Modularization Approaches in the Context of Monolithic Simulations

Master's Thesis of

Frederik Reiche

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

Reviewer:	Prof. Dr. Ralf H. Reussner
Second reviewer:	Jun.-Prof. Dr.-Ing. Anne Koziolek
Advisor:	M.Sc. Sandro Koch
Second advisor:	Dipl.-Inform Jörg Henß

15. March 2018 – 14. September 2018

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.

Bretten, 07.09.2018

. . . .

(Frederik Reiche)

Abstract

Quality characteristics of a software system such as performance or reliability can determine its success or failure. In traditional software engineering, these characteristics can only be determined when parts of the system are already implemented and past the design process. Computer simulations allow to determine estimations of quality characteristics of software systems already during the design process. Simulations are build to analyse certain aspects of systems. The representation of the system is specialised for the specific analysis. This specialisation often results in a monolithic design of the simulation. Monolithic structures, however, can induce reduced maintainability of the simulation and decreased understandability and reusability of the representations of the system. The drawbacks of monolithic structures can be encountered by the concept of modularisation, where one problem is divided into several smaller sub-problems. This approach allows an easier understanding and handling of the sub-problems.

In this thesis an approach is provided to describe the coupling of newly developed and already existing simulations to a modular simulation. This approach consists of a Domain-Specific Language (DSL) developed with model-driven technologies. The DSL is applied in a case-study to describe the coupling of two simulations. The coupling of these simulations with an existing coupling approach is implemented according to the created description. An evaluation of the DSL is conducted regarding its completeness to describe the coupling of several simulations to a modular simulation. Additionally, the modular simulation is examined regarding the accuracy of preserving the behaviour of the monolithic simulation. The results of the modular simulation and the monolithic version are compared for this purpose. The created modular simulation is additionally evaluated in regard to its scalability by analysis of the execution times when multiple simulations are coupled. Furthermore, the effect of the modularisation on the simulation execution times is evaluated.

The obtained evaluation results show that the DSL can describe the coupling of the two simulations used in the case-study. Furthermore, the results of the accuracy evaluation suggest that problems in the interaction of the simulations with the coupling approach exist. However, the results also show that the overall behaviour of the monolithic simulation is preserved in its modular version. The analysis of the execution times suggest, that the modular simulation experiences an increase in execution time compared to the monolithic version. Also, the results regarding the scalability show that the execution time of the modular simulation does not rise exponentially with the number of coupled simulations.

Zusammenfassung

Qualitätsmerkmale eines Software-Systems wie Zuverlässigkeit oder Performanz können über dessen Erfolg oder Scheitern entscheiden. Diese Qualitätsmerkmale können im klassischen Software-Ingenieurswesen erst bestimmt werden, wenn der Entwurfsprozess bereits vollendet ist und Teile des Software-Systems implementiert sind. Computer-Simulationen erlauben es jedoch Schätzungen dieser Werte schon während des Software-Entwurfs zu bestimmen. Simulationen werden erstellt um bestimmte Aspekte eines Systems zu analysieren. Die Repräsentation des Systems ist auf diese Analyse spezialisiert. Diese Spezialisierung resultiert oft in einer monolithischen Struktur der Simulation. Solch eine Struktur kann jedoch die Wartbarkeit der Simulation negativ beeinflussen und das Verständnis und die Wiederverwendbarkeit der Repräsentation des Systems verschlechtern. Die Nachteile einer monolithischen Struktur können durch das Konzept der Modularisierung reduziert werden. In diesem Ansatz wird ein Problem in kleinere Teilprobleme zerlegt. Diese Zerlegung ermöglicht ein besseres Veständnis und eine bessere Handhabung der Teilprobleme.

