meta-model introduced in section 3.6 is based on this modeling approach and extends it for the use of context-dependent access control policies [40].

3.4 Model-Driven Security

Firstly, software security is defined. The definition used in this thesis is derived from “A Reference Model of Information Assurance Security” [9] and splits software security in three parts:

- Confidentiality: "A system should ensure that only authorised users access information"

- Integrity: "A system should ensure completeness, accuracy and absence of unauthorised modifications in all its components"
- Availability: "A system should ensure that all the system's components are available and operational when they are required by authorised users"

Since our approach wishes to derive access control policies, we mainly focus on *Confidentiality*. A definition of how access control systems try to ensure *Confidentiality* is given in section 3.5.

Model-driven security (MDS) applies the model-driven approaches and the concepts of model-driven software development to security in order to solve software security problems. The security of software depends on a wide variety of factors and details. These vary from high-level aspects such as the design of policies to low-level aspects like avoiding insecure software patterns or avoiding buffer overflows. Many of these security concepts, goals or techniques can be described formally and validated or verified [1].

Since not each software developer has deep knowledge and comprehension of security related topics [22], model-driven approaches can help with the development of secure systems in multiple ways. Security-specific knowledge can be built into a part of the meta-model and the developers only have to use it. The automatic generation of source code can help prevent low-level security issues by preventing errors on the implementation level. Verifying models which were annotated by the role responsible for security could reveal issues in the system a developer of a single component might not be aware of.

3.5 Access Control Strategies

Access control mechanisms ensure the following properties of the system [4]:

(a) access rights to resources are granted only to authorized entities.

(b) access rights to resources are not denied to authorized entities.

There are four common techniques used for access control systems: mandatory access control (MAC) [26], discretionary access control (DAC) [25], role-based access control (RBAC) [16] and attributed-based access control (ABAC) [19]. These techniques are briefly defined with a focus on the core concepts. For all of them, several variations have been published.

3.5.1 Mandatory access control

In access control systems which use mandatory access control [26], labels are are assigned to resources and users. A label consists of a security level and a category. Security levels are defined in an ordered fashion, e.g. unrestricted, secret, top-secret. Users can access a resource if the category of their label matches the one on the resource and their security level is at least as high as the one of the resource. Managing such a system takes a high amount of work.

3.5.2 Discretionary access control

The discretionary access control [25] grants each user access to his own resources and the user can grant other users access to his resources. The access granted can vary in different ways from read-only to write, or even if the new users can grant access to the resource to other users. While this technique takes little managing effort it places the complete security management in the hands of the individual users.

3.5.3 Role Based Access Control

Role-based access control [16] consists of five different entities: users, roles, permissions, operations and objects. Users are grouped in roles which can be structured hierarchically. Permissions are assigned to each role. A permission is an operation like read, write or delete on a certain object. Access to an object is only granted if the role assigned the user is authorized to access the object. Users can have multiple roles assigned to them. It is currently the most used access control strategy [14].

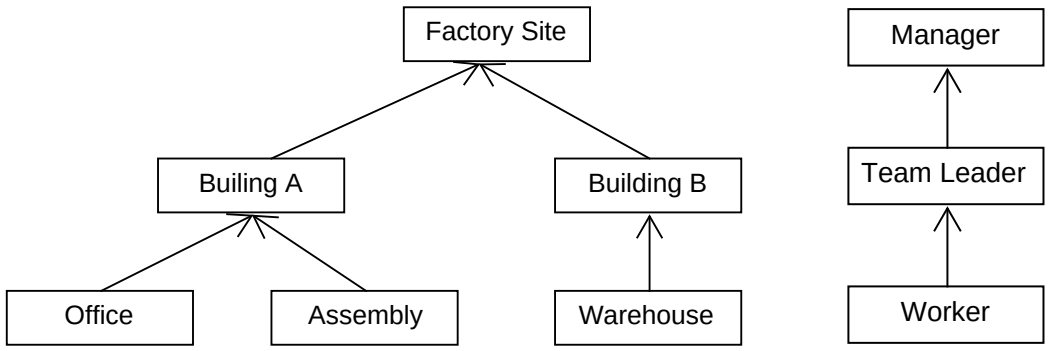
3.5.4 Attribute Based Access Control

Attribute-based access control [19] (also known as policy-based access control or rule-based access control) grants or denies access to resources based on attributes provided by the request. Depending on the scenario, everything can be an attribute, e.g. names, roles, job titles, the time of the request, the nature of the requested access like read or write or the type of the requested resource. Attribute-based policies are stored in permission assignment constraints (PACs) and can be expressed positively or negatively. Even though RBAC is currently the most used access control strategy, ABAC is predicted to be the future state of the art [14].

3.6 Context Meta Model

The *ContextModel* was introduced by Boltz et al. in "Context-Based Confidentiality Analysis for Industrial IoT" [5]. It is a model-driven security approach which extends the DDSA. With this meta model, it is possible to give elements in the PCM model certain contexts. Contexts serve two purposes: They can represent properties of the element, like a state, or they can be understood as an access rule if they are defined for data elements. The *ContextModel* defines three types of context classes, which are used to represent different properties. The class structure of the *ContextModel* and these different subclasses are shown in figure 3.8.

- *SimpleContext*: A *SimpleContext* represents a fixed global value. An example is the current status of a machine or the assigned shift of an employee. The machine status in this case would be an unique *ContextType*, since a machine can only have one status at a time. The shift context is not unique; it could be possible for an employee to be assigned a double shift like evening and night shift, or a day shift and on stand-by the rest of the time.



(a) Top Down (b) Bottom Up

Figure 3.7: Examples of *HierarchicalContexts*

- *HierarchicalContext*: *HierarchicalContext* can be used to represent the hierarchical dependencies of a property. In the running example, this would be applicable to the location of a worker inside the production plant or the rank of an employee. These hierarchical contexts are represented in figure 3.7. The location context in this case is a top-down context, meaning a context further down in the tree structure could have has more rights since it inherits all the access rights of the higher up contexts. For example, if a worker inside "Building A" is allowed to access a machine, it implies that workers from "Assembly" as well as workers from the "Office" have these rights. But if workers from "Assembly" are allowed to access the machines, it doesn't imply that all workers inside "Building A" have access rights. On the contrary, workers inside "Office" don't have access to the machines, unless explicitly stated in a different policy.

  As a notation in the shown graphs, the arrow indicates that the context set is a child of the other node, e.g. "Assembly" is a child node of "Building A". The inheritance direction defines if the child inherits from the parent or the other way round.

- *RelatedContext*: A *RelatedContext* can be used to refer to a different *ContextSet* which needs to be fulfilled in order for this context to be true.

To group different contexts of the same type, the *ContextType* can be used. As an example, the different shifts a worker can have, like day or night shift, are contexts of the same type. Here it should be ensured that two contexts of the same type are also of the same context class, otherwise the comparison of contexts of the same type might not work as expected. This contraint also ensures that the contexts of *HierarchicalContext* are all of the same type.

A *ContextSet* is a collection of *ContextAttributes*. For example, for the use case 3 of the running example the context set would be *{Worker, Office, Remote}*. The use case 9 could be described with the context set *{Technician, External, Remote, Failure}*. The attribute of the *ContextType* defines if multiple *ContextAttributes* of the same type are allowed or not.

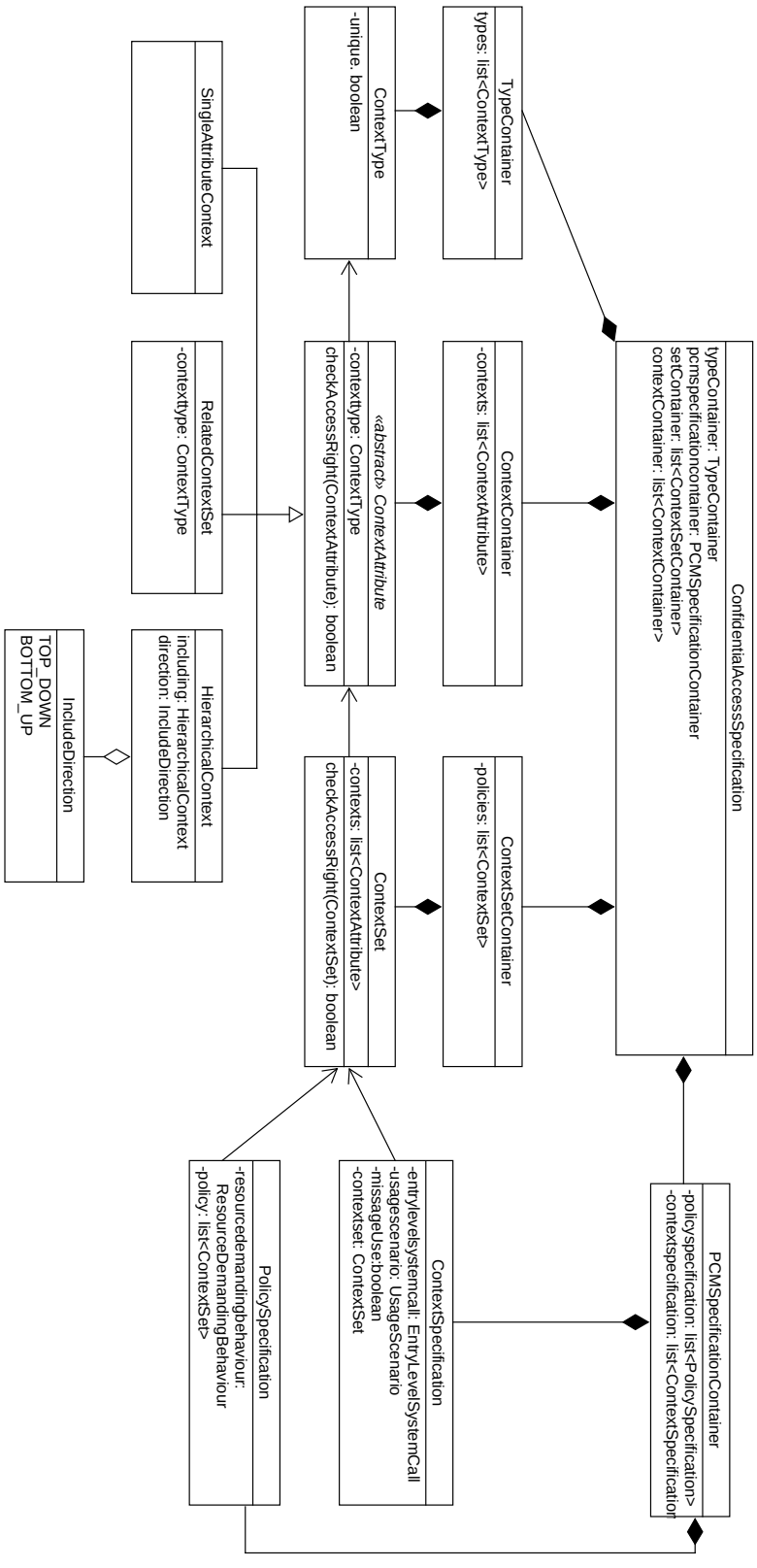


Figure 3.8: Class diagram of the *ContextModel*

These *ContextSet* can be assigned to *SEFFs* with a *PolicySpecification*, which describes the access policy for this data element. If the *ContextSet* is part of a *ContextSpecification*, it describes the context for which the related usage scenario should be able to be executed.

4 Related work

In this chapter similar concepts to that of this thesis are listed. Our approach has three different areas which have an influence, namely the automatic generation of access control policies, the model driven development of secure systems and the concept of security by design. The current state of the art is shown, the similarities and differences to our approach described and our contributions presented.

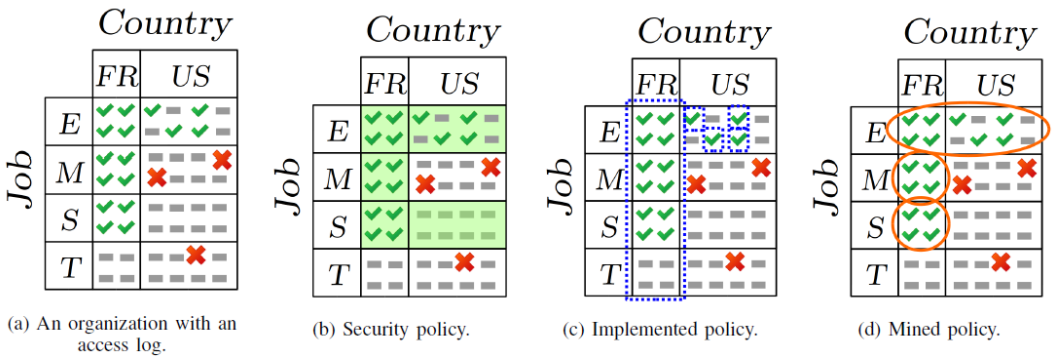
4.1 Automatic Policy Generation

In "On the Impact of Generative Policies on Security Metrics" [45] Verma et al. describe the impact of generated policies on the security metrics of a system. Since modern systems tend to gets more dynamic and complex and manual creation of policies becomes more and more difficult, they introduce concepts to determine if generating security policies is beneficial in improving the security of the system. We use this concept of establishing equivalence between a generated and a manual system for our evaluation setup in section 8.4.

In "Generative policy model for autonomic management" [44] an approach is presented with which managed devices are able to generate policies for their own operations. In the introduced management system an interaction graph contains all the allowed activities for each device. The policies for each device are then generated according to this interaction graph. This enables the architectures to achieve some form of self-management. The authors claim that their approach provides several benefits over the state of the art role based concepts and allows the modeling of more dynamic and contextual systems. Their paper illustrates the need for the automatic generation of policies in large scale modern systems. Compared to their approach we do not want to introduce new design artifacts like the interaction graph but want to use already existing artifacts. Additionally, we do not want to generate the policies during runtime as done with the managed devices but instead want to derive the context based access policies for the data elements during the design phase.

One concept for generating policies is *Policy Mining*. The survey "A Survey of Role Mining" [32] categorizes the differences of various RBAC mining concepts. The basic concept of role mining is that most organizations concerned with secure systems already have user-permission assignments defined in some form. This information is then used to identify new roles in the system. The survey was conducted since in recent years several role mining techniques have been developed. This fact shows again the need for automatic policy generation. Since the survey was conducted on RBAC systems we instead focus on approaches which use ABAC access control systems, as these are predicted to be the future state of the art [19]. A concept for role mining in ABAC systems is introduced in

"Mining Attribute-Based Access Control Policies from Logs" [46] and "Mining ABAC Rules from Sparse Logs" [10]. In figure 4.1 the basic concept of this policy mining concept is illustrated. In (a) the user-permissions of the current system are shown, in this case they are present in the form of log entries. (b) shows the system described by the organisational security policy and (c) the actual implemented ABAC policy. (d) shows the mined policy created by the ABAC mining algorithm.



(a) An organization with an access log. (b) Security policy. (c) Implemented policy. (d) Mined policy.

Figure 4.1: Basic concept of mining policies [10, Fig.1]

The similarity to our approach is that the usage and misusage diagrams represent a certain scenario which should be allowed or forbidden which is similar to the information in the log entries containing allowed or denied user requests. The difference is that in order for the log entries to be created the system has to be already implemented and running. These approaches are therefore more suitable to be used to migrate an existing system. Our approach uses the information in the usage and misusage diagrams to derive the access policies during the design phase. This also allows for the possible detection of security issues in the configuration of the system during this early phase in the development. The policy mining algorithms use generalization. An algorithm that generalizes well creates policies which not only fit the logged entries but also include non-logged requests for which a significant number of similar requests have been authorized. This concept was excluded from the scope of this thesis since it would have been too time consuming to develop in the given time frame. Instead it is outlined as future work in chapter 10.

The paper "Access control policy combining: Theory meets practice" [27] introduces a policy combining language which can express a variety of policy combining algorithms. This addresses the issue that many policy languages only have fixed policy combining strategies which makes it hard to extend them with possible new rules. In the paper the concepts of the currently existing policy combining algorithms were also briefly explained. These concepts were used to create the rules we defined for combining and reducing the derived context sets. We didn't use an existing policy combining language in our approach. Although this brings some disadvantages the advantage of our approach is that both the usage diagram context descriptions and the policy specifications are part of the context meta model. Therefore, we don't need to use transformations between different models and are not concerned about consistency between them.

4.2 Model Driven Development of Secure Systems

Our approach will be based on a model driven concept for developing secure systems. In this section we compare the available concepts and explain why we think the chosen concept best fits for our approach. In their survey *An Extensive Systematic Review on Model-Driven Development of Secure Systems* [34] Nguyen et al. grouped the significant MDS approaches. For the purpose of this thesis we focus on approaches which use structured system descriptions.

SecureUML [29] aims at modeling access control policies for RBAC and then uses these policies to transform them into a complete access control infrastructure. It extends UML diagrams with annotations for roles, permissions, users and access control policies. Although it can be applied to a wide range of scenarios, the fact that it is based on a RBAC strategy makes it difficult to extend for a context-based confidentiality.

UMLSec [21] is an approach for modeling secure software systems applicable to different platforms. It is extending UML with features to model systems and analyze their security. Its modeling process is use case driven which assigns every use case diagram a goal tree which can have three states: undetermined, satisfied or denied. Although the use case driven nature would suit our approach, *UMLSec* only supports RBAC access policies.

The *SecDFD* approach [42] allows to analyze security-centric information flow policies on the design model. It used a graphical notation similar to Data Flow Diagrams which has additional security concepts and helps in the early discovery of design flaws. Here the disadvantages are that a possible already existing architecture description of the system needs to be transferred into the graph notation and that having a single SecDFD graph for the complete system might be difficult for large architectures.

With the *iFlow* approach [23] a UML model of an application can used to automatically generate the code of the app as well as the formal models for it. The *MODELFLOW* language can be used to model systems including their behaviour and security requirements. The problem with this approach is setting up the complete security domain with transitions on the granularity level of methods can be difficult.

The context meta-model presented by Boltz et al. in “Context-Based Confidentiality Analysis for Industrial IoT” [5] introduces an approach to model context-based confidentiality properties on the architectural level. The context based nature of this approach adds the possibility to model the dynamic elements of the systems and make is similar to the concepts of ABAC access control systems.

The problem that all of these approaches have in common is that the manual creation and maintenance of policies is taking a lot of effort. In our approach we therefore introduce the possibility to derive policies from usage diagrams with the intention to reduce this effort. We use the context model for the modeling of the access policies since it allows both the context specifications for the usage and misusage diagrams and the policy specifications for the data elements to be modelled in the same meta model.