In dieser Arbeit wird ein Ansatz präsentiert, um die Kopplung von neu entwickelten oder bereits existierenden Simulationen zu einer modularen Simulation zu beschreiben. Dieser Ansatz besteht aus einer Domänenspezifischen Sprache (DSL), die mit modellgetriebenen Technologien entwickelt wird. Die DSL wird in einer Fallstudie angewendet, um die Kopplung von zwei Simulationen zu beschreiben. Weiterhin wird die Kopplung dieser Simulationen mit einem existierenden Kopplungsansatz gemäß der erzeugten Beschreibung manuell implementiert. In dieser Fallstudie wird die Vollständigkeit der Fähigkeit der DSL untersucht, die Kopplung von mehreren Simulation zu einer modularen Simulation zu beschreiben. Weiterhin wird die Genauigkeit des Modularisierungsansatzes bezüglich der Verhaltensbewahrung der modularen Simulation gegenüber der monolithischen Version evaluiert. Hierfür werden die Resultate der modularen Simulation mit denen der monolithischen Version verglichen. Zudem wird die Skalierbarkeit des Ansatzes durch die Betrachtung der Ausführungszeiten untersucht, wenn mehrere Simulationen gekoppelt werden. Außerdem wird der Effekt der Modularisierung auf die Ausführungszeit in Relation zur monolithischen Simulation betrachtet.

Die erhaltenen Resultate zeigen, dass die Kopplung der beiden Simulationen der Fallstudie, mit der DSL beschrieben werden kann. Die Resultate bezüglich der Evaluation der Genauigkeit weisen Probleme bei der Interaktion der Simulationen mit dem Kopplungsansatz auf. Nichts desto trotz bleibt das Verhalten der monolithischen Simulation in der modularen Version insgesamt erhalten. Die Evaluation zeigt, dass die modulare Simulation eine Erhöhung der Ausführungszeit im Vergleich zur monolithischen Version erfährt. Zudem deutet die Analyse der Skalierbarkeit darauf hin, dass die Ausführungszeit der modularen Simulation nicht exponentiell mit der Anzahl der gekoppelten Simulationen wächst.

Contents

List of Figures

List of Tables

Acronyms

ADL	Architecture Description Language
API	Application Programming Interface
BOM	Base Object Model
BP	Business Process
DES	Discrete-Event Simulation
DEVS	Discrete Event System Specification
DIS	Distributed Interactive Simulation
DSL	Domain-Specific Language
EMD	Earth Mover's Distance
EMF	Eclipse Modelling Framework
FOM	Federate Object Model
GQM	Goal Question Metric
HLA	High-level Architecture
IEEE	Institute of Electrical and Electronics Engineers
IntBIIS	Integrated Business IT Impact Simulation
IS	Information System
KAMP	Karlsruhe Architecture Maintainability Prediction
LCIM	Levels of Conceptual Interoperability Model
MBSD	Model-Based Software Development
MDA	Model-Driven Architecture
MDSD	Model-Driven Software Development
MoC	Model of Computation
MOF	Meta Object Facility

MSE	Modular Simulation Environment
OE	Organisational Environment
OMG	Object Management Group
OMT	Object-model Template
RDSEFF	Resource Demanding Service Effect Specification
RTI	Run-time Infrastructure
SADL	Simulation Architecture Description Language
SEFF	Service Effect Specification
SISO	Simulation Interoperability Standards Organisation
SOM	Simulation Object Model
UML	Unified Modeling Language
XML	Extensible Markup Language

1 Introduction

Wrong design decisions in the development of software can significantly influence the quality characteristics of the resulting product. Examples of quality characteristics can be performance or reliability. Determining the quality characteristics of a software system in the design phase can avoid necessary changes later in development which reduces costs or even prevents failure of the total product. For example, it is possible that stakeholders do not accept the product if it does not respond in a specific time frame under load expected to be normal (performance). The impact of design decisions that negatively influence quality characteristics of a software system can often only be determined in later phases of development when parts of the software already exist. Simulations can be used to determine estimates of quality characteristics to analyse the impacts of design decisions already in the design phase of the software system. These estimates help developers to choose or compare design decisions regarding their impact on the software system without actually realising them first. Simulations are build to analyse certain aspects of a system. The representation of the system is highly specialised for this analysis. Therefore, only features of a system which are relevant to the analysis of the desired aspects are represented. In order to analyse the impact of design decisions regarding quality characteristics on software systems such as performance, not only the systems architecture but also use-cases and the deployment hardware can be of importance. The specialisation of the system often results in a monolithic structure. Monolithic structures can induce problems in maintainability and decrease the understandability and reuse of the representation of the system. In a monolithic simulation, a significant part of the structure has to be understood to reuse it. It can be time and cost consuming to gain understanding of a monolithic system so that it can be more effective to implement a simulation again [1]. These problems can be encountered with the approach of modularisation where one problem is divided into several smaller problems [2]. In the context of simulations, modularisation can be used to describe a large simulation by several smaller simulations. Each smaller simulation can represent one feature of the system. The smaller scope of each simulation can improve its understandability. In the modularisation of simulations, a coupling approach has to be used to connect multiple simulations. Also, the coupling approach provides capabilities to enable correct interaction between simulations.