4.3 Security by Design

In their paper "A Scenario-Driven Role Engineering Process for Functional RBAC Roles" [33] Neumann and Strembeck present a scenario-driven role engineering process for RBAC roles which uses scenario as the main concept for creating policies. They see a scenario as a collection of permissions that are applied in a particular order to reach a predefined goal. The subject trying to execute this goal needs to own all permissions that are needed to complete every step of the particular scenario. Figure 4.2 shows the basic model they use in their approach. Permissions and constraints are defined in catalogs, tasks and work profiles are created in accordance to these permissions. A scenario is seen a series of tasks, and the work profiles which are equal to a role of a user consist of one or more tasks. From these definitions, a RBAC-model is derived. One benefit they see with their approach is that changes in functionality of the system can be easily added by adapting the scenario model from which the new RBAC-model is then derived.

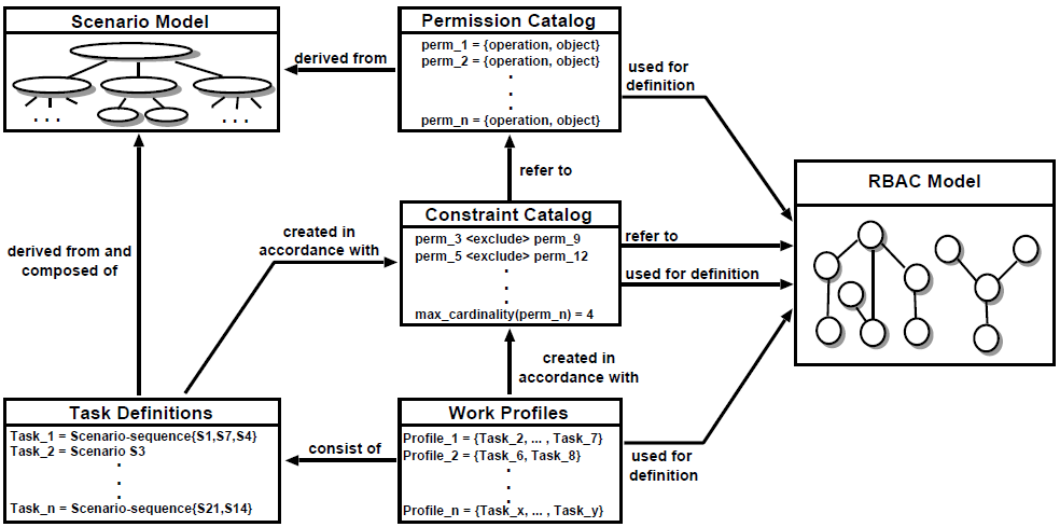


Figure 4.2: Scenario-driven role engineering process [33, Fig.4]

The similarity to this concept is the use of scenario models as the point of definition of permissions. Also the use of a permission and constrain catalog is similar to our concept of defining usage and misusage scenarios. Deriving the RBAC-model from the scenarios is similar to the derivation process we use in our approach. The difference to their approach is that they use roles for the definition of the access rights of an entity which is a role-based approach. By using context attributes we use a concepts which is more similar to the ABAC technique, since the contexts of an entity can be seen as attributes. The difference to a pure ABAC approach is that contexts can also specific the access policies of a data element. So a context fulfills two purposes, while an policy in ABAC is specified based on the relations of attributes.

Tuma et al. describe in their paper "Automating the Early Detection of Security Design Flaws" [43] an approach for the automatic detection of security flaws during the design phase of secure systems. Compared to manual assessment with design inspections, which

are time consuming and error-prone, their paper analyses the potential of automating the rules performed during an inspection in order to speed up the security analysis. They suggest that automated techniques during the realization of secure software systems could guide security analysts towards a more complete inspection of their software design. Our approach is similar in that by evaluating the derived policies from the usage and misusage diagrams potential design flaws in the software can be found. The difference is that their approach tries to find these security issues by comparing the models to design anti-pattern. In our approach the found issues are either within the context configuration of the system which indicates a user error, or the composition of the system leads to contradicting access policies which indicates a composition error.

5 Deriving Policies

The first part of this thesis derives policies from the use case descriptions to the data elements. The basic concept is explained in the first section, illustrating how the different models are used in our approach and describing the fundamental steps of the latter. Each step is then described in detail in the following sections.

5.1 Concept

Fernandez and Hawkins suggest the utilization of use case descriptions as a way of creating access right policies [15]. In the PCM the use case descriptions are modeled in the usage model as operations executed on the system. These system operations manipulate data elements in the components of the system. The access control of these single data elements is abstracted in this approach to operations. Operation are provided within the PCM by *SEFFs* in the repository model. The advantage of system models is that the domain expert, who is responsible for it, does not need to know the details of the system composition or how each component is implemented, only the interaction with the system needs to be specified. On the other hand, from the modeled information in the usage model it is not obvious which components and functions are called inside the system. Figure 5.2 illustrates this situation with the use case 1 of the running example. In the usage model, only the system call to start a new task is called. As a context for this action, the context set *{Worker, Assembly, NightShift}* is assigned. In the called system the call is connected to the *MachineController*. For each change in the current task the machine updates a central database in the system modelled as *TaskServer*. This allows for workers in the office to request the status of different machines. In this scenario, the derived context policy not only needs to be applied to the *SEFF startTask* of the machine, but also to the *updateTask SEFF* of the server.

The process of deriving policies can be grouped into three main steps as shown in figure 5.1.

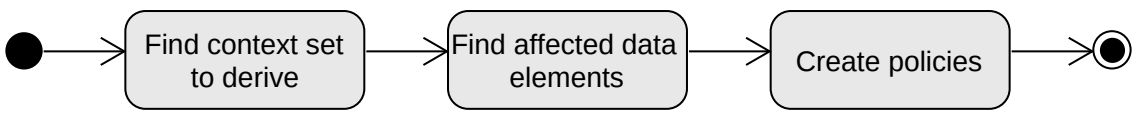


Figure 5.1: Basic concept of the derivation process

The first step is to determine by what contexts a call to the system is affected. For each system call, all the affected components and the therein affected *SEFFs* have to be found. Lastly, the context of the system call has to be applied to the *SEFF* as an access policy specification.

Additionally, the designed approach should be backwards compatible. This means that it should still be possible to create manual access policy specification on the *SEFF* level. These manually created policies have then to be considered during the execution of the approach.

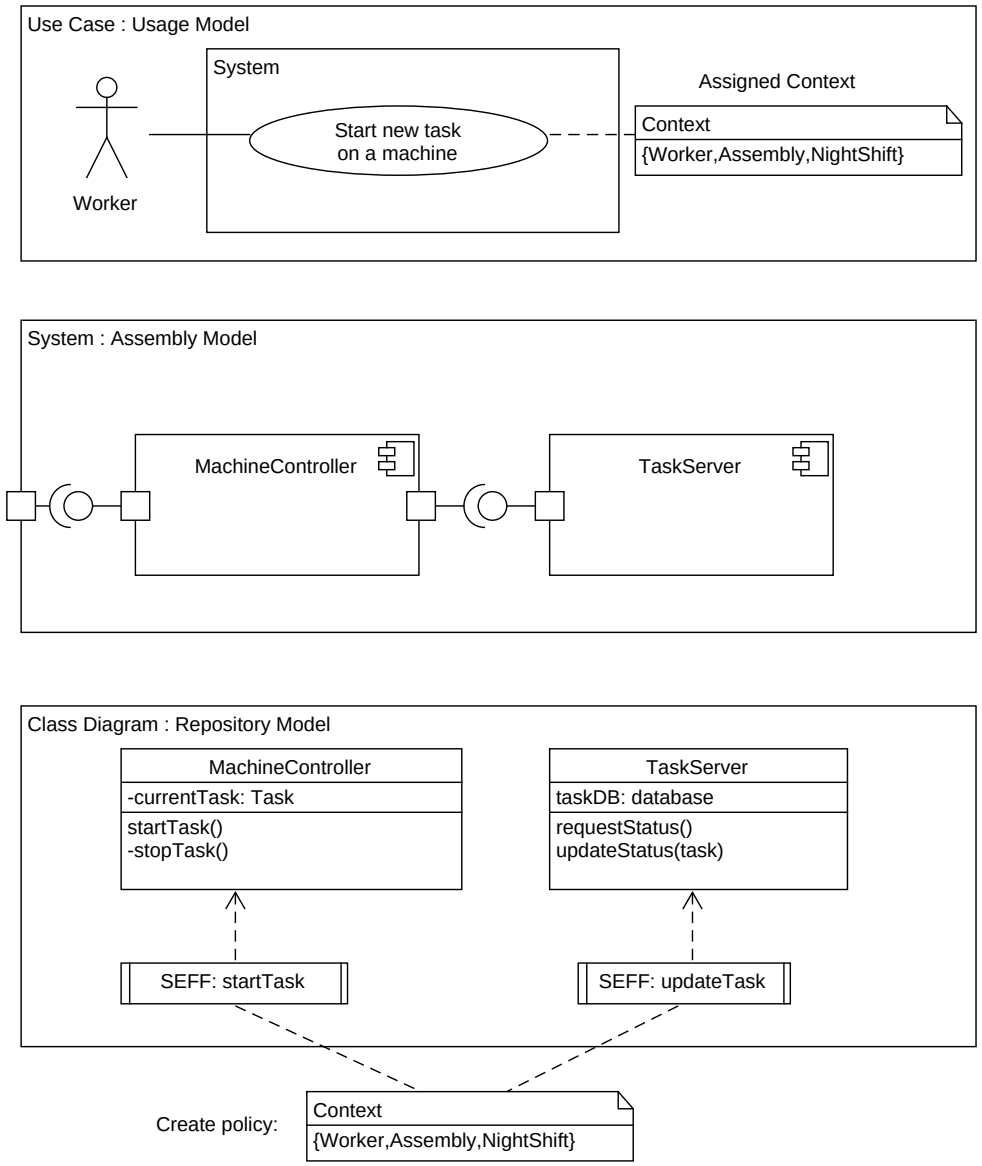


Figure 5.2: View of the different models used in the derivation

5.2 Usage Model Context Set

The usage model is where the contexts which should be derived are defined. Each *EntryLevelSystemCall* represents a call to the system for which the allowed permissions and forbidden constraints can be defined in the form of context sets. The *ContextSpecification* in the context model allows the context set to either be defined for the complete *ScenarioBehaviour* or a single *EntryLevelSystemCall*. This allows for different options by which context sets the system call and, therefore, all the affected *SEFFs* are affected.

- Only the system call is affected: Here, the context sets are defined directly for the specific system call. Since there is no default defined for the scenario, the context sets are directly derived to the *SEFFs*
- Only the usage scenario is affected: In this case, the context is specified for the complete scenario and, therefore, the default for each system call. Since the system call does not specify a derivation, only these context sets are applied to the affected *SEFFs*
- Both are affected: In this case, multiple methods are possible. The scenario context could be seen as a default, and the system call overrides this default context. The scenario context could be seen as the base context, and the system call only specifies an extension of this base context. Alternately, both contexts could be seen as two separate cases which should both be derived.
- No context specified: Since neither the system call has a specific nor the scenario a default context which should be derived, this is treated as undefined context.

5.3 Finding affected Service Effect Specifications

To find the affected *SEFFs*, the chain of function calls has to be followed. The start is the *EntryLevelSystemCall* in the usage diagram. It defines the called interface and method. In the *AssemblyContext* of the system model, the matching *OperationProvidedRole* has to be found. This allows the selection of the first affected component. The corresponding *SEFF* can be selected using the method signature. Each *SEFF* can call other components with a *ExternalSystemCall*.

Figure 5.3 shows this in an example. Here, component A is the first affected component of the system call. The component itself calls two other components, so if one of its *SEFFs* calls an external component, this call could go to either component B or component C. They are both in the same *AssemblyContext* as component A. Component C shows the issues with *CompositeComponents*. Externally, they appear to be like normal components, but contain an assembly context themselves; in this case, component D. An external call for component D does not go to the same assembly, instead the match has to be made with a component in a different assembly, component E.

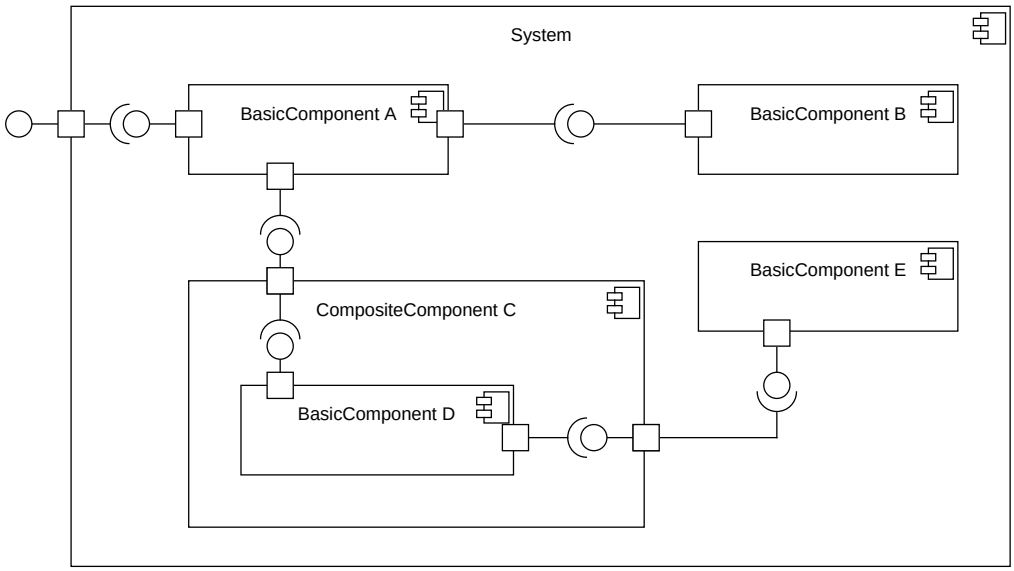


Figure 5.3: System model with nested components

5.4 Creating policies

With the first two steps from the previous sections two lists are generated. The first contains the affected data elements of the user action and the second the context sets which are currently assigned to the user for this request. With this information, the new policies for the data elements are created.

There were two design options for how the new policies should be created. Either a new policy is created for each permutation of the elements in the list, or it is checked if a policy already exists for the current affected data element, and the new context set is added to that policy. The first option has the advantage that it keeps the creation process simple, but on the other hand more policies are created. The second option keeps the number of policies small, but makes the creation of them more complex. For example, since multiple policies could already exist for the affected data element, it must then be decided to which the new context set is added.

The first option was chosen. The creation of multiple *PolicySpecification* makes it easier for the user to see which policies were created, and by which usage diagram. This simplifies debugging. Additionally, it keeps a clear separation between the manually created policies and the policies which are derived and created by the algorithm. The second reason for this option was that it creates a clear separation of concerns between the two parts of this thesis. In the deriving process, only the policies are derived and no combining of any sort should occur. The second part then combines the policies in the context model affecting the same data elements.

6 Combining Policies

The second part of this thesis focuses on the combination and reduction of the derived policies. In the first section the basic concept is explained. The second second explains the different rules defined which can be applied to the context model.

6.1 Rules Combining Concept

During the derivation process it is possible that multiple access policies are derived for the same *SEFF*. These can happen in two possible ways: multiple usage diagrams affect the same system function, or a component is called by multiple other components. If multiple *PolicySpecifications* exist for the same *SEFF*, all of the assigned *ContextSets* grant access to the data element represented by this *SEFF*. The same is true for *PolicySpecifications* which contain multiple context sets. Table 6.1 shows how the resulting access policy is defined in this case.

Assigned:	Set1 = {Worker, Remote} Set2 = {Worker, Assembly}
Access Policy:	Set1 ∨ Set2

Table 6.1: Example for multiple context sets

The first context set grants access to workers who request remote access, the second to workers in the assembly. To be granted access to this object, an employee must fulfill at least one of these conditions. The context set can, therefore, seen as a logical disjunction. Additionally, a less specific policy grants access to a more specific context set, as long as all the contexts are included in in. For the example, the context set *{Worker, Assembly, NightShift}* would also be granted access, since it is included in *Set2*. The same is true for hierarchical contexts. Here, the less specific context set does not have to contain the exact same context, but must only contain a hierarchical element which passes it on to the more specific one. For example, the context set *{Team Leader, Remote, DayShift}* would also be granted access, since all the contexts are contained ("Team Leader" inherits his access rights from "Worker", since it is bottom up).

Since the policies specified in the context model are later used by the access control system to decide if access should be granted or not, the design in our approach is to create a minimal set of policy specifications by combining context sets. The reason for this is that the combination of policies can be done during the development phase of the system. Not combining policies can lead to more policies which need to be checked during runtime.

The table 6.2 shows an example for the combination of rules. Here, *Set1* and *Set2* do not need to be saved, since they are already included in the third context set.

Assigned:	Set1 = {Worker, Assembly, Remote} Set2 = {Worker, Assembly} Set3 = {Worker}
Access Policy:	(Set3)

Table 6.2: Example for combining rules

Since negative contexts from the misusage diagrams are also derived, the priority of the different possible context sets is defined. P (Permit), D (Deny) and IN (Indeterminate) are the possible values a certain context set request can have for a specifc *SEFF*. P is the case if the policy matches the requested context set, D if the requested context set is explicitly forbidden by a negative context set, and IN if the context set does not match the assigned context sets. We define that all undefined access policies should be denied, therefore IN results in a denied request. The priority order we define is: $D > P > IN$. This means a negative context set, derived from a misusage diagram, overrides aa assigned positive context set. And a positive context set overrides the implicit denil of undefined behaviour. Table 6.3 shows an example for this.

Assigned:	Set1 = {Worker, Remote} Set2 = {Worker, Assembly} Set3 = negative, {Technician}
Access Policy:	(Set1 $\vee$ Set2) $\wedge$ $\neg$ Set3

Table 6.3: Example for multiple context sets with negative context set

The third context set is derived from a misusage diagram, forbidding the "Technician" access to this *SEFF*. To be granted access to the object, the user has to fulfill either one of the positive context sets, and he must not fulfill the negative context set. Since the context meta-model does not allow the storage of the negative information of context sets, this information is only available during the execution of the approach. As the IN state is defined as "access denied", the information of *Set3* does not have to be saved in the final context model. This assumption holds true as long as there are no conflicting context sets. Table 6.4 shows this case.

Here, a worker in the office should not have remote access to this data element. At the same time, the context set *Set1* grants access to workers in the office. If here the information of *Set2* would simply be omitted, the created context model would grant access to the contex set {Worker, Office, Remote}. This would result in an access rights violation, since the explicit forbidden case is now allowed. For *Set3* and *Set4*, there is no issue, since after omitting the information of *set3*, *set4* would still have access, and *set3* would be denied access.

Assigned:	Set1 = {Worker, Office} Set2 = negative, {Worker, Office, Remote} Set3 = negative, {Technician} Set4 = {Technician, Remote, Failure}
Access Policy:	error

Table 6.4: Example for error case

6.2 Rule Definitions

For this approach, multiple combination rules have been defined. Since the development process was iterative, rules were implemented consecutively. The next sections give an overview of the defined rules which are present in the current approach.

6.2.1 Same Context Set

Assigned:	Set1 = {Worker, Assembly} Set2 = {Worker, Assembly}
Access Policy:	Set1

Table 6.5: Same context set

This rule is used to combine two context sets which contain the exact same elements. This rule is also needed as a basis for some of the other rules, which, by being applied, result in duplicate context sets.

6.2.2 Simpler Context Set

Assigned:	Set1 = {Worker, Assembly} Set2 = {Worker, Assembly, Remote}
Access Policy:	Set1

Table 6.6: Simpler context set

This rule combines two contexts, for which one includes the other. This rule does not consider inheritance of hierarchical contexts.

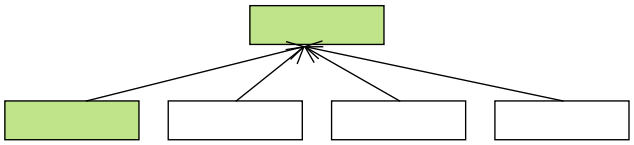
6.2.3 Parent Child Relation

This rule combines context sets which are hierarchically included in the other. Here, the inheritance is considered as shown in figure 6.1. With the direction *BOTTOM_UP*, the child

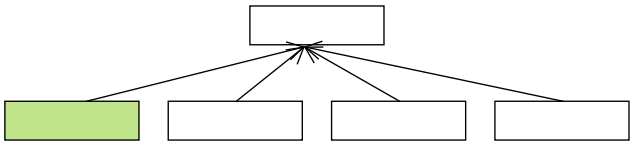
Assigned:	Set1 = {Worker, Assembly} Set2 = {Worker, Building A} Set3 = {Manager, Warehouse} Set4 = {Worker, Warehouse}
Access Policy:	Set2 ∨ Set4

Table 6.7: Hierarchical context set

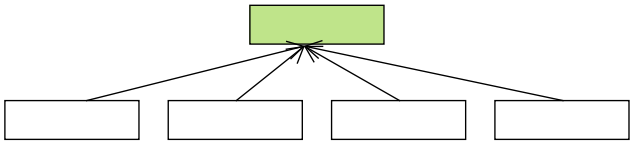
node is the less specific context and therefore includes the parent node. For *TOP_DOWN*, the parent includes the child node, and therefore is less specific than it. As seen with *Set3* in table 6.7, the rule also works for inheritance over multiple hierarchy levels. Here, the manager is inheriting access from the worker context, which is two layers.



(a) Initial applied contexts



(b) Applying rule, direction *BOTTOM_UP*



(c) Applying rule, direction *TOP_DOWN*

Figure 6.1: Parent child rule

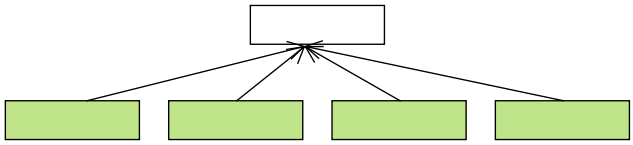
This rule also considers a combination of both directions. Table 6.8 shows an example. Here, *Set1* includes *Set2*, because both of its hierarchical contexts are less specific.

Assigned:	Set1 = {Worker, Building A} Set2 = {Manager, Assembly}
Access Policy:	Set1

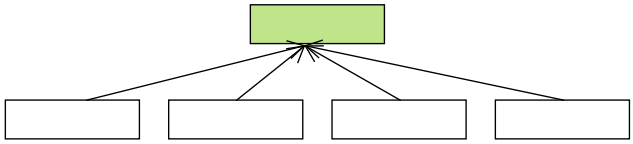
Table 6.8: Parent child both directions

6.2.4 Substituting Parent

The concept of this rule is shown in figure 6.2. If all child elements are allowed to access an *SEFF*, the child elements are replaced and a new context set is created with the parent node.



(a) Initial applied contexts



(b) After applying substitute parent rule

Figure 6.2: Substitute parent rule

Assigned:	Set1 = {Worker, Office} Set2 = {Worker, Assembly}
Created Set:	Set3 = {Worker, Building A}
Access Policy:	Set3

Table 6.9: Substituting child nodes with parent node

This rule is only defined for the direction *TOP_DOWN*. The reason for this is that hierarchical contexts are usually tree graphs. In this case, this rule would always be true, since each node only has a parent node.

For this rule, an additional constraint is in place. If this rule is enabled, it should be ensured that the actual instances of the hierarchical contexts during runtime only contain leaf nodes. Otherwise, for this rule, the definition of IN is not valid anymore, since the context set *{Worker, Building A}* would now have access without explicitly being allowed. This must be considered when enabling this rule in a project.

6.2.5 Negative Rule affecting same context set

Assigned:	Set1 = {Worker, Assembly} Set2 = negative, {Worker, Assembly}
Access Policy:	error

Table 6.10: Same context error

This rule is similar to the same context rule except that one of the two context sets needs to be a negative context.

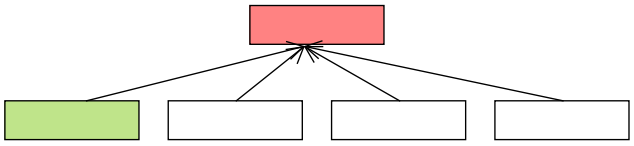
6.2.6 Negative Rule for simpler context set

Assigned:	Set1 = {Worker, Assembly} Set2 = negative, {Worker, Assembly, Remote}
Access Policy:	error

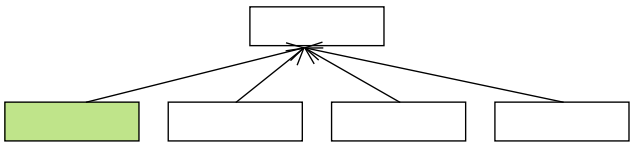
Table 6.11: Simpler context error

This rule is similar to the simpler context rule except that the more specific context needs to be the negative policy.

6.2.7 Negative Rule affecting hierarchical contexts



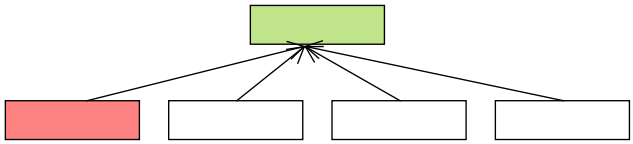
(a) Initial applied contexts



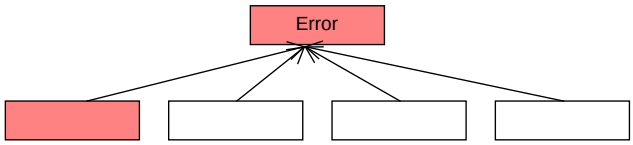
(b) After applying rules

Figure 6.3: Hierarchical context set with negative - allowed case

This rule is similar to the parent child rule, the more specific context needs to be the negative policy.



(a) Initial applied contexts



(b) After applying substitute parent rule

Figure 6.4: Hierarchical context set with negative - error case

Assigned:	Set1 = {Worker, Assembly} Set2 = negative, {Worker, Building A} Set3 = {Worker, Building B} Set4 = negative, {Worker, Warehouse}
Access Policy:	error (Set3 & Set4)

Table 6.12: Hierarchical context set error

6.2.8 Merging Policies affecting the same SEFF

This rule does not combine context sets, it just restructures the existing *PolicySpecifications*. Since a *PolicySpecifications* can have multiple *ContextSets*, this rule merges all context sets to one policy specification. This is mostly a cosmetic rule, since the number of context sets affecting the *SEFF* stays the same but the context model becomes more readable.

6.2.9 Removing temporary negative context sets

As mentioned earlier, the context meta-model does not allow the storage of negative context sets. This rule removes all the temporarily created negative policies from the context model.

6.3 Order of Rules

In this section, the order in which the rules are executed is briefly explained. The resulting policy of an *SEFF* can be seen as the following form:

$$(Set1 \vee Set2 \vee Set3 \vee ...) \wedge \neg NegativeSet \wedge \neg NegativeSet2 \wedge ...$$

All the positive context sets of the *SEFF* build a disjunction, which is in a conjunction with all the negative context sets. Therefore, it was decided to see the disjunction as a separate term and resolve it first. For the implementation this meant that the positive rule definitions are executed first. Combining context sets always leads to the less specific context set remaining. If rules would resolve multiple times, the least specific or in other words, most inclusive context set would remain. Therefore, the design decision was made to execute the rules in a loop. This also meant that the actual order of the individual rules did not matter.

For the conjunction part of the term, a loop of all negative rule definitions is executed. This was also the reason for splitting the rules and their negative counter parts into two separate rule definitions.

There are some rules which have a special purpose and are not executed with the other rules, but at the end of the loop process. These are the cleanup rule (section 6.2.9) and the merge rule (section 6.2.8).

7 Implementation

In this chapter the actual implementation of the introduced concepts is described. At first, the architecture of the created software is shown, which consists of four parts. Then the separate parts are illustrated using pseudo code and class diagrams. In the last section, the test setup is briefly explained. The repository with the code can be found here: [28].

7.1 Architecture

The approach was implemented using Java as Eclipse Plugins. The architecture of the complete approach can be seen in figure 7.1.

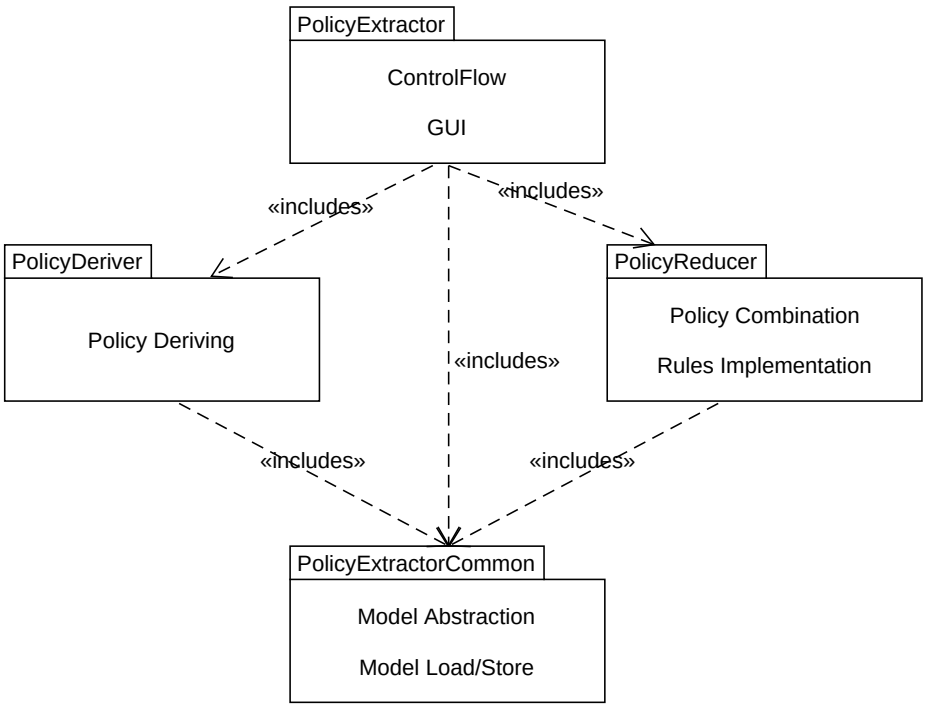


Figure 7.1: Plugin structure

In order to have a separation of concerns [30] between the two parts of the approach, they have been implemented as two separate projects. *PolicyDeriver* implements the deriving part of the approach and in *PolicyReducer* the combination process and the different rules are implemented. The fact that the two parts are split into separate plugins allows them to

be used in possible future work independently of each other in case only one functionality is needed. As input parameters both receive the needed models and the settings affecting certain parts of the behaviour, and they both return the adapted context model.

The loading and saving of the different used models was put into the separate project *PolicyExtractorCommon*. It also provides other basic functions needed in the other plugins and introduces abstraction classes which help encapsulate often needed functions which are performed on the models. To be able to execute the approach as a whole the additional plugin *PolicyExtractor* was created. It controls the interaction between the *PolicyDeriver* and *PolicyReducer* and contains the classes needed to provide a rudimentary GUI for user interaction. For the design of the complete approach the model–view–controller (MVC) [7] software design pattern was used as a guideline.

7.2 Common Functionalities

The component *PolicyExtractorCommon* provides four packages.

- *model*: In this package the functions to load and store the data of the models from files is provided.
- *modelabstraction*: Here the abstraction classes for the different models are provided. An example is getting the *EntryLevelSystemCall* contained in a *ScenarioBehaviour* or creating or removing context sets from the context model. For the context model additional classes are provided. *HierarchicalContextAbstraction* contains all the functions needed for the handling of hierarchical contexts, and *ContextSetRecord* and *ContextSetRecordCompare* can be used for the comparison of context sets in the implementation of the rules.
- *settings*: The settings package provides a class to store the parameters provided by the user via the GUI and pass it on to the other plugins. Additionally, the settings class can be used in the test cases to adjust the behaviour which needs to be tested.
- *util*: This package provides basic helper functions and utility classes such as a logger class.

7.3 Deriving Policies

The *PolicyDeriver* provides the plugin for deriving the context sets. The constructor takes PCM models, context model and user settings as input parameters. The interface for this plugin allows for the execution of the algorithm and to get the new context model with the derived policies. Algorithm 1 describes the implementation of the concepts described in 5.2. All logic impacting the PCM models was again encapsulated in a separate class *PalladioAbstraction*. This allowed the function to be keep as simple as possible and as close as possible to the activity diagram defined in the concept. It could also help to easily extend the approach to other meta models than the PCM in the future.

Algorithm 1 Deriving Contexts in PCM

```
1: for scenario = 1, 2, ... do
2:     for systemcall = 1, 2, ..., N do
3:         Calculate affected seffs by this systemcall
4:         for seff = 1, 2, ..., N do
5:             Calculate the contextsets which need to be applied
6:             for deriverRecord = 1, 2, ..., N do
7:                 Label: Apply context to seff
8:                 Create policy specification
9:                 Set values accordingly
10:                Add policy to model
11:            end for
12:        end for
13:    end for
14: end for
```

Algorithm 2 Find affected seffs for the current systemcall

```
1: Initialize list of affected seffs
2:
3: Label: FindMatchingComponent
4: Find match between interface and component in assembly
5: Label: ComponentMatch
6: if BasicComponent then
7:     Apply context
8:     if exists ExternalCall then
9:         for externalAction = 1, 2, ..., N do
10:            Label: FindExternalComponent
11:            if Find matching component in current assembly then
12:                Goto ComponentMatch
13:            else
14:                Search for match 1 recursion level higher
15:                Goto FindExternalComponent
16:            end if
17:        end for
18:    end if
19: end if
20: if Composed Structure then
21:    Recursive call, but with new interface and assembly
22:    Goto FindMatchingComponent
23: end if
24:
25: return list
```

Algorithm 2 describes the implementation of how the affected *SEFFs* are found, beginning from the *EntryLevelSystemCall*. At first, the matching component is attempted to be found in the current assembly. For the initial call of this function the assembly is the system. If the matching component is found, two cases are possible.

If it is a *BasicComponent* the *SEFF* matching the called operation signature has to be found and it can be added to the list of affected *SEFF*. If in the *SEFF* calls to other components are made, the matching component has to be found for that call. Here parameters like signature of the called function and the current components are used to find the matching component in the current assembly. If there is no matching component, the call goes to a component outside of the current assembly. Either outside of the complete system or, if the current assembly is inside a *ComposedStructure*, the call could go to a component in the assembly one level higher. The parameters are then updated and the function is called recursively for the assembly a level higher. Here the recursive depth is limited to the depth of nested *AssemblyContexts*.

If a matching component is a *ComposedStructure* for the current assembly, the matching component has been found. But since the *ComposedStructure* has an assembly inside it, the affected components in that assembly have to be found. This is done with a recursive call to the function. In each recursive call the hierarchy of nested assemblies is updated and passed as a parameter. Therefore, if external calls in nested components have to be found, the nested levels can be returned correctly. *ComposedStructure* themselves can not have *SEFF*, only *BasicComponents*.

7.4 Calculate context set to apply

Algorithm 3 describes the way the context sets affecting a system call are calculated from its assigned context sets as designed in section 5.2. The cases for which either only the *EntryLevelSystemCall* or the *UsageScenario* are affected are fairly straight forward as is the case for which the context is undefined. In the case for which both have context sets assigned, the behaviour is switched with the settings introduced in section 7.6.

For the case that the context sets should be combined, a new context set is created which contains the contexts of both context sets. A *ContextSpecification* can only have one *ContextSet* assigned to it, but since multiple specifications can affect the same system call or scenario, the combined contexts have to be generated as permutations. Each system call context set has to be combined with all scenario context sets and vice versa.

Algorithm 3 Calculating the context set which needs to be applied

```
1: Initialize recordlist as list of DeriverRecords
2: Set calllist = list of contextsets from systemcall
3: Set behaviourlist = of contextsets from behaviour
4:
5: if calllist is empty then
6:     Create records from behaviourlist
7:     Return recordlist
8: end if
9: if behaviourlist is empty then
10:    Create records from calllist
11:    Return recordlist
12: end if
13: if Settings.combineContextSets is enabled then
14:    Create records from calllist
15:    Return recordlist
16: else
17:    SystemCall contexts have priority, use them
18:    if systemcall is misusage then
19:        Still use behaviour context
20:    end if
21: end if
22:
23: Return recordlist
```

In case of no combination, there are again two options. Either the context sets have the same priority or the system call context set overrides the default context of the scenario. In case of same priority, the context sets lists can simply be merged. In case the system calls have higher priority, two scenarios have to be considered. In case all the system call context sets are misusage cases, the default context of the scenario must still be applied. Possible misusage cases defined for the complete scenario have to applied as well even though the system calls have priority.

For each calculated context set which affects the system call a *DeriverRecord* is created. The record is needed since for the *PolicySpecifications* created for the affected *SEFFs*, not only the information of the context set is needed, but also the information whether the created policy is negative. Additionally, the information about which system call and scenario affect this *SEFF* with this context set is saved. This information is required if an error case is detected during the combination process in order to generate a detailed error description. For the name generation of the generated policy this information is also used.