An example of a simulation to gather software quality measures is the *Palladio* approach by Reussner et al. [3]. This approach applies simulations to analyse component-based software systems with respect to the quality characteristics performance and reliability. The features represented for this analysis in Palladio are use-cases of the software system, its architectural design including the behaviour of the used components and its deployment on hardware resources. However, further influencing factors can impact the systems quality characteristics, depending on the context the software systems are used in. An example is the use of software systems in a business setting. Here business processes can have an impact on the actual performance of a software system. For instance, a software system can express acceptable quality characteristics when a normal load is applied. However, when a business process includes a break after which every employee starts to use the software-system again at the same time, the system can experience repeated overload and thus a decrease in performance. Such problems arise when software systems are not designed dependent on each other [4]. The IntBIIS [4] approach utilises *Palladio* to simulate the interaction of business processes with software systems to gather quality characteristics (e.g. response times). With this approach, IntBIIS can find interdependent influences between business processes and software-systems regarding performance already in the design process. IntBIIS includes *Palladio* with its business process representation in a monolithic structure. Therefore, the representations of the business process are designed to match the structure and representations of Palladio. The approach of matching to a specific structure can result in problems. An example is that the business process representation cannot be reused in another simulation when it does not provide the same representations and concepts of software systems. The principle of modularisation can be applied to avoid such a strict matching to a particular structure. In the context of IntBIIS, the smaller simulations are Palladio and the business process simulation itself. In this approach, Palladio would only provide certain information (e.g. the response time of a software system) to other simulations. To achieve interaction, the business process simulation and *Palladio* have to be coupled with an coupling approach. Also, the business process simulation would have to request the provision of the response times of the software system from Palladio. Due to the application of a coupling approach, the provided information can be transferred independently to their representation in the other simulation.

To provide support in the modularisation of monolithic simulations, a DSL is created with model-driven technologies to describe the coupling between several simulations. This approach is developed by inspecting the simulation IntBIIS. Also, existing approaches for simulation coupling and interoperability are examined. An example of such an approach is the High-level Architecture (HLA) as a current standard for simulation interoperability. Furthermore approaches on simulation modelling as well as simulation composition approaches are inspected. Additionally IntBIIS is analysed to gather insight about information that can support the extraction of smaller simulations out of a monolithic simulation. We also investigate already existing decoupling approach for this purpose.

This thesis is therefore organised as follows: First, Chapter 2 provides the foundations of modelling and simulation. Insight in the simulation of software systems and business processes is provided in Chapter 3. Chapter 4 provides and discusses work related to the topics in this thesis. Information supporting the extraction of models contained in monolithic computer simulations found by inspection of IntBIIS is given in Chapter 5. Chapter 6 describes the provided DSL to describe the coupling between simulations. Thereafter, the evaluation of our modularisation approach with the DSL is presented in Chapter 7. Finally, Chapter 8 provides a final conclusion and a presentation of future work in the context of this thesis.