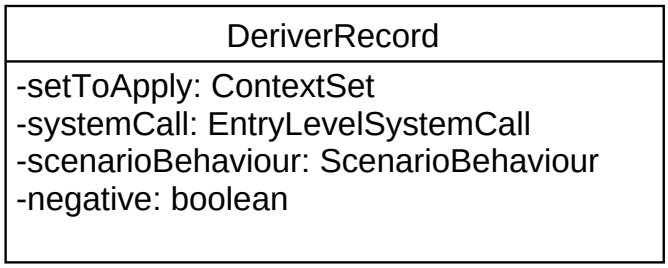


Figure 7.2: DeriverRecord class

7.5 Applying Rules

The implementation of the concepts illustrated in chapter 6 is shown here. As described above, this part was implemented as a stand alone plugin to have a strong separation of concerns. The main class of the plugin is *PolicyReducer*, shown in the class diagram 7.3. For the initialization, the context model and the user settings are needed. The context model is not passed directly to it; instead the abstraction class introduced in section 7.2 is used. This is needed since it contains the information about the negative policies, which cannot be stored in the native context model. The *PolicyReducer* contains a list of rules which can be applied to the model. With the settings passed to it, different rules can be enabled or disabled. The class has a main function called *execute* which will run the plugin. The pseudo code for this function is given in algorithm 4. The function iterates over the enabled rules and tries to apply them to the context model. The function contains a while loop which will be exited if one iteration doesn't result in any changes to the context model. This concept is similar to a *fixed-point iteration*. Since each rule could change the existing context model and create or remove policies, only one iteration over all rules would not be sufficient. Here it is necessary to ensure that two policies do not create a pattern which results in an endless loop. This could happen if the output of the first rule would match as input of the second and vice versa. This was attempted by designing the rules properly and by testing for this behaviour. Nevertheless, a hard coded condition to end the loop was added to ensure the program does not crash. In this case, an error is thrown.

The benefit is that the actual order of the rules does not impact the result of the generated context model. Also, the rules could be designed in a way to keep each rule relatively simple, relying instead on the other rules applied in the next iteration. If this weren't the case, each rule would need to be aware of the order of all rules and which steps have already happened before its execution. This would have been hard to coordinate with the option of disabling certain rules.

Algorithm 4 Apply rules to context model

```
while changed do
    Initialize the rules which should be applied
    Label: Apply rules to model
    for rules = 1, 2, ..., N do
        for seff = 1, 2, ..., N do
            Label: Apply specific rule
            Compare each context with others affecting this seff
            for contextSet_base = 1, 2, ..., N do
                for contextSet_compare = 1, 2, ..., N do
                    if rule can be applied then
                        Create rules record with all needed information
                    end if
                end for
            end for
        end for
    end for
    Label: Execute Rules
    for rules = 1, 2, ..., N do
        for rulesRecord = 1, 2, ..., N do
            if new context created needs to be created then
                Create context set according to record
                Add context set to model
                Add context set to policy
            end if
            Remove context set from policy
        end for
    end for
    if No rule was applied then
        Exit while
    end if
end while
Cleanup context model
```

The class diagram 7.3 also shows this general setup of the rules. The *PolicyReducer* only has a list of *IRulesDefinitions*, which provides two methods: one for applying the rule to the model, and the second to execute the rule. These two methods are represented in algorithm 4 by the labels "Apply rules to model" and "Execute Rules". In the first method, the rule will iterate over the context model and try to apply itself to it. A rule works on all context sets for one *SEFF* and compares them to each other. If the rule can be applied to a context set, the model will not be changed immediately but instead a *RulesRecord* is created which contains all relevant information. For example, which rule was applied,

which *SEFF* was affected, if a context set will be removed or a new one created. The second method will then execute the rule for each record set created.

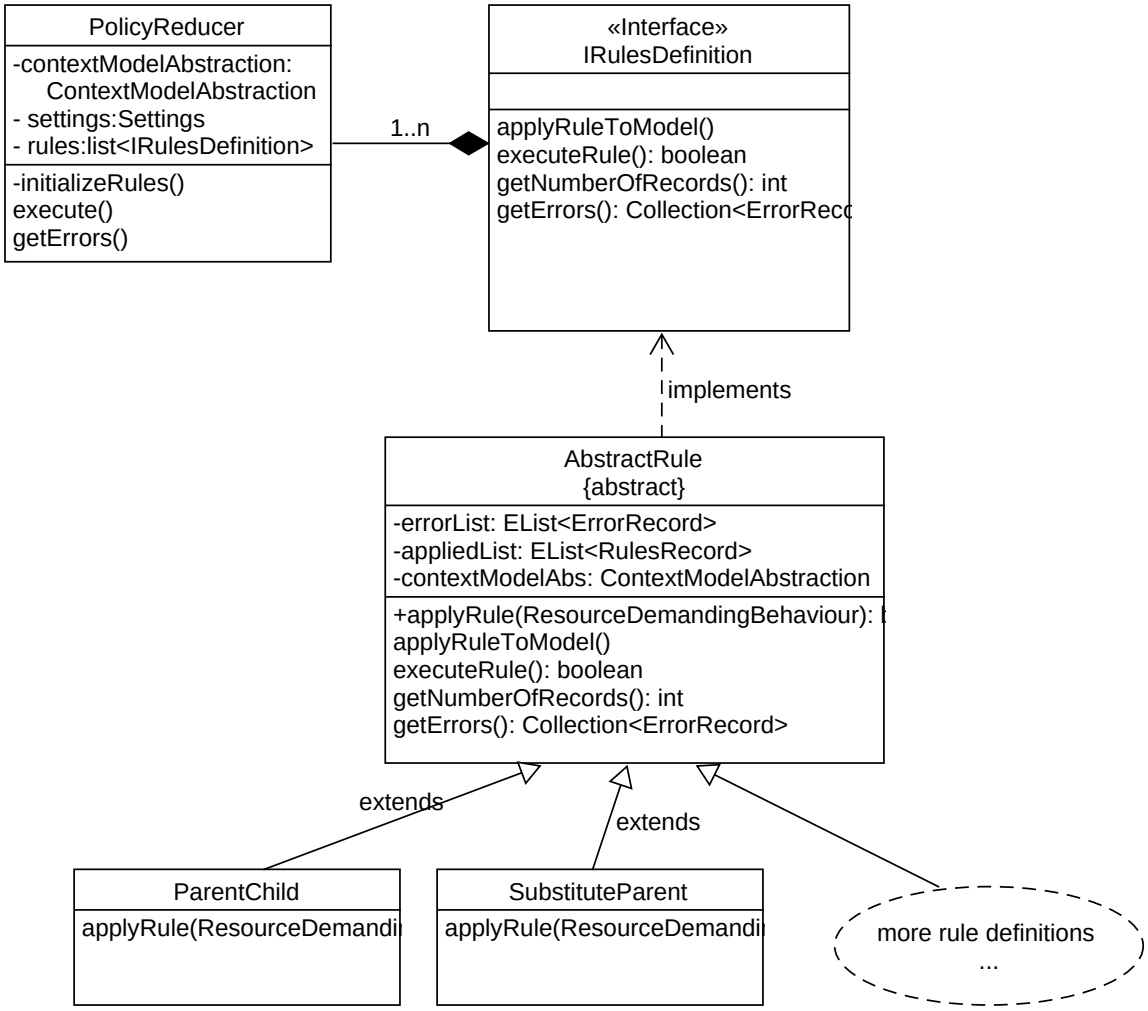


Figure 7.3: Class structure for rules

The separation of these functions was done for multiple reasons. The first was that some rules delete context sets which would lead to a change in the list over which is iterated, which is not supported by the implementation language. Another reason was during the design phase, it was assumed that if two rules affected the same context sets during the same iteration, the resulting *RulesRecords* have to be resolved of a possible conflict before the execution. This was not the case. The third reason was that since the *executeRule* method works on records which contain all the required information, this method only needed to be implemented once. This was done in the *AbstractRule*. It contains all functions which are the same for all rules and only leaves the abstract function *applyRule(ResourceDemandingBehaviour)* for the specific rule classes to be implemented, which is done according to the described rules in section 6.2. In order to reduce code

duplication the classes *ContextSetRecord* and *ContextSetCompare* were created to capsulate common functionality.

In chapter 6 the order of positive and negative rules was explained. The implementation of this lead to two loops of the while loop shown in algorithm 4: the first for the positive rules and the second for the negative rules. As the final step, a cleanup step is executed. Here, the rules *NegativeCleanup* and *MergeSEFF* are run. They are removed from the normal iteration loop and run once since they only have to be applied once per design.

7.6 Executing the program

The two parts were implemented as separate plugins as mentioned at the beginning of this chapter. The fourth plugin developed as part of this thesis uses these two parts and executes them in an ordered manner. Additionally, a rudimentary GUI is provided. It is not state of the art, as it was not the focus of this thesis, and rather only enables the user to vary the input parameters for the software.

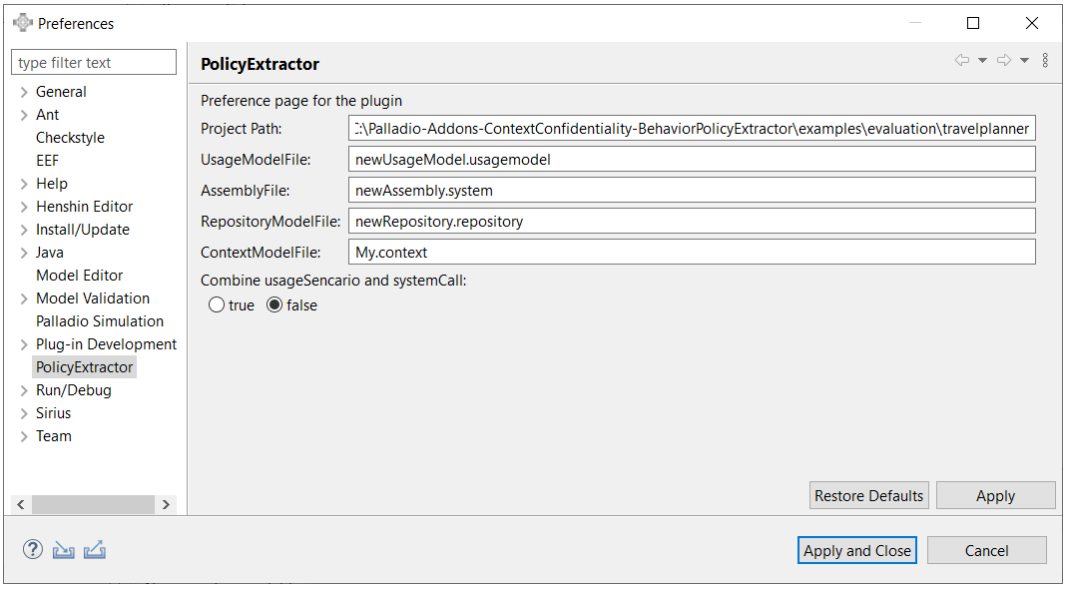


Figure 7.4: Setting the parameters for the program in the GUI

With these settings, the behaviour of different parts of the implementation can be adapted to the user's needs. These settings also enable the test setup mentioned in the next section since it allows the parameters for the test execution to be adapted accordingly. Figure 7.4 shows the settings which can be adjusted by the user.

7.7 Test Setup

During the development of the software, the tests were developed in parallel. For the creation of the tests JUnit was used. JUnit is a unit testing framework for the Java programming language. The development was done with Java version 11.0.8.

Similar to the architecture of the four implemented software plugins, four test projects were implemented. Each of the test projects uses unit-tests to test the individual classes of the component under test. For the *PolicyDeriver* and the *PolicyReducer* additional system-tests have been created in which the functionality of the component as a whole is tested. In the test project of the *PolicyExtractor*, tests for the interaction between the two components are placed. Additionally, the functionality for the measurements of the metrics used in the evaluation have been implemented as test classes. This allows for reproducible measurements and results.

During the development process, continuous integration (CI) [12] was used. This helped to identify software regressions introduced by changes in the source code, and the generation of test coverage of the code ensured that the code had been sufficiently tested. With 62 test cases a line coverage of 84% is achieved. Some of the uncovered lines are part of the GUI, which is not executed during the test phase.

8 Evaluation

In this chapter, the proposed solution and the designed approach are evaluated with four case studies. The first section describes the method which was used to evaluate this thesis. The used case studies are then introduced. Each of the three defined goals which have been evaluated has a separate section explaining the test setup and the results. Subsequently, the threads to validity are explained and limitations and assumptions are shown. Finally, the different results are summarized in a short recap.

8.1 QGM Plan

To evaluate this thesis we used the Goal-Question-Metric (GQM) [2] method which is a goal oriented approach. For the QGM-method, specific goals are defined which the designed approach should fulfill. Specifying these goals helps to collect relevant data during the evaluation. After establishing the goals, they are used to develop several questions which should be answered by the case study. In general, a goal will result in the generation of multiple questions. Finally, the metrics are defined with which the questions can be examined on a quantitative basis. In table 8.1, the goals we defined for this thesis can be found.

Goal 1 : Accuracy	
Purpose Focus Process Stakeholder	Evaluate the correctness for deriving policies from the security point of view
Goal 2 : Effort reduction	
Purpose Focus Process Stakeholder	Improve the efficiency of policy creation from the developer point of view
Goal 3 : Scalability	
Purpose Focus Process Stakeholder	Analyze the performance of deriving policies from the developer point of view

Table 8.1: Goals defined for our approach

The first thing which we wanted to achieve with our approach was the possibility of deriving the correct access control policies from the usage and misusage diagrams. In section 3.5, the two main premises of access control systems are described which should not be violated by the created approach. Therefore, this goal can be summarized as wanting to achieve high accuracy while applying the proposed solution. The second goal we wanted to accomplish was an improvement in efficiency for a developer when creating policies for an access control system during the design phase. A developer creating access control policies should have a reduced effort with the proposed solution compared to manual creation. The third goal is to analyze the performance of this new approach. This goal is less important than the first two goals, since for the theoretical evaluation of the approach as a whole, the performance of the program is secondary. But in case the approach is going to be used in a real environment, it is useful to know how the performance is since it has a big impact on usability [20].

Goal 1	Accuracy
Question 1.1	Are the correct elements affected while deriving the policies?
Question 1.2	Are the correct elements affected while combining the policies?
Question 1.3	Are all erroneous elements found?
Metric 1.1	Precision
Metric 1.2	Recall
Goal 2	Effort reduction
Question 2.1	How high is the effort using this approach?
Metric 2.1	Ratio of number of policies between both approaches
Goal 3	Scalability
Question 3.1	How does the runtime scale with certain parameters?
Metric 3.1	Runtime

Table 8.2: Goals, questions and metrics used for the GQM-method

The defined goals and the questions and measurements derived from them are shown in table 8.2. For goal G1 three questions arise. The first is how accurate is the algorithm regarding the derivation of the policies from the usage diagrams to the *SEFFs*. The second is how accurate is the combination and reduction of the context sets in the second step. The third is that if, in case there is an erroneous configuration as described in chapter 6, to see if all possible error cases are detected. All these questions are affected by the quality aspect *Accuracy*. The two metrics M1.1 and M1.2 are Precision and Recall.

Definition 1

$$P = \frac{TP}{TP + FP}$$

Definition 2

$$R = \frac{TP}{TP + FN}$$

For Precision P, true positives (TP) and false positives (FP) are used. For Recall R, true positives (TP) and false negatives (FN) are used. The definitions of TP, FP and FN depend on the question being answered. The evaluation and the results for this goal are shown in section 8.3.

The second goal aims at increasing the efficiency. The question derived from this goal is to evaluate if there is an improvement in efficiency in regards to time and, if so, how great is the improvement. Additionally, an insight into which parameters might influence the effort for access policy creation could be helpful for future research projects. The metric used for answering this question is the number of policies which need to be created with the help of the new approach compared to the basic approach of using only the context model.

Definition 3

$$EffortReduction = 1 - \frac{number\ of\ policies\ new\ approach}{number\ of\ policies\ base\ approach}$$

Section 8.4 analyses this question and shows the results regarding it. With the third goal, the performance of the developed approach is inspected. The question derived is how well the program scales in regard to specific parameters. The metric used to answer this question is the runtime needed to execute the program with the input varied at the specific parameter. In section 8.5, this aspect of the developed approach is evaluated.

8.2 Case studies

For the evaluation of the defined goals different case studies are used. Each of the case studies has been used in one or more evaluations for new approaches regarding access control systems and security rules [24] [40] [11]. To the knowledge of the author of this thesis, no case study which contains both a security description based on contexts, and a system architecture, which can be used to model the system inside PCM, is available. Therefore, these case studies have been selected so that at least one aspect, the PCM models, is available and some information about the intended security aspects, although not as a context model, is available. Their description of the expected use cases of the system was used to create a context model based on their security aspects. The created context models were made to be as fitting as possible to the described use cases of the case studies. The different studies were chosen to cover different aspects in regards to their PCM-models.

	SMSApp	DistanceTracker	TravelPlanner	EnergyScenario
Interfaces	2	2	8	6
RepositoryComponents	3	2	4	6
AssemblyContexts	3	2	4	8
SEFFs	6	6	10	11
Usagemodels	4	1	1	6

Table 8.3: Comparison of case studies regarding PCM elements

Table 8.3 shows these different aspects. For example, the *DistanceTracker* and the *TravelPlanner* case study only contain one usage diagram, the *TravelPlanner* being the more complex one in regards to repository and system. The *SMSApp* represents a well rounded project, and the *EnergyScenario* case study describes the most complex system of these four with a use case in Industry 4.0. In the following, the four case studies are shortly introduced and the usage scenarios and security model for each is briefly explained. The used models can be found here: [28].

8.2.1 ContactSMSManager

The application *ContactSMSManager* [24] consists of two mobile apps: The *ContactManager* for managing contact data, which can be used to create new contacts, view existing contacts or delete them. Additionally, an SMS can be sent to a selected contact. For this functionality the *SMSManager* is used, which provides the function to send the message to a selected number. The security concern with this case study is that only the user should be able to access the contact information from the *ContactManager*, and that the *SMSManager* tasked with sending the SMS should only have access to the number of the receiver, but not the other information connected with that contact. The purpose is to show that the *ContactSMSManager* doesn't leak the personal contacts of the user. For this purpose, the *removeName* is provided inside the *ContactManager* to ensure that only the information which is allowed to be declassified from a contact is provided to the *SMSManager*.

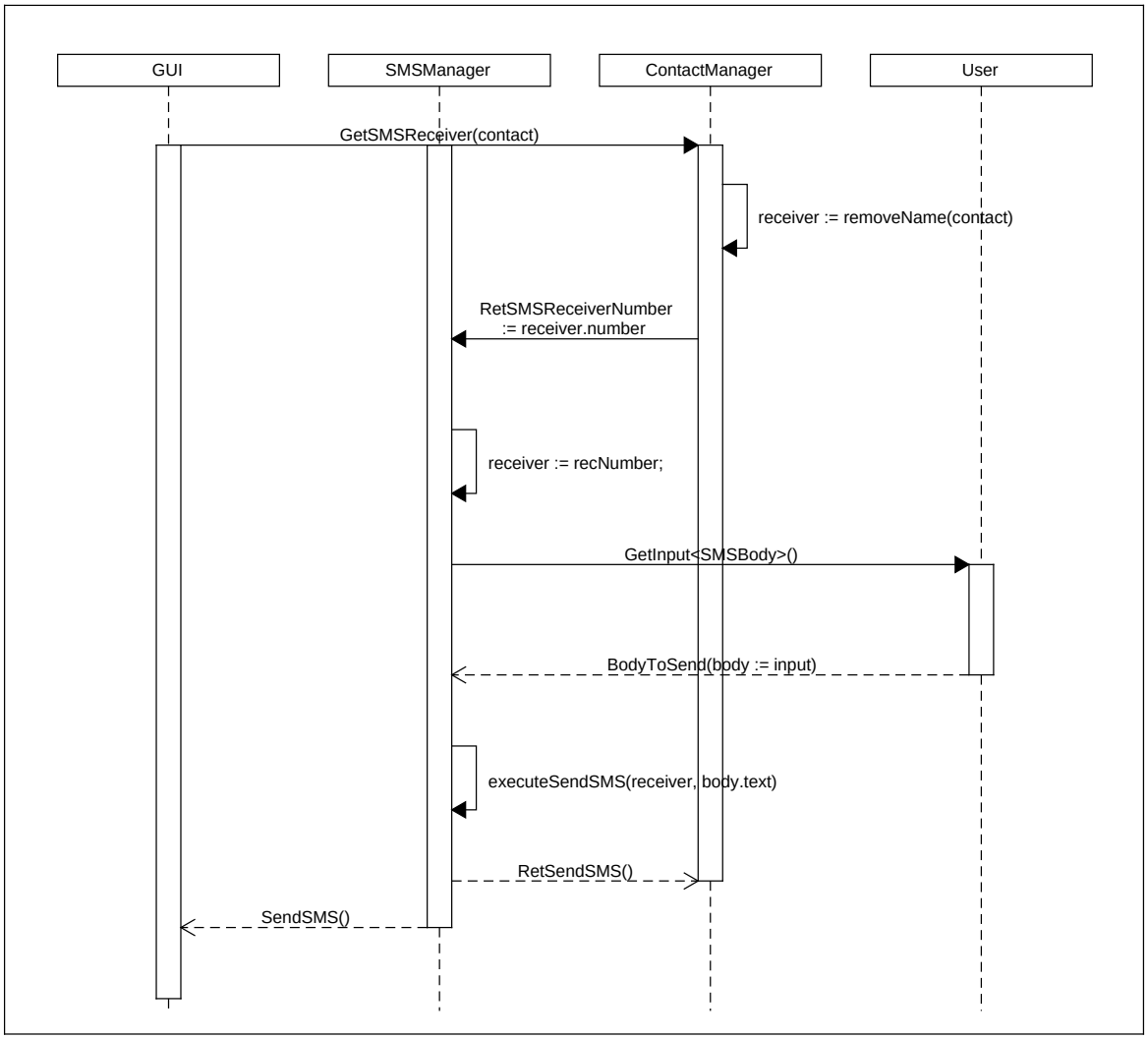


Figure 8.1: Main sequence diagram of the *ContactSMSManager* app [24]

The case study contains four *UsageModels*. Three of them model the different actions the user can do to edit his contacts : *AddContact*, *DeleteContact* and *ListContacts*. The fourth is the usage diagram for the sending of an SMS. This behaviour is shown in figure 8.1. It describes the interaction between the different components in this scenario. The *GUI* is the view component in the application, coordinating the interactions between the different modules and the user. The action to send an SMS has happened before the start of this sequence diagram and the contact to which the SMS should be sent is selected. This triggers the *GUI* to receive the information for this contact from the *ContactManager*. Before passing the receiver information to the *SMSManager*, the name of the contact is removed with the help of the function mentioned above. The user is then asked for the input which will be the content of the SMS. Afterwards, the SMS will be sent by the *SMSManager* to the receiver.

ContextName	Type	ContextValues
Actor	Single	User, GUI
Contact Information Access Level	Hierarchical (down)	complete ↓ declassified

Table 8.4: Context model for case study *ContactSMSManager*

For the evaluation, these scenarios were used to create the context model shown in table 8.4. The context *Actor* is indicating which entity is executing the system function. The two possible values are the *User*, or the *ContactSMSManager* itself, represented by the context *GUI*. The *Context Information* context represents the amount of information a certain request wants to access. It is a hierarchical context, since the context with the higher level of access rights *Complete* should also be able to access the lower ones, in this case the declassified ones. For this case study, four scenarios are created. *S1* contains the normal, good use case of the system while the second also contains misusage parameters explicitly forbidding the access to not declassified information for all requests not coming from the user. In addition to these two positive cases, two negative scenarios are defined which should result in errors being found during the execution of the program. The first is a a result of the fact that both the user and the *GUI* need access to the *getContactList* function. If the misusage scenario for the *sendSMS* only forbids access to a context *complete* instead of the context set *{complete,GUI}*, the access will also be blocked for the user, who wants to access it with the context set *{complete,user}*. The second represents a misconfiguration by the *GUI* trying to call the *deleteContact* function, which should not be allowed.

8.2.2 Distance Tracker

The *DistanceTracker* is a jogging app developed with the *IFlow* approach [24]. The user can use the mobile application *DistanceTracker* to start GPS tracking prior to running and the app will record the current position periodically during the run. Once the user stops tracking, the list of recorded GPS positions will be used to calculate a route which will then

be sent to the web service *TrackerService*. The web service can be used to communicate with other users and compare the tracked distances. From a security perspective, the app shall only be allowed to track the current position if started, the web service shall not receive the position of the user, only the calculated distance, and the app shall only have access to the current location of the current run and not store GPS-positions from earlier runs.

The main use case of this system is described in the sequence diagram 8.2. The *User* is greeted by a welcome message at the start of the app. Then the user is able to start the tracking of the run with *StartTracking* and stop it with *StopTracking*. Internally, the *DistanceTracker* stores the information if a tracking is currently in progress or not in a state flag. After the user stops the current run, the run distance is calculated. The user then has the option to release the information to the *TrackerService*. Only if he confirms the release should the information be sent to the server, otherwise the information should be deleted.

The context model for this case study is shown table 8.5. The context *Actor* distinguishes between the user of the app and the app itself. The *Distance Data* is a hierarchical context with a depth of 3. It describes in which format the data is requested. *raw* contains the complete GPS information, *calcualted* is the distance contained internally by the *DistanceTracker*, and *declassified* is the distance data after being released by the user to the *TrackerService*. For this case study again two positive scenarios are created, one with (*S2*) and one without (*S1*) the misusage information. The scenario *S3* describes a wrong configuration of the system for which the app tries to declassify the distance data by itself, which is forbidden by a misusage diagram. *S4* is a scenario in which the data which should be sent to the *TrackerService* is the raw data and not the distance which was released by the user. The last scenario describes a misusage in which the data which should be released by the user is not the calculated distance but instead still the raw data.

ContextName	Type	ContextValues
Actor	Single	User, App
Distance Data	Hierarchical (up)	raw ↑ calculated ↑ declassified

Table 8.5: Context model of *Distrance Tracker* case study

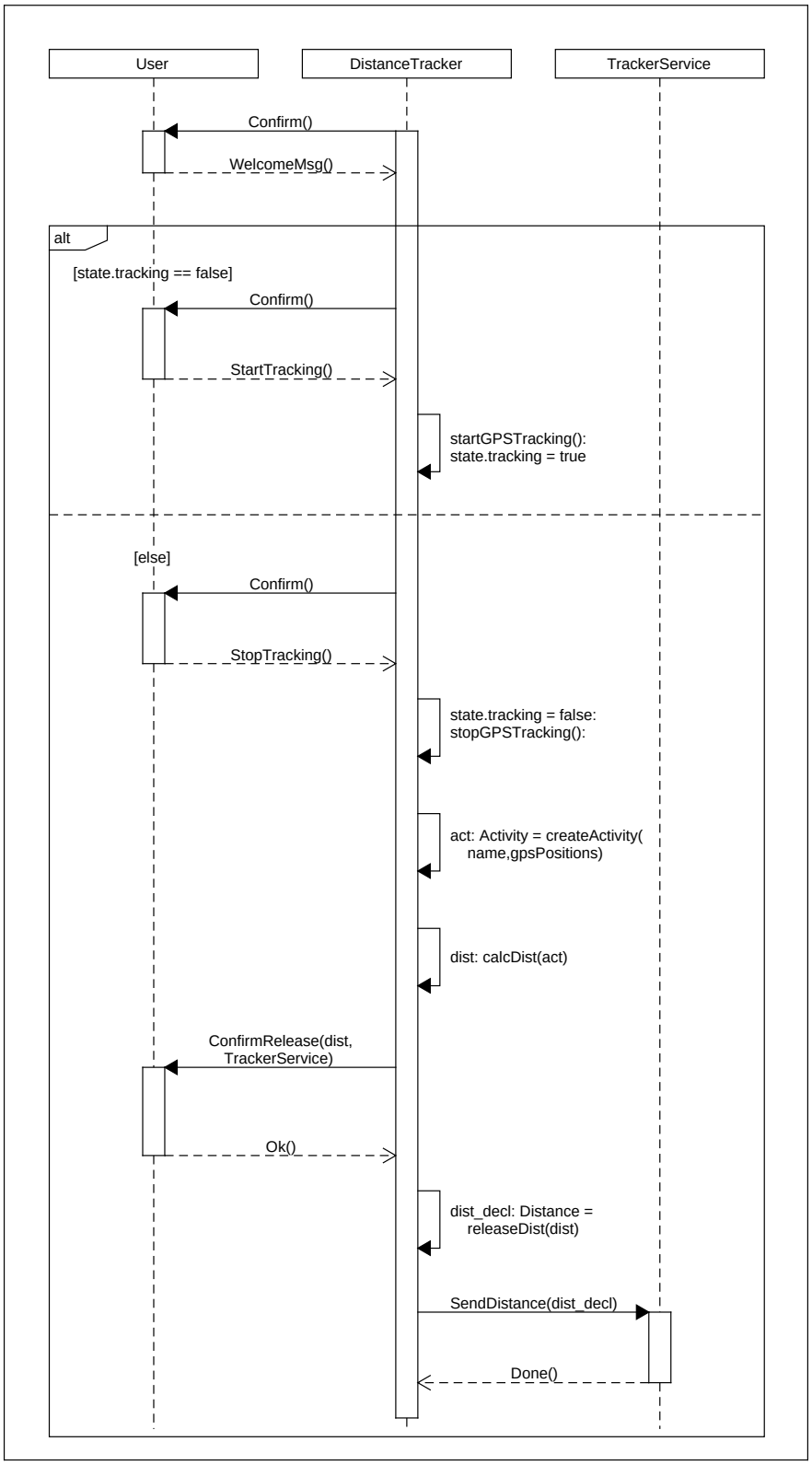


Figure 8.2: Main sequence diagram of the *DistanceTracker* app [24]

8.2.3 Travelplanner

The case study *TravelPlanner* [40] is a distributed application which allows a user to find relevant flight offers for his trip and book them. The *TravelPlanner* app can be run on mobile devices which allows the users to check the *TravelAgency* web service for available and suitable flight offers. If the user wants to book one of the proposed offers, he can select the chosen offer and book it directly over the web service of the *Airline*. All the payments carried out in this scenario are done with the help of a *CreditCardCenter* app. This app contains the user's credit card information, for which the user has to verify himself. Additionally, he can release his credit card data to the airline in order for them to collect the money if a flight is booked. In this case, the airline also informs the travel agency about the successful booking and pays them a commission.

The sequence diagram in figure 8.3 shows the main use case of this case study. All the actions are triggered by the user. The initial action is requesting the current flight offers according to the search parameters. This request is started by the user and then propagated from the user to the *TravelPlanner*, *TravelAgency* and then the *Airline*. To book a selected flight, the credit card data has to be released to the *TravelAgency* first. This is one of the misusage scenarios for this case study. In order to release the credit card data, the user has to be authorized first. This is a second misusage scenario. The two policies needed to model the access rights are that releasing the credit card information is only allowed after the user has authorized himself and that booking the flight is only allowed to users who have released their credit card information. From the description of this use case the context model is defined. Table 8.6 shows the created context model. *CreditCard-Status* is a *HierarchicalContext*, which inherits its access rights bottom up. The two values for this context are either *authorized* or *non-authorized*. Both the context *TravelAgency-CreditCard-Status* and *App-Status* are *SingleContexts*. The first one indicates if the user has already released his credit card information, the second contains the information about whether the user has already requested the current flight offers or if he has already booked a flight and received a confirmation.

ContextName	Type	ContextValues
CreditCard-Status	Hierarchical (up)	authorized ↑ non-authorized
TravelAgency-Status	Hierarchical (down)	locked ↓ declassified
App-Status	Single	not requested, requested, confirmed

Table 8.6: Context model of travelplanner

For this case study, five scenarios are created. *S1* is the normal use case, described in sequence diagram 8.3. The second also describes the positive use of the system, but explicitly contains the described forbidden use cases as misusage diagrams. In addition to the two positive cases, three negative scenarios are defined which should result in

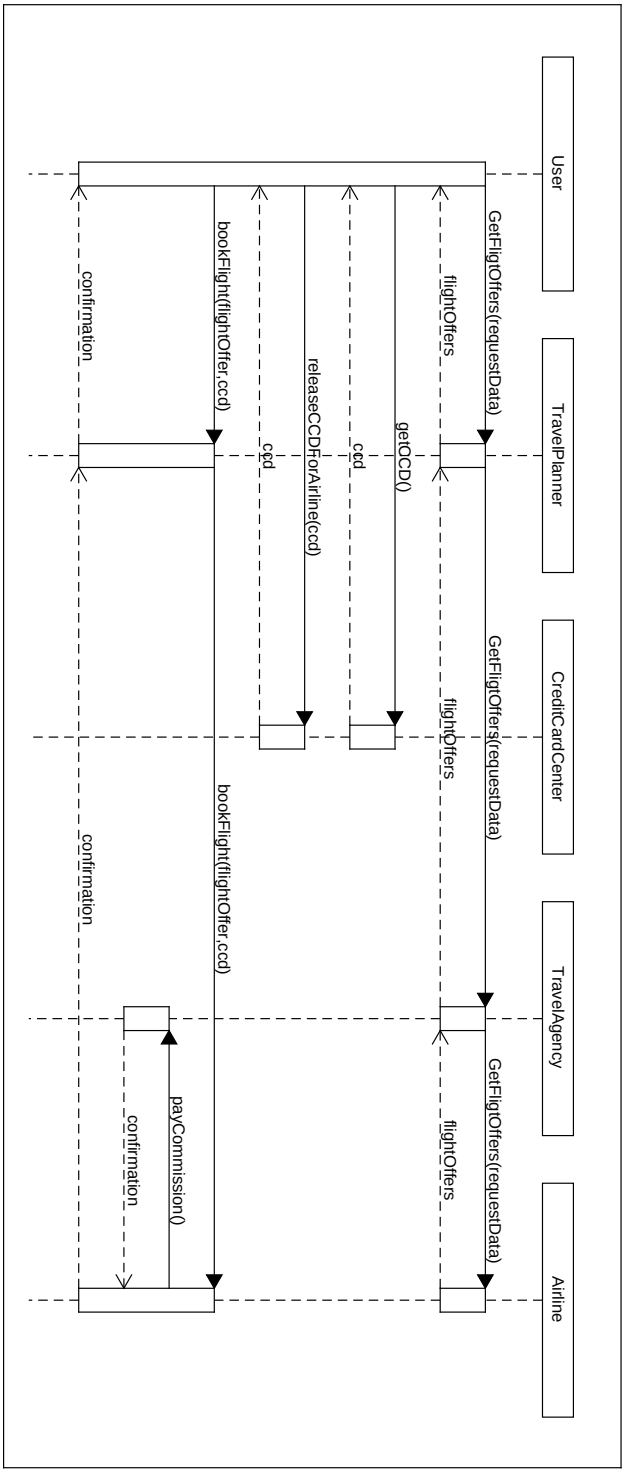


Figure 8.3: Sequence diagram for flight booking [40]

errors being found during the execution of the program. The first misusage scenario handles the case of the user defining the initial context set for the complete usage scenario as a default value, and not specifying a more detailed context set for the operations which need to be secured. The default value clashes with the forbidden context sets of the misusage diagrams. The second and third misusage scenarios handle the different cases of hierarchical misusage. The first ensures that the inherited access rights from the *CreditCard-Status* do not lead to the access for unwanted context sets, and the second one for the *TravelAgency-Status*.

8.2.4 Energy Scenario

The case study *Energy Scenario* describes an energy management system (EnMS) in the TRUST 4.0 environment [11]. Information about the flow of energy contains a high level of information about business activities which allow conclusions about expenses and workflow of the business. Therefore, this information has to be protected by some kind of access control. Figure 8.4 shows the main scenario of this case study. The system has two different kinds of sensor. The *PushingSensor* periodically pushes its data to the server, the data of the *PullingSensor* has to requested. The *EnerChart* collects the data and periodically sends it to the *OPCUAServer*. The different users have access to the data through interfaces provided by the *Trust40* platform, which periodically polls the server. Each of the parts of the system which need to store data have a separate instance of a time series database (*TimeSeriesDB)* to save the data. The system has 3 defined use cases. The energy officer should always be allowed to request the data of the EnMS. For company internal expense calculations the EnMS should allow the request for the aggregated information for a defined time period. In the third use case, an external service company should be granted temporary access to the data of the EnMS to conduct maintenance.