2 Modelling and Simulation Foundation

This chapter provides foundations for topics required in this master thesis. (Meta)models and Model-Driven Software Development (MDSD) are introduced in Sec. 2.1. This introduction includes the definition of models, metamodels and metametamodels in Sec. 2.1.1. MDSD and Model-Based Software Development (MBSD) is described in Sec. 2.1.2. Also the terms of view-based modelling is described in Sec 2.1.3. One application of models is the description of the behaviour of real or virtual systems in a computer. This field of application is called simulation. Sec. 2.2 provides a introduction into computer simulations. Sec. 2.2.1 includes different approaches on how to model a simulation. One modelling approach is the discrete-event simulation. This approach is presented in more detail in Sec. 2.2.2. Multiple systems models can be used to create a single simulation. 2.2.3 defines two structures for the connection between the different models. A goal in the simulation community is to assemble a simulation of other simulations. A simulation consisting of other simulation is called a *modular simulation* in this thesis. Required capabilities of simulations and approaches to achieve this goal are introduced in 2.2.4. An approach to connect simulations with each other is introduced in 2.2.5. The description is followed by an introduction in approaches to describe modular simulations in Sec. 2.3.

2.1 Model Driven Software Development

Models are widely used to make information easier accessible or usable by abstracting certain features. For example, we use the model of military troops for tactical training (e.g. navy ships). In a tactical training, the represented navy ship is an abstraction of a real navy ship, the "original". However, this representation can have altered (e.g. size) or omitted (e.g. no interior) features. The definition of a model is given in the following section.

2.1.1 Models - Definition and Properties

Models are defined corresponding to the definition by Stachowiak et al. [5]. Here, models are defined as a formal representation of a real- or virtual world entity (**abstraction**). Furthermore,the models are created for a specific purpose. Also, they are only meaningful to their users as well as for specific operations and time-spans (**pragmatism**). For example, the model of a navy ship is only created for training exercises. It cannot be used for the transportation of persons. Models contain only a subset of attributes of the original to serve the intended purpose (**reduction**). In the navy ship example, the ship model contains only the appearance of real ships. However, the engine and exhausts are omitted due to their irrelevance for the given purpose. By this definition, models represent a particular set of features/attributes. A model instance is generated when values are assigned to the features/attributes of a model. For example the features of a ship could be a colour of the hull or the speed of the ship. In a model instance the value red can be assigned to the feature "colour" and 30 Knots to "speed" .

A model itself is described by another model - the so-called metamodel. The metamodel abstracts elements and properties of a model as well as the possible structure. Elements and structure are defined by relationships, constraints and modelling rules [6]. These definitions are one of four aspects needed to construct valid models. A model described by the metamodel is called an instance and conforms to the metamodel [3]. Völter et al. [6] defines the four aspects of valid model creation:

1. The **abstract syntax** describes the elements of the models and their relations, independent of their representation.
2. At least **concrete syntax** has to be provided to describe the representations of the abstract syntax.
3. The **static semantic** determines the criteria that qualify a model as well-formed through a set of rules and constraints not covered or described by the concrete syntax.
4. The **dynamic semantic** describes the meaning of the meta-model by means like, for example, natural-language.

A metamodel can be described by another model, which is then called a metametamodel. It is possible that another model can describe this model as well. However, in this thesis, we use the four-layered approach of the MOF of the Object Management Group (OMG) [7], where the metametamodel describes itself. The relation between these levels is depicted in 2.1. Models can be used as supporting- or central artefacts in software development. This topic is discussed in the following section.

2.1.2 Model Driven Software Development

Models are used in various fields of application in computer science. For example, in programming or software development. In programming, machine code is abstracted with the purpose of better understandability and easier manipulation. Assembler code can be seen as a abstraction of a programs binary code. Its purpose is to enable programmers to better read, understand and manipulate the contents of the code. Instances of these abstractions (i.e. the written source files) can be transformed back into the form they are abstracted from (i.e. the binary code). This transformation is done within the process of compilation.

In software development, models can be not only be used to abstract code but also other features of a software system. For example, the usage of the system or the behaviour of the internals of the system (e.g. communication between classes). In MBSD, these models are used for documentation and can be seen as secondary artefacts. Another use is to depict and display certain aspects of the development or the system (e.g. for internal-