ContextName	Type	ContextValues
Employment	Hierarchical (up)	internal ↑ external
Role	Hierarchical (up)	expert ↑ normal
Location	Hierarchical (down)	remote ↓ on site
Reading method	Single	manual, periodic

Table 8.7: Context model of energyscenario

The derived context model for this case study can be found in the table 8.7. The context *Employment* describes if the user is employed by the company or is from a hired external service company. The context *Role* is used to distinguish between a normal employee

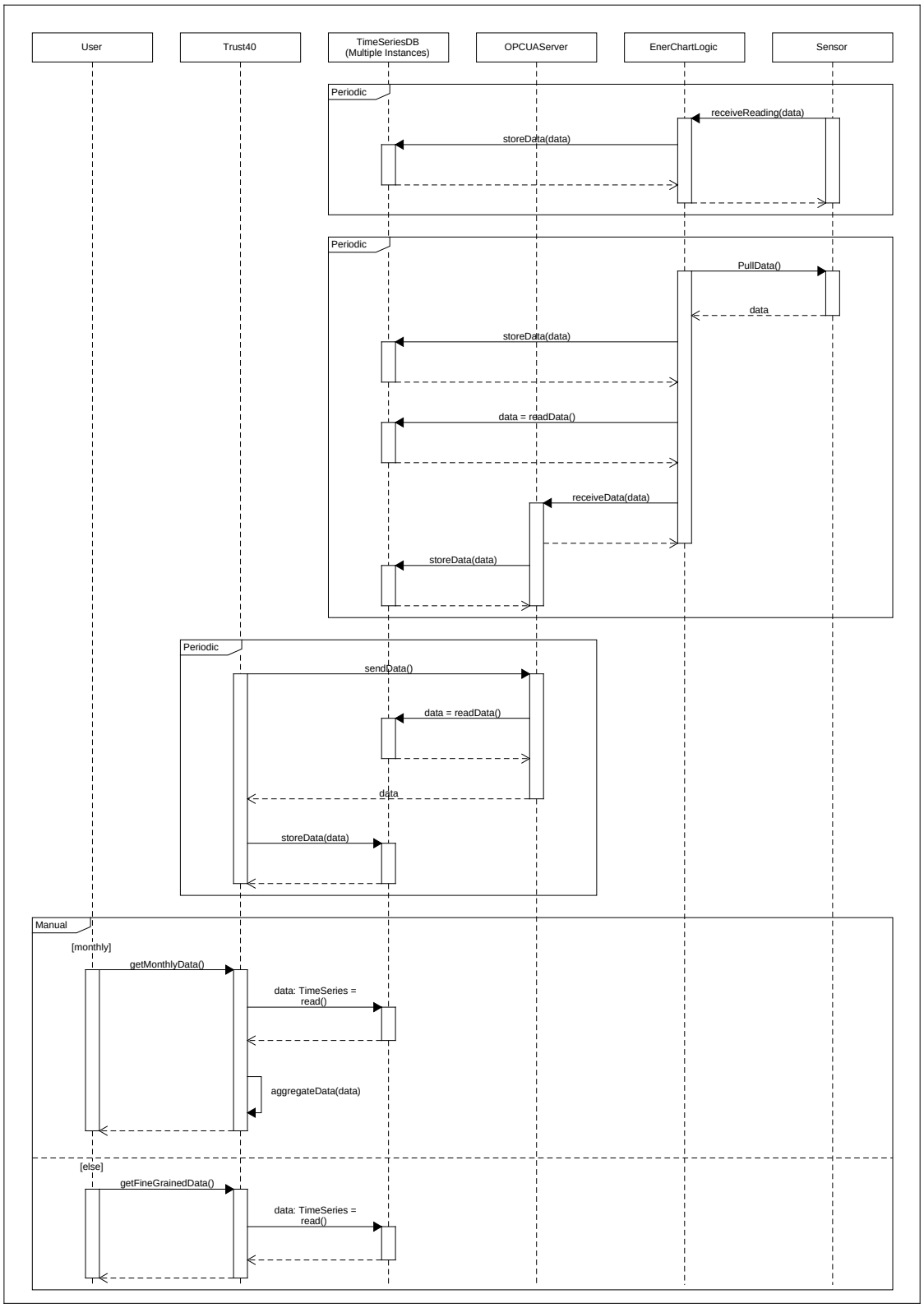


Figure 8.4: Sequence diagram for the *EnergyScenario* case study

or for an employee who has more access rights in regards to the EnMS. The context set *{normal, internal}* should be allowed to access the monthly data by calling the function *getMonthlyData()* while the external employee shouldn't. The energy officer is described by *{expert, internal}* and the external technician by *{expert, external}*. The context *Location* is used to describe if the user has access to the information while being on the company site or if he is able to request the information remotely. The last context distinguishes between manual access by the user or periodic access from the system and the sensors. For this case study, five scenarios are defined. *S1* describes the positive use of the system as described in figure 8.4, and *S2* describes the same usage but additionally contains the forbidden use cases modelled as misusage diagrams. *S3* describes a wrong configuration in which the periodic functions are manually requested by the user. *S4* describes a scenario in which an external employee tries to request the monthly data from the EnMS which is a functionality only provided to internal users. In scenario *S5* the external technician tries to request the energy data without having the access rights granted.

8.3 Accuracy

For the goal G1 *Accuracy* three questions were defined. The first is regarding the accuracy of deriving the policies, the second regarding the combination process of the context sets and the third is in regard to error detection. However, the metrics derived for these three questions are the same: Precision and Recall, as suggested by Metz.[31] Only the definition of the parameters used to evaluate these metrics is going to differ from question to question.

The first metric is *Precision*. For this metric, the true positives (*TP*) and false positives (*FP*) are used to calulate the precision $P = \frac{TP}{TP+FP}$. The definition of *TP* and *FP* depends on the question being answered. For the accuracy of deriving policies, *TP* is going to be the number of *SEFFs* which are affected by an *EntryLevelSystemCall* and, after the execution, have a policy assigned to them. *FP*, in this case, is the number of *SEFFs* which have a policy assigned to them for the corresponding *EntryLevelSystemCall* even though they are not affected by it.

For the question of how accurate the *PolicyReducer* is, *TP* is defined as the number of context sets for which the reducer has found elements which can be reduced. *FP* are the context sets which are affected by the reducer even though they are not affected by the corresponding *EntryLevelSystemCall*.

In regards to errors, *TP* are the errors found while executing the algorithm. *FP*, in this case, are all errors which have been detected but which are not actual errors.

The second metric used is *Recall*. For this metric the true positives (*TP*) and false negatives (*FN*) are used to calulate the recall $R = \frac{TP}{TP+FN}$. Again, the definition of *TP* and *FN* depends on the question. *TP* is going to be the same for the calculation of *R* as it has been for *P*. For deriving policies, *FN* is the number of *SEFFs* which do not have a policy assigned to them for the corresponding *EntryLevelSystemCall* after the execution of the *Deriver*. *FN* in regards to the *PolicyReducer* is the number of context sets which should be affected by the reducer but haven't been affected be the execution. For the detection of errors, *FN* are all the possible error cases not detected by the algorithm.

	TP	TF	FN	Precision in %	Recall in %
SMSApp					
S1	7	0	0	100	100
S2	7	0	0	100	100
DistanceTracker					
S1	6	0	0	100	100
S2	6	0	0	100	100
Travelplanner					
S1	8	0	0	100	100
S2	8	0	0	100	100
EnergyScenario					
S1	19	0	0	100	100
S2	19	0	0	100	100

Table 8.8: Accuracy for deriving policies

The results for the accuracy measurement in regards to deriving policies are shown in table 8.8. For all the scenarios in all the case studies we achieve a precision and a recall of 100%. This is an expected result, since the classes have been tested as described in chapter 7.7 and that no predictions or heuristics are used during the deriving process. The use of these could be a topic for future works as described in section 10.1. An observation that can be made is that both the scenario with only the use of usage diagrams and the scenario with misusage diagrams have the same amount of expected derived policies (*TP*). This is explained by the fact that the derived misusage policies do not lead to policies in the final context model, since the model is not able to represent negative policies as described in section 3.6. Instead, they are only used during the combination process to evaluate the correctness of the context model. A derivation between the *TP*-value between scenario 1 and 2 of a case study would rather indicate a wrong setup of the evaluation scenarios.

	TP	TF	FN	Precision in %	Recall in %
SMSApp					
S1	7	0	0	100	100
S2	10	0	0	100	100
DistanceTracker					
S1	6	0	0	100	100
S2	9	0	0	100	100
Travelplanner					
S1	8	0	0	100	100
S2	12	0	0	100	100
EnergyScenario					
S1	15	0	0	100	100
S2	22	0	0	100	100

Table 8.9: Accuracy for reducing policies

The results for the accuracy measurement in regards to the policy reduction and combination are shown in table 8.9. For all the scenarios in all the case studies we achieve a precision and a recall of 100%. This is again an expected result in the same manner that the modules have been tested by unit tests as described in chapter 7.7 and that no predictions or heuristics are used during the combination process. The difference between the above results is that in these measurements we expect to see a difference between the scenarios with and without misusage diagrams. Since the misusage diagrams result in temporary policies used for validation there exist more policies for which rules can be applied. This is the explanation for the increase in TP-value between the scenarios 1 and 2 in each case study.

	TP	TF	FN	Precision in %	Recall in %
SMSApp					
S2	0	0	0	100	100
S3	1	0	0	100	100
S4	1	0	0	100	100
DistanceTracker					
S2	0	0	0	100	100
S3	1	0	0	100	100
S4	1	0	0	100	100
S5	1	0	0	100	100
Travelplanner					
S2	0	0	0	100	100
S3	4	0	0	100	100
S4	1	0	0	100	100
S5	3	0	0	100	100
EnergyScenario					
S2	0	0	0	100	100
S3	8	0	0	100	100
S4	2	0	0	100	100
S5	4	0	0	100	100

Table 8.10: Accuracy for detecting errors

Table 8.10 shows the results of the measurement of error accuracy. Again, the same reasons apply which result in the precision and a recall of 100% being the expected values. The scenarios 1 were excluded in this evaluation since they do not contain misusage diagrams and therefore no contradicting policies can exist. The scenario 2 for each case study shows a TP of 0, which means no errors are expected to be found. These scenarios are used to show that with a correct configuration with both usage and misusage diagrams, no errors are expected and no errors are mistakenly thrown. The rest of the scenarios have errors according to the expected misconfiguration.

8.4 Effort reduction

For the second goal G2 *Effort Reduction*, the improvement of efficiency of the developer while creating an access policy system should be measured. Therefore, the two approaches should be compared with each other in regards to the effort needed to create the complete system access policy specification. For the measurement, not the actual effort for creating the access policy description is measured, since for this evaluation, no experiment was conducted. Instead, the effort for creating the system specification is approximated by the number of policies needed to specify the complete system.

The assumption in this evaluation is that the developer creating the specification is trained in the used environment and knows the tools used for modeling the system and for the creation of the policies, in this case *Palladio*. Additionally, we assume that during the normal development process the process artifacts are already created. This means the effort for creating usage diagrams, repository and system models is not measured as part of this evaluation. Another assumption made for the evaluation of this measurement is that the effort for creating policies takes the same amount of time with both approaches. This assumption is based on the fact that both the *Policy Specification*, used for modeling policies of the data element on the *SEFF* level, and the *Context Specification*, used for modeling policies on usage diagram level, take the same number of elements to create, only deviating in the link to the PCM-element connected to them. Since the developer creating the policies knows the environment, we assume finding the corresponding elements inside the PCM model takes roughly the same amount of time for both the approach using SEFFs and the approach using usage diagrams.

With these assumptions made, the analysis for the improvement in efficiency is done by comparing the number of policies needed to be created by the developer. The base value for the comparison is the manual creation of policies with the context model. The policies have to be created for each *SEFF* individually. The second value is the number of policy specifications needed to be created with the new approach. For system calls in the usage models, the policy specifications have to be created. Additionally, the unwanted usage scenarios can be specified with the help of misusage diagrams. For the measurement, these two cases where split into two separate measurements since the complete system can theoretically be described with only the usage diagrams and no misusage diagrams. To ensure that both approaches cover the same access policy specification of the system we use the results from chapter 8.3. Since the approach has shown to have high precision and recall we assume that the derived policies matched the polices created for the comparison. The scenarios used for the different measurements of each case study are described in section 8.2. For the measurement without the use of misusage diagrams, scenario 1 of each case study is used, and for measurements with the specification of misusage diagrams, scenario 2 is used. Both scenarios cover the same use of the system, with scenario 2 also explicitly ensuring the unwanted access of certain elements is guaranteed.

The results of the measurement without the use of misusage diagrams is shown in table 8.11. The *SMSApp* case study has the least amount of reduction with an effort reduction of 33%. The case studies *DistanceTracker* and *Travelplanner* have a reduction of 50% or more. This shows that with the new approach a clear reduction of effort can be achieved. This conclusion was made with a limitation for the base approach. It was ensured that the

	Number of policies		%	Reduction in %
	New	Base		
CS1: SMSApp	4	6	67	33
CS2: DistanceTracker	3	6	50	50
CS3: Travelplanner	3	8	38	62
CS4: EnergyScenario	6	11	55	45

Table 8.11: Comparison with only usage diagrams

number of policies in the base approach was limited to only 1 policy per *SEFF*. So with this assumption, the minimal subset of policies needed to describe the system was used. In a real environment it might be the case that the user does not work in the optimal way and instead creates multiple policies affecting the same *SEFF* (for example for better human readable context policies), which would therefore increase the number of total policies. This would lead to an even lower ratio between the two approaches meaning an even higher effort reduction percentage.

	Number of policies		%	Reduction in %
	New	Base		
CS1: SMSApp	6	6	100	0
CS2: DistanceTracker	5	6	84	16
CS3: Travelplanner	5	8	63	37
CS4: EnergyScenario	9	11	82	18

Table 8.12: Comparison with misusage diagrams

The results of the measurement with the access right violations explicitly modeled as misusage diagrams are shown in table 8.12. Compared to the results without the use of misusage diagrams, it is observed that each case study has a lower effort reduction. In the case of case study 1, the *ContactSMSManager*, there is no effort reduction at all. This result was expected, since the number of policies in each case study increased by adding the misusage diagrams, but the number of data elements represented by the *SEFFs* stayed the same. So in regards to the effort reduction, the use of misusage diagrams seems to be counterproductive. But with the use of the misuage diagrams, the benefit of the additional verification of the access policies is added. So in a productive use of this new approach, a certain compromise between verification and effort reduction has to be made.

To analyse which parameters have an influence on the effort reduction we use the PCM parameters shown in table 8.3. Since the number of *Interfaces*, *RepositoryComponents* and *AssemblyContexts* indirectly influence the number of *SEFFs*, we only analyze the effort reduction in relation to the number of *SEFFs* and *Usagemodels*. The two graphs in figures 8.5 and 8.6 show the effort reduction in regards to these two parameters of the case studies. The first shows the relation between the number of *SEFFs* and the effort reduction, the second between the number of usage diagrams and the amount of effort reduction. In both diagrams no clear relation between these parameters and the amount of effort reduction can be seen. By looking more closely at the case study with the highest effort reduction,

the *Travelplanner* case study, which has only one usage diagram, but the second highest amount of *SEFFs*, the assumption can be made that the effort reduction might depend on the relation between these two parameters. This relation is shown in figure 8.7. The graph indicates that this assumption might be correct, but the data size is too small to come to a clear conclusion. This could be a topic for further research. In all three graphs it can be clearly seen that the amount of effort reduction is directly influenced by the use of misusage diagrams.

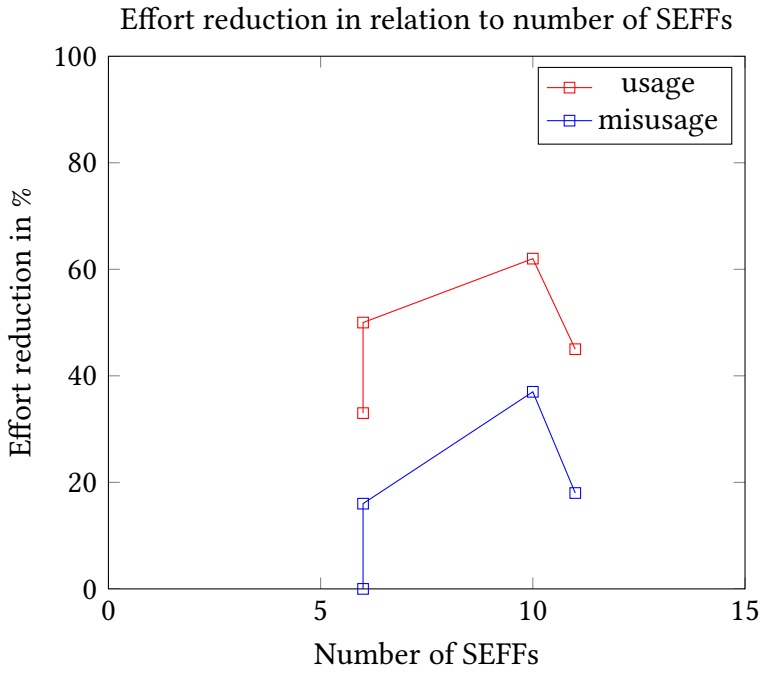


Figure 8.5: Effort reduction in relation to number of SEFFs

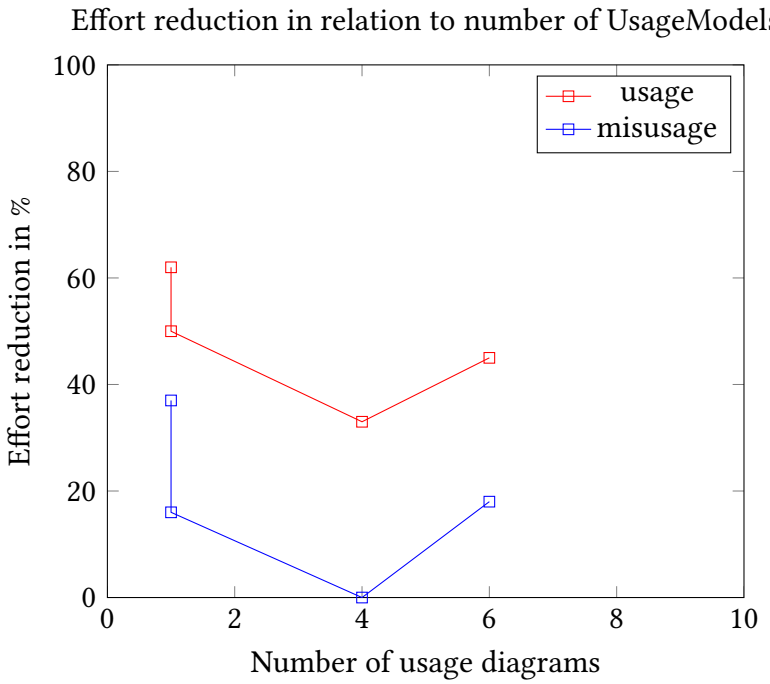


Figure 8.6: Effort reduction in relation to number of usage models

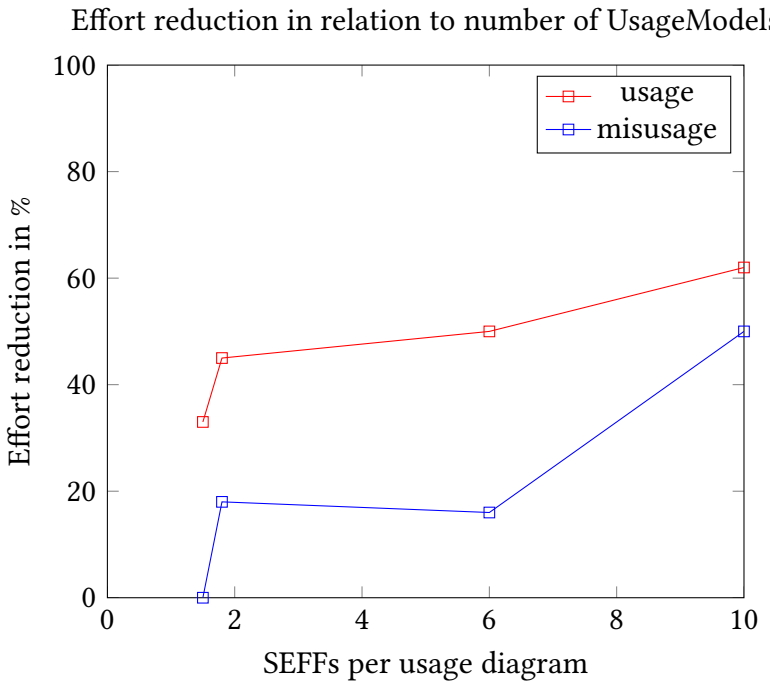


Figure 8.7: Effort reduction in relation to the amount of SEFFs per usage diagram

8.5 Scalability

For the goal of scalability the case studies are not used for the evaluation; instead a model is generated according to certain parameters. Each parameter is varied in a range while the other parameters are kept the same to see the effect of this parameter. Both PCM-model as well as context-model parameters are varied. For the test setup, a model generator was implemented which would create the different models accordingly. The models generated are the usage model, the system model, the repository model and the context model. The generated models are created consistent to each other. To account for variance in execution time due to side effects each measurement for one parameter variation was run multiple times and the average of the measurements was taken. Also during the first runtime analysis, it was observed that the first few runtime measurements highly differ from the rest of the measurements. A warmup phase was introduced in which the measurements were run a few additional times and the results of the earliest measurements were ignored.

Id	Parameter
PCM	
1	# interfaces
2	# methods per interface
3	# assembly composition depth
4	# assembly composition with
5	# usage diagrams
6	# system calls
ContextModel	
7	# context sets
8	# contexts per context set
9	# hierarchical context depth
10	# hierarchical context width
11	# policies
12	# contexts per policy

Table 8.13: Parameters which are varied during the runtime analysis

The list of parameters which are varied are shown in table 8.13. Parameter 1 describes the number of interfaces existing in the repository model. Increasing this parameter results in more interfaces per component and therefore in an increase in the total number of *SEFFs*. Additionally, this parameter also affects the total number of *AssemblyConnectors* in the system model. Parameter 2 influences the number of operations each interface has. This also increases the total number of *SEFFs* the same way parameter 1 does, but does not affect the number of *AssemblyConnectors* present in the assembly.

Parameters 3 and 4 both affect the composite components inside the system model. With parameter 3, the depth of nested composite components is varied. A value of 10 means that beginning on the highest level, the system composition, each assembly context on this level will be a composite composition containing themselves composite components with a depth of 9. Parameter 4 varies the width of each composite component. A value of 10

means each composite component contains a chain of 10 assembly context with, beginning with the first, each of them being connected to the next in chain over an *AssemblyConnector* and having an *ExternalCall* on the repository level. Even though parameter 3 increases the number of composite components on the assembly level, the total number of basic components is not affected, and therefore the number of *SEFFs* stays the same as well. Parameter 4 on the other hand increases the number of basic components and therefore the number of *SEFFs*.

Parameters 5 and 6 vary the parameter of the usage model. Increasing parameter 5 leads to more usage diagrams and therefore more scenario behaviours respectively. Parameter 6 affects how often each operation of the provided system interfaces is called. Both parameters lead to an increase in *EntryLevelSystemCalls* and therefore an increase in the numbers of policy specifications affecting each *SEFF*, but the number *SEFFs* stays the same when varying them.

Parameter 7 changes how many context sets are available and parameter 8 increases the amount of context each context set has assigned to it. These parameters don't affect the amount of *SEFFs* present. Parameter 9 and 10 change the hierarchical contexts inside the context model. With the hierarchical context depth, the amount of layers in a hierarchical context is increased, meaning the longest path from root to the furthest child gets longer. The hierarchical context width changes how many child nodes each hierarchical context has in one layer. Parameter 11 changes how many policies are present in the model when the reducer is applied, and parameter 12 increases the number of context sets each policy specification has assigned to it.

Id	ParameterValue						
	1	10	25	50	100	150	200
1	0.0064	0.0361	0.2128	0.4145	9.932	-	-
2	0.0004	0.0053	0.0208	0.0684	0.2646	0.5387	0.9406
3	0.0004	0.0012	0.0021	0.0043	0.01	0.0153	0.0326
4	0.0004	0.008	0.0237	0.0796	0.2691	-	-
5	0.0005	0.0047	0.0198	0.0658	0.2283	0.5478	0.87
6	0.0005	0.0052	0.015	0.0629	0.2369	0.5019	0.8953

Table 8.14: Runtime results for PCM parameters

The results of the runtime analysis by varying the PCM-model parameters (Parameters 1-6) can be found in table 8.14. Additionally, the results are shown as a graph in figure 8.8. In general, the results can be grouped into 3 categories. The first group is the parameters which not not to influence the runtime at all in a significant way. In our case this is parameter 3, the depth of the hierarchical context. The second group is the parameters affecting the runtime in a linear way. These are the parameters 2,4,5,6. The last group seems to cause an exponential effect on the runtime. This seems to be the case for parameter 1. One observation is that the number of affected *SEFFs* seems not to be the influential factor for this exponential increase in runtime, since not all parameters which increase the amount of affected *SEFFs* are in this third group.

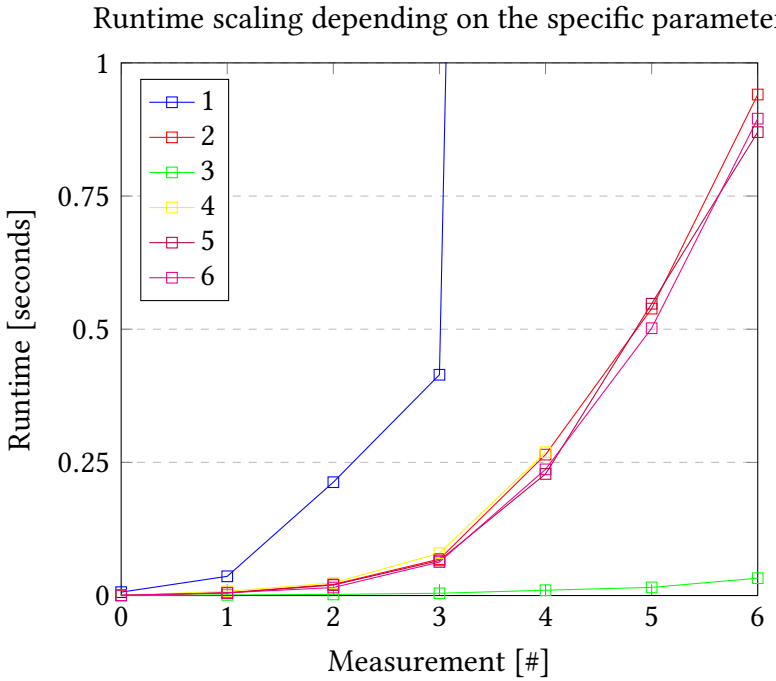


Figure 8.8: Graph plotting the runtime results

There are two incomplete measurements in the table 8.14. One is for the parameter 1 for values higher than 100. Here, the problem was that the measurement did not seem to be able to be completed in a reasonable time frame and was manually aborted. This observation aligns with the rest of the measurements for this parameter indicating an exponential effect in runtime. One possible explanation could be that the number of *Assembly Connectors* causes this issue. For real use cases, this high number of *Assembly Connectors* for a single *Assembly Context* seems not to be realistic and can therefore be ignored. The second incomplete measurement is for parameter 4, also for values over 100. Here, the issue was a stack-overflow exception caused by the recursive handling of external calls described in chapter 7.3. Since the issue was found during the evaluation step, the time frame of this thesis didn't allow for a bigger functional change inside this handling of the algorithm. Similar to the first parameter, the issue seems not to be relevant for realistic use cases, since the issue was caused by a long chain of components (more than 1000 components chained after each other). For the impact of this parameter on the runtime, we take the trend of the first measurements to assume a linear runtime effect.

Table 8.15 displays the results for the parameters 7-12, which affect the context model. These parameters are not plotted since only one of them seems to have a significant effect on the runtime of the program. Parameter 11, which increases the number of policy specifications, seems to have a linear effect on the runtime. The results for this section were computed on laptop, the specifications can be found in table 8.16.

Id	ParameterValue						
	1	10	25	50	100	150	200
7	0.0042	0.0443	0.0348	0.0193	0.0225	0.0211	0.0186
8	0.0013	0.0014	0.001	0.0012	0.0014	0.002	0.0012
9	0.0004	0.0006	0.0005	0.0009	0.0005	0.0005	0.0006
10	0.0007	0.0005	0.0004	0.0004	0.0005	0.0009	0.0005
11	0.0001	0.0005	0.0093	0.0234	0.108	0.3202	0.7865
12	0.0003	0.0005	0.0005	0.0003	0.0005	0.0006	0.0004

Table 8.15: Runtime results for context parameters

Parameter	Value
System Manufacturer	LENOVO
System Model	20ARS2V900
Processor	Intel(R) Core(TM) i7-4600U CPU @ 2.10GHz, 2694 Mhz, 2 Core(s), 4 Logical Processor(s)
Installed Physical Memory (RAM)	12.0 GB
OS Name	Microsoft Windows 10 Pro

Table 8.16: Hardware specification

8.6 Threats to Validity

The threats to validity are structured into four categories, according to the guidelines for case study research of Runeson and Höst[38]. The categories are described in the following.

- *Internal validity*: Internal validity ensures that only the factors we expect to have an influence are the influencing factors and therefore our assumption about cause and effect is correct. We assume that the influencing factors for this evaluation are the used scenarios and models. This factor was reduced by not creating new models and instead using models which are already existing and have been used in other evaluations. Additionally, the case studies had descriptions of their use case and their security constraints. This information was used to create the used context models and the additional scenarios for the misusage cases. A possible invalidity might have been introduced with this step. This risk we tried to minimize by using four case studies. For the performance evaluation, only one parameter was changed at a time to ensure the effect of it can be measured without the influence of other effects.

- *External validity*: External validity allows the application of the findings of this evaluation to other contexts and therefore to generalize them in respect to other situations and use cases. Runeson and Höst [38] describe that case study based evaluations are good for understanding a phenomenon instead of yielding a result which has a great representativeness. According to them, results conducted with case studies can help understand cases with similar characteristics. The results of this evaluation,

therefore, may be applicable to cases which both have a dynamic environment which can be modelled with contexts and for which a security description on the usage model level, meaning on the interface level to the system, is possible. This belief is affirmed by the fact that multiple external case studies were used which all indicated the same behaviour of our approach.

- *Construct validity*: This category ensures that the metrics used to answer the questions defined in section 8.1 are correct and that the measurements can be explained according to how the system should behave. The *accuracy* metrics we defined according to Metz[31]. The resulting measurements of precision and recall were as we expected. For the performance measurement the runtime seems to be the perfect fit. Here, the results of the measurements could also be explained with the used algorithm. For the effort reduction, we used a metric for which we had to make some assumptions. Here, a conducted experiment might have yielded more accurate results but we think that the results in some cases of more than 50% reduction give some indication that the new approach can have an effort reduction compared to the normal context based approach. Additionally, we gained some insights into how the use of misusage diagrams influence this metric. In general, there is less room for interpretation of the results with the use of metrics than, for example, a survey where the answers have to be further interpreted.

- *Reliability*: Reliability means that the results are reproducible and do not depend on the researcher executing the evaluation. For the *accuracy* measurements, the values needed to calculate precision and recall do not vary for different executions since the program does not depend on heuristics or randomness. For the *effort reduction*, no experiment was conducted which is good in regards to reliability. In the case studies, different policies could be used for the evaluation, but here the limitation we made to limit the number of policies to 1 per *SEFF* ensures the value may only vary in a beneficial direction. The results of the *performance* measurements may vary depending on the hardware on which it is executed but the general influence of the different parameters and their effect on the runtime should be the same. For all three goals, the measurements of the metrics were done using an automated test setup as described in chapter 7.7. This ensures that all the results are reproducible and do not depend on the executing researcher.

8.7 Summary of the validation

With the GQM-method and the four used case studies, the conducted evaluation showed that the new approach is able to accurately derive and combine policies from usage and misusage diagrams and that, in most cases, the effort for the creation of policies can be reduced. Accuracy was chosen as the most important goal, since the correct handling of access policies is fundamental to each access system, and the evaluation concluded that both parts, derivation and reduction, had a precision and recall of 100 %. The results suggest a high accuracy of our approach. Additionally, the accuracy of detecting errors showed that possible errors in the configuration of the system done by the user can be

detected with the help of misusage diagrams. This validation of the context model impacts the effort reduction since the effort of creating the misusage diagrams is added. The evaluation of effort reduction concluded that even with the assumption of having only 1 policy per *SEFF* the new approach is able to reduce the effort in creating policies in regards to using the context based access system. Additionally, it was analyzed which parameters had an effect on the runtime and to what extent. This analysis can be helpful for future works which use this thesis as a foundation.

9 Assumptions and Limitations

For this thesis we made some assumptions and limitations during the development of the approach to make it feasible for the given time frame. In this chapter, these assumptions and limitations are summarized and in the next chapter we describe how some of them can be removed in future.

9.1 Assumptions

One of the assumptions made was that all the models used during the derivation and reduction process are valid. This means the underlying Ecore models have been checked for any violations, for example, in regards to multiplicities of elements, required fields or upper and lower bounds of elements. Additionally, we assume that the models are "logically" correct. This means that, for example, in the *ComposedComponent* there is no instance of itself which would lead to an endless depth of nested components, or that the *AssemblyContexts* are not connected in a circular way with themselves and without system interfaces. We added checks at possible problematic locations and tested accordingly, but we cannot guarantee that all possible misconfigurations were found [8].

For the evaluation we made the assumption that the effort for creating the *PolicySpecification* on the *SEFF* level takes the same amount of time as the creation of the *ContextSpecification* in the usage diagrams. This was due to the fact that both specifications contain roughly the same number of elements to create. This was based on another assumption that the developer creating the policies knows both the PCM models and the context models equally well. These assumptions were made to be able to make an observation in regards to the effort reduction properties of the approach without conducting an actual experiment. Since multiple case studies were used which all indicate some effort reduction we feel confident in our belief that this new approach can reduce the effort for creating policies. However, a real experiment might need to be conducted to confirm these assumptions.

Since access control per definition grants access to resources, we made the assumption that *SEFFs* can be used as an abstraction for data elements. Additionally we restricted this further in that each data element is only represented by one *SEFF* and vice versa. This assumption also includes the expectation that the complete security description can be done with the information of the *SEFFs*.

9.2 Limitations

One limitation of the current approach is that the policy specifications in the context model are assigned to *SEFFs* instead of *AssemblyContext*. This leads to the fact that different

instances of the component cannot be differentiated and are treated as one in regards to the security aspects. For the evaluation, this didn't have an effect since, as a workaround, multiple versions of a component can be created in the repository for each used instance. For the use of our approach this might need to be kept in mind, since it could lead to wrong configurations. A solution to this would be to change the context model, which is outlined in section 10.4.

Another limitation is that the instance of the different model types is currently limited to one for each model type. This didn't have an impact on the evaluation since the four case studies already fulfilled this limitation. In a real life setting, this might not be the case. The changes needed to adapt the software for multiple files are described in section 10.2.

With the current implementation it is not possible to store the context derived from the misusage diagrams. This means they are only created temporarily and removed after the execution of the program. The reason for this is that the context model does not allow the storage of negative contexts or policies. While it might be helpful to be able to store these negative contexts, there are also some negative aspects connected to it. For example, the evaluation of a policy misconfiguration might be moved from the design phase to the runtime phase if conflicts are not resolved. If no conflicts are present there is currently no need to store the negative information about context. In the end, this limitation is part of the context model and cannot be resolved within this thesis.

The current approach only works well for systems where the security description is available in the form of system usage. For example, the information forbids the use of a certain function of a component, but the use of this component inside the system is not clear. This is due to the fact that, in the derivation process, it is not possible to assign context specifications to internal calls directly, only on system calls. However, since the approach is backwards compatible to the normal context model, it is still possible to manually create *PolicySpecifications* which will then be considered during the combination process. Since it is not possible to save negative context specification in the context model, it only allows the addition of positive access policies. It is therefore not possible to, for example, explicitly forbid access for one particular function, instead the complete chain of functions is always affected.

Another limitation is that the role responsible for the creation of the usage diagrams has to know the security aspects of the system. In this case, this role is the domain expert. If the security expert of the system is a different person than the domain expert, the responsibility for the usage models is shared between two roles, which conflicts with the idea of having separation of concerns between the roles.

During the evaluation, two limitations of the current software implementation were also found. The first is the limit for maximum length of the chain of external calls. The second, the number of interfaces of each component. We assume that both of these limits are greater than the realistic values present in projects, therefore they should not be problematic for the use of the software.

The design decision not to use a policy combining language but instead implement the rules as classes working directly on the context model is also a limitation. With this approach, if new policies should be added, as suggested in section 10.3, the code of the software needs to be changed. With the use of a policy combining language the changes can be made directly in the language by adding a new policy combining algorithm. The

advantage of the chosen design is that no additional model in the form of the policy combining language was needed, otherwise the specified contexts would have to be transformed from the context model into the language model after the derivation process, and back after the combination process. With the current approach, both parts can be done using only the context model. This also reduced the risk of possible complications during the development process.

Another limitation is that the current rules do not include any heuristics or any form of generalization. So for a rule to be applied to the context set, the exact criteria of its definition have to be met. A possible way to extend this behaviour is illustrated in section 10.1.

With the approach it is also not possible to resolve possible errors found during the combination process. In the case of an error, additional knowledge of the security expert is needed since it could arise for two different reasons: Either the configuration of the context is wrong and needs to be adapted, or if the assigned context for the usage scenario seems to be valid, the reason is that it is not realisable with the current system assembly and component composition.

10 Future Work

In future work, several topics can be further investigated. The use of heuristics might be helpful in removing some of the limitations of the current approach, more implementation of rule-sets might give a higher effort reduction and the extension to include multiple files and systems might increase the applicable use cases. These topics are briefly described in this chapter.