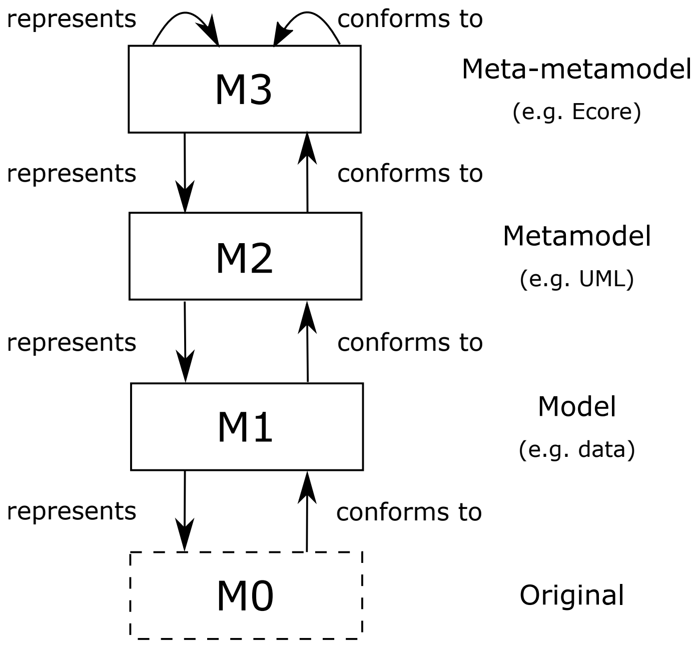


Figure 2.1: Model layers defined by in the MOF [3]

or stakeholder communication). Concerning the code-abstraction aspect and the use of models in software development the approach of MDSD was invented. In this approach, models are first class development artefacts [6]. Therefore, models are used for code generation or analysis and constitute a central part in development. It can be seen as another level added to the abstraction of machine code to better manage the complexity of source code [6]. Here, source-code is abstracted by models representing whole classes, interactions or even architectures of systems. This abstraction can be used to achieve advantages like increased development speed. Also, the use of MDSD promises better software quality through automate transformations of the model to code. Additionally, the different modelled aspects can be reused in other software systems like, for example, product lines [6]. Another benefit is the management of consistency in the software development. In MDSD, consistency between documentation and code can be achieved by the model-to-code transformation. Typically, all changes to the system are first made in the model when possible. The model-to-code transformation then provides these capabilities since all changes are already in the model.

2.1.3 View-Based Modelling

View-based modelling is used to provide a focused approach for stakeholders. Different views present aspects of the same model for different purposes. For example, the same model is used to express a components-based software system. However, the view of system architect only provides the model elements to display and manipulate the software's structure. On the other hand, the view specified for the deployer of the system only shows deployment relevant information like the hardware, on which the components can be deployed on [8]. Goldschmidt et al. [8] use the terms of *view type* and *view* to distinguish between the definition level and the instance level. Here, the definition level is specified by the *view type* and the instance level by the *view*.

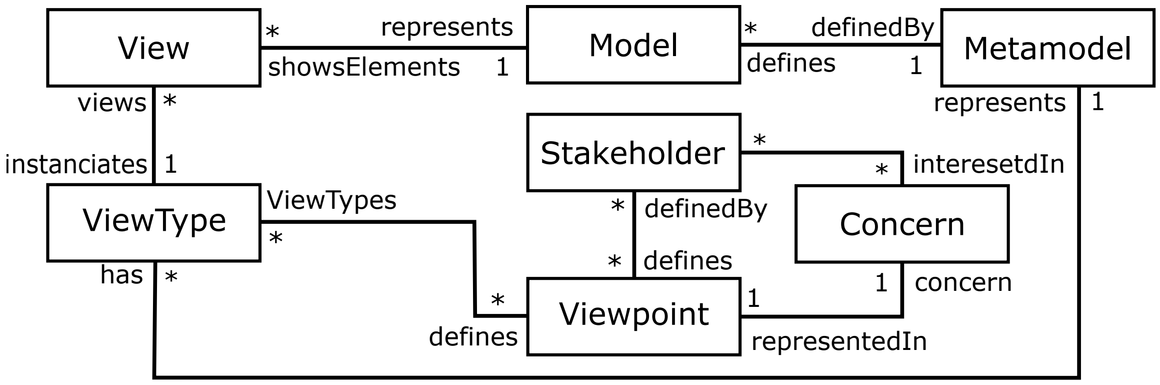


Figure 2.2: Terminology of view-based modelling according to Goldschmidt et al. [8]