10.1 Adding Heuristics

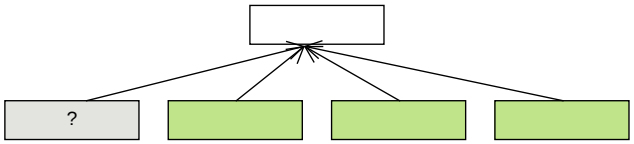
Some of the approaches mentioned in chapter 4 use heuristics to merge and simplify policies. An example for this is shown in figure 10.1. The element in the example is affected by all but one child context of a hierarchical context set (direction down). With the current design decision to treat contexts which are not explicitly allowed as forbidden contexts no combination is possible. Instead of this hard limitation, a different approach would be to have them as an uncertain state. Then a heuristic could be "if 75% of child contexts are allowed, and no negative child contexts are present" the child contexts get substituted for the parent context. This would extend the implementation of the rule mentioned in section 6.2.4.

The problem with optimization based policy creation is that the simplified policies might over-fit the input data if it does not generalize well and therefore would not be robust [46]. To generate good heuristics a lot of test data is needed in order to have an accurate and robust prediction model [36]. These were reasons why this feature was not included as part of this master thesis.

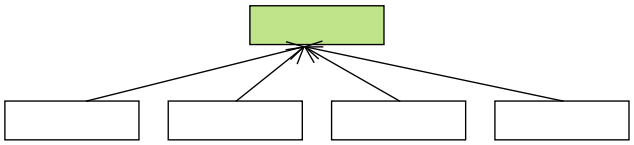
10.2 Increase Scope of Approach

The current implementation only allows one file for each model type like the repository model or the context model. For the scope of this thesis this limitation had no impact on the functionality. For a real usage in an environment with multiple developers this might be different. For example, different developers could work on separate components and describe the context model for the parts they are responsible for. To add this feature the *modelloader* class needs to be extended to be able to handle multiple files per model type and in the algorithms, another loop iterating the models needs to be added. A more difficulty aspect to handle is how possible changes to the models need to be saved, for example to which file new created policies should be added.

Another aspect is the system boundaries. Currently, the created approach works on one system and the usage models begin on the interface level of the system. A future



(a) Initial applied contexts



(b) After applying rule using heuristics

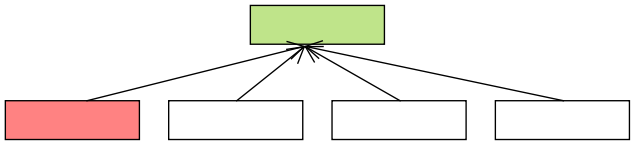
Figure 10.1: Simplifying hierarchical context set with heuristics

work could increase this scope to allow multiple systems. Here, the chain started at one *EntryLevelSystemCall* might not only affect components inside the system but also components in different systems. Additionally, the validation done with the misusage diagrams could be extended to multiple systems. For example, multiple systems might have no access violations in the context model configuration on their own, but the interaction between different systems could lead to possible access violations.

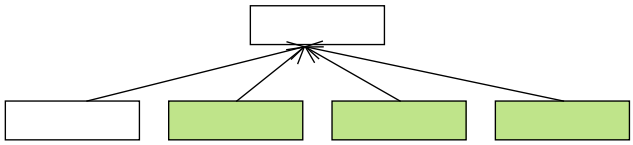
10.3 Adding more Rule Implementations

The iterative process used during development meant rules were created in succession after the previous rule was added to the *policyreducer* and tested. For some of the rules which were created in the design step of the thesis the time frame didn't allow for them to be implemented and tested in a sufficient manner. These rules can be implemented in future to add more functionality to the plugin.

Figure 10.2 shows an example in which the parent element of a hierarchical context (direction down) is part of a context set as well as a negative child element. Currently, this configuration would result in an error as described in 6.2.7, since the child element should not have access, but inherits it from the parent node. One possible way to resolve this situation is seen in b). The parent node is removed from the context set, and instead all the child elements are added to it. Therefore these child elements would not inherit the access policy indirectly, but would have explicit allowed access. For the negative child element no access policy is created. The side effect would be that policies with context sets only referencing the parent context would not be allowed anymore. But for an access control system where only the policy definition contains the parent context and where the actual instances will contain only the leaf nodes, this rule would work without this side effect. This rule would, therefore, depend on the actual implementation of the access control system using the context based approach. Additionally, other researchers might come up



(a) Initial applied contexts



(b) After applying new rule

Figure 10.2: Possible rule for positive parent context with negative child context

with ideas for new rules and more possible use cases in the future. With the current design of the *PolicyReducer*, new rules can be easily added to the current algorithm.

10.4 Move from SEFFs to Instances

With the current approach, the policies which are derived from the usage model are being applied to the *SEFFs* of components in the repository model. The problem with this is that if a component is instantiated multiple times at different places in the assembly model, all of the policies affecting these *AssemblyContexts* will be mapped to the same *BasicComponent* in the repository model. Therefore, no differentiation between the instances is possible anymore in the context model. This can lead to errors for misconfiguration being thrown even though the usage diagram and the misusage diagram would affect different *AssemblyContexts* and therefore would not cause an error. On the other hand, possible combinations of policies on one *SEFFs* might lead to the accidental allowance of unwanted contexts since the combined context sets actually affect different elements. For the evaluation of this thesis this was not a problem since it was possible to create duplicated components in the repository model if it was needed. For a real use of this approach this workaround cannot be used since duplicated code is one of the major indicators of poor maintainability of software [35].

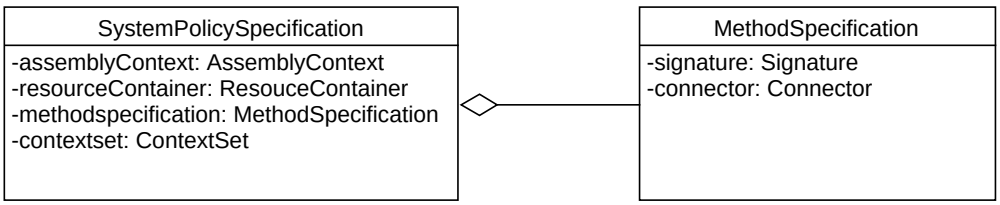


Figure 10.3: Possible context model change

Figure 10.3 shows the changes made in the context model for it to handle this feature. Instead of the *PolicySpecification* being applied to a *SEFF* in has a *AssemblyContext* as a reference and the affected method. This functionality was not implemented in this thesis since the realization of this problem came during the later stage of the development. A change in one of the basic functionalities could have had possible major impacts which might have not been feasible to fix in the given time frame. To implement this in future work the algorithm finding the affected data elements needs to be adapted from *SEFF* to the *AssemblyContext* and the according signature and interface.

11 Conclusion

In this thesis we introduced an approach for the automatic generation of policies on the data element level by deriving context specifications from the usage description of the system. This approach combines the idea of generating policies from existing artifacts with the context based access control which is a model driven security approach. The goal was to analyze if this approach can lead to an effort reduction in the creation of access policies while simultaneously deriving a correct model without violating the fundamentals of access control systems.

In the main part we described the two parts of the approach. The deriving of policies from usage and misusage diagrams depends on the PCM models. The calls to the system's interfaces are the entry point from which all affected data elements have to be found. For our approach, these data elements are *SEFFs*. The second functionality is the application of defined rules on the resulting context model. Here, the different rules are introduced and the concepts behind them are explained. For both parts, the implementations is briefly explained with the help of pseudo code.

In the evaluation we used four case studies to analyze the three goals we defined for our approach. We showed that the derivation of the policies, as well as the reduction and the combination of them can be achieved with great accuracy. The case studies also indicate that with the new approach an effort reduction can be achieved. The measurement of this metric was done with some assumptions and an experiment might need to be done in further work to confirm the assumed effort reduction. Additionally, a performance analysis was carried out to see the impact of different model parameters on the runtime. Since the two parts of the approach, the derivation of policies from usage and misusage diagrams, and the combination of context sets according to the defined rule set, have been implemented as separate stand-alone plugins with clear interfaces we are confident that they can be used in future work.

In summary, we introduced an approach which allows the automatic generation of access policies based on the context information assigned to usage diagrams. Additionally, the approach can detect possible security flaws in the development of secure systems during the design phase by using the information modelled in the misusage diagrams.

Bibliography

[1] M. M. Alam, R. Breu, and M. Breu. "Model driven security for Web services (MDS4WS)". In: *8th International Multitopic Conference, 2004. Proceedings of INMIC 2004.* 2004, pp. 498–505. DOI: 10.1109/INMIC.2004.1492930.

[2] V. R. Basili and D. M. Weiss. "A Methodology for Collecting Valid Software Engineering Data". In: *IEEE Transactions on Software Engineering* SE-10.6 (1984), pp. 728–738. DOI: 10.1109/TSE.1984.5010301.

[3] Steffen Becker, Heiko Koziolek, and Ralf Reussner. "The Palladio component model for model-driven performance prediction". In: *Journal of Systems and Software* 82.1 (2009). Special Issue: Software Performance - Modeling and Analysis, pp. 3–22. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2008.03.066. URL: http://www.sciencedirect.com/science/article/pii/S0164121208001015.

[4] Matthias Beckerle and Leonardo Martucci. "Formal definitions for usable access control rule sets from goals to metrics". In: (July 2013). DOI: 10.1145/2501604.2501606.

[5] Nicolas Boltz, Maximilian Walter, and Robert Heinrich. "Context-Based Confidentiality Analysis for Industrial IoT". In: *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. accepted, to appear. IEEE, 2020.

[6] Hugh Boyes et al. "The industrial internet of things (IIoT): An analysis framework". In: *Computers in Industry* 101 (Oct. 2018), pp. 1–12. DOI: 10.1016/j.compind.2018.04.015.

[7] F. Buschmann et al. *Pattern-Oriented Software Architecture, A System of Patterns*. EBL-Schweitzer. Wiley, 1996. ISBN: 9780471958697. URL: https://books.google.de/books?id=gJjgAAAAMAAJ.

[8] J.N. Buxton, B. Randell, and NATO Science Committee. *Software Engineering Techniques: Report on a Conference Sponsored by the NATO Science Committee, Rome, Italy, 27th to 31st October 1969*. NATO Science Committee, 1970. URL: https://books.google.de/books?id=O7kmAAAAMAAJ.

[9] Y. Cherdantseva and J. Hilton. "A Reference Model of Information Assurance Security". In: *2013 International Conference on Availability, Reliability and Security*. 2013, pp. 546–555. DOI: 10.1109/ARES.2013.72.

[10] C. Cotrini, T. Weghorn, and D. Basin. "Mining ABAC Rules from Sparse Logs". In: *2018 IEEE European Symposium on Security and Privacy (EuroS P)*. 2018, pp. 31–46.

[11] "Datenfluss-basierte Modellierung und Analyse von Privatheit und Vertraulichkeit in Industrie 4.0-Systemen". In: *Schlussbericht des Verbundprojekts Trust 4.0*. to be released.

[12] P.M. Duvall et al. *Continuous Integration: Improving Software Quality and Reducing Risk*. A Martin Fowler signature book. Addison-Wesley, 2007. ISBN: 9780321336385. URL: https://books.google.de/books?id=MA8QmAEACAAJ.

[13] Federal Ministry for Economic Affairs and Energy (BMWi). "IT Security in Industrie 4.0 - Action fields for operators". In: (2016). URL: https://www.plattform-i40.de/PI40/Redaktion/EN/Downloads/Publikation/guideline-it-security-i40-action-fields.html.

[14] National Cybersecurity Center of Excellence. *Attribute Based Access Control*. https://www.nccoe.nist.gov/projects/building-blocks/attribute-based-access-control. 2017.

[15] Eduardo Fernández and J. Hawkins. "Determining Role Rights from Use Cases". In: Jan. 1997, pp. 121–125. DOI: 10.1145/266741.266767.

[16] David F. Ferraiolo and D. Richard Kuhn. *Role-Based Access Controls*. 2009. arXiv: 0903.2171 [cs.CR].

[17] Nurit Gal-Oz and Eduardo Fernández. "A Model of Methods Access Authorization in Object-oriented Databases." In: Jan. 1993, pp. 52–61.

[18] Safaa Hachana, Nora CUPPENS-BOULAHIA, and Frédéric Cuppens. "Role Mining to Assist Authorization Governance: How Far Have We Gone?" In: (Oct. 2012).

[19] Vincent Hu et al. "Guide to attribute based access control (ABAC) definition and considerations". In: *National Institute of Standards and Technology Special Publication* (Jan. 2014), pp. 162–800.

[20] P. Jain, S. Dubey, and A. Rana. "Analysis and Performance Evaluation of Software System Usability". In: *International Journal of Computer Applications* 43 (2012), pp. 24–29.

[21] Jan Jürjens. "UMLsec: Extending UML for secure systems development". In: vol. 2460. Jan. 2002, pp. 412–425. ISBN: 978-3-540-44254-7. DOI: 10.1007/3-540-45800-X_32.

[22] Jan Jürjens. "Using UMLsec and Goal Trees for Secure Systems Development". In: *Proceedings of the 2002 ACM Symposium on Applied Computing*. SAC '02. Madrid, Spain: Association for Computing Machinery, 2002, pp. 1026–1030. ISBN: 1581134452. DOI: 10.1145/508791.508990. URL: https://doi.org/10.1145/508791.508990.

[23] K. Katkalov et al. "Model-Driven Development of Information Flow-Secure Systems with IFlow". In: *2013 International Conference on Social Computing*. 2013, pp. 51–56. DOI: 10.1109/SocialCom.2013.14.

[24] Kuzman Katkalov. "Ein modellgetriebener Ansatz zur Entwicklung informationsflusssicherer Systeme". PhD thesis. University of Augsburg, Germany, 2017. URL: http://opus.bibliothek.uni-augsburg.de/opus4/frontdoor/index/index/docId/4339.

[25] Butler Lampson. "Protection". In: *ACM SIGOPS Operating Systems Review* 8 (Jan. 1974), pp. 18–24. DOI: 10.1145/775265.775268.

[26] Leonard Lapadula and D. Bell. “Secure Computer Systems: A Mathematical Model”. In: 4 (Jan. 1997).

[27] Ninghui Li et al. “Access control policy combining: Theory meets practice”. In: Jan. 2009, pp. 135–144. DOI: 10.1145/1542207.1542229.

[28] Thomas Lieb. *org.palladiosimulator.pcm.confidentiality.context.policyextractor*. https://github.com/Trust40-Project/Palladio-Addons-ContextConfidentiality-BehaviorPolicyExtractor. 2020.

[29] Torsten Lodderstedt, David Basin, and Jürgen Doser. “SecureUML: A UML-based modeling language for model-driven security”. In: vol. 2460. Jan. 2002, pp. 426–441. ISBN: 978-3-540-44254-7. DOI: 10.1007/3-540-45800-X_33.

[30] Robert Martin. “Agile Software Development: Principles, Patterns, and Practices”. In: (Jan. 2003).

[31] Charles E. Metz. “CE: Basic principles of ROC analysis”. In: *Seminars in Nuclear Medicine*. 1978, pp. 8–283.

[32] Barsha Mitra et al. “A Survey of Role Mining”. In: *ACM Comput. Surv.* 48.4 (Feb. 2016). ISSN: 0360-0300. DOI: 10.1145/2871148. URL: https://doi.org/10.1145/2871148.

[33] Gustaf Neumann and Mark Strembeck. “A Scenario-Driven Role Engineering Process for Functional RBAC Roles”. In: *Proceedings of the Seventh ACM Symposium on Access Control Models and Technologies*. SACMAT ’02. Monterey, California, USA: Association for Computing Machinery, 2002, pp. 33–42. ISBN: 1581134967. DOI: 10.1145/507711.507717. URL: https://doi.org/10.1145/507711.507717.

[34] Phu H. Nguyen et al. *An Extensive Systematic Review on Model-Driven Development of Secure Systems*. 2015. arXiv: 1505.06557 [cs.SE].

[35] *Refactoring: Improving the Design of Existing Code*. USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN: 0201485672.

[36] Marco Túlio Ribeiro et al. “Beyond Accuracy: Behavioral Testing of NLP models with CheckList”. In: *ACL*. 2020.

[37] Vasja Roblek, Maja Meško, and Alojz Krapež. “A complexity view of Industry 4.0”. In: *SAGE Open* 6 (June 2016). DOI: 10.1177/2158244016653987.

[38] Per Runeson and Martin Höst. “Guidelines for conducting and reporting case study research in software engineering”. In: *Empirical Software Engineering* 14.2 (2009), pp. 131–164. URL: http://dx.doi.org/10.1007/s10664-008-9102-8.

[39] Stephan Seifermann. “Architectural Data Flow Analysis”. In: Apr. 2016, pp. 270–271. DOI: 10.1109/WICSA.2016.49.

[40] Stephan Seifermann, Robert Heinrich, and Ralf Reussner. “Data-driven software architecture for analyzing confidentiality”. In: *2019 IEEE International Conference on Software Architecture (ICSA 2019), Hamburg, 25.-29. März 2019*. IEEE International Conference on Software Architecture. ICSA 2019 (Hamburg, Deutschland, Mar. 25–29, 2019). IEEE, New York, NY, 2019, Art. Nr.: 8703910. ISBN: 978-1-72810-528-4. DOI: 10.1109/ICSA.2019.00009.

[41] Thomas Stahl, Markus Voelter, and Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. Hoboken, NJ, USA: John Wiley & Sons, Inc., 2006. ISBN: 0470025700.

[42] K. Tuma, R. Scandariato, and M. Balliu. "Flaws in Flows: Unveiling Design Flaws via Information Flow Analysis". In: *2019 IEEE International Conference on Software Architecture (ICSA)*. 2019, pp. 191–200. DOI: 10.1109/ICSA.2019.00028.

[43] Katja Tuma et al. "Automating the Early Detection of Security Design Flaws". In: *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. MODELS '20. Virtual Event, Canada: Association for Computing Machinery, 2020, pp. 332–342. ISBN: 9781450370196. DOI: 10.1145/3365438.3410954. URL: https://doi.org/10.1145/3365438.3410954.

[44] D. Verma et al. "Generative policy model for autonomic management". In: Aug. 2017, pp. 1–6. DOI: 10.1109/UIC-ATC.2017.8397410.

[45] Darshika Verma et al. "On the Impact of Generative Policies on Security Metrics". In: June 2019, pp. 104–109. DOI: 10.1109/SMARTCOMP.2019.00037.

[46] Zhongyuan Xu and Scott D. Stoller. "Mining Attribute-Based Access Control Policies from Logs". In: *Data and Applications Security and Privacy XXVIII*. Ed. by Vijay Atluri and Günther Pernul. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 276–291. ISBN: 978-3-662-43936-4.