View types define rules to structure views that can be created. Each view has to be created in accordance to these rules [8]. Burger [9] mentions that view types "can be interpreted as a metamodel for actual views" [9]. Goldschmidt et al. [8] defines view types by two aspects to provide a connection between view types and metamodels. The first aspect specifies that "a view-type defines a set of meta-classes whose instances a view can display" [8]. The second aspect is the definition states that a view type "defines a concrete syntax and a mapping of the abstract metamodel syntax" [8]. This definition targets the second level of the MOFs hierarchy depicted in Fig. 2.1. A view type can define several viewpoints. One or more stakeholders can be interested in several concerns. The view point expresses these concerns in view types [9]. For example, view points on a component-based software architecture can be concerned about system independent and system dependent specifics.

Views are instances of the view types[8]. Therefore they can be seen in a relation as parallel to those of models to metamodels. Thus, views correspond to the first level of the MOFs hierarchy [3]. Each view can have different properties. Those properties are described by Goldschmidt et al. in [8]. The resulting terminology of view-based modelling used by Goldschmidt et al. [8] is partially shown in Fig. 2.2

2.2 Software Based Simulation

In science and industry, cases exist, where data-gathering from a system or manipulation of a system is not possible, infeasible (e.g. high cost or low efficiency) or even dangerous. An example of systems that cannot be manipulated are cosmic systems. Also, a problem is the data-gathering of systems not existing at that time (e.g. software systems in design). Another example can be found in the running example of tactical training. The participation of every vehicle and person in the tactical training of one trainee would result in high costs.

The gathering of data despite these prior mentioned problems is desirable. Therefore, systems are described by models to enable their representation in a computer.

Systems are viewed as entities acting together or in dependence on each other to achieve a common goal [10]. A system is represented by mathematical expressions or logical relationships. These relationships describe the different assumptions defining the system [10]. The representation of a system is called a *simulation model.* If the simulation models of a simulation can be solved by means like algebra, calculus or statistics, exact results can be obtained [10]. The calculation of exact results can get harder or even impossible with rising model complexity. Computers are used to imitate (simulate) the behaviour of models to cope with this problem of rising complexity. In computer simulations, models are numerically evaluated over time [11]. Different modelling types can be used to provide the system structure and to represent the behaviour. Examples for those types are Petri nets, event relationship graphs or queueing networks [12]. This approach also poses the advantage to change or enhance the system preliminary without providing real resources. Also, problems in a systems representation can be found before they appear in reality [13].

Simulations have to represent time to model real-world or virtual systems. The representation of time enables the user to gather time-dependent data like time-related measures (e.g. response time of simulated software systems [3]). It is often also possible to manipulate time to advance it faster or slower than normal [13], so that the term *simulation time* has been introduced. This term signals the possible differences between real wall-clock-time and the time represented in a simulation. The state of the system is defined by a set of variables called *System State Variables* (hereafter called *state variables*) [10]. The system is described by these variables related to a specific time to provide all information necessary for a certain purpose of investigation [11]. By the representation of the system through variables, it is possible to manipulate the system and control its behaviour. Different approaches exist to represent time and state in a simulation. Because of the importance of these topics, they are introduced in the following section.

2.2.1 Simulation State and Approaches for Simulation Modelling

One modelling factor of a simulation is the representation of states and the transition between them. The state is represented through attributes which are necessary to describe the simulation at a certain point in time [10]. Simulations can be either modelled continuously, discretely or as a combination of both [12].

The state of continuous systems is described by variables modelled through explicit functional forms, difference- or differential equations [13]. These forms enable a continuous change of the state of a system over time [10]. In discrete-time simulations, changes to the variables are executed instantaneously at defined discrete points in time [12]. Rules have to be specified for how and when time advances [12]. Lots of systems are not purely discrete or continuous. Therefore a combined approach can be used. Here the interaction of both approaches is one of the main concerns [13].

Multiple approaches of simulation modelling emerged for continuous or discrete models. Each approach is designed with its distinct goal. In this thesis, mainly discrete approaches are presented. Additionally only the approach of Discrete-Event Simulation (DES) is discussed in more depth. The system's behaviour in the classical discrete modelling world-view is modelled in a top-down approach. Entities in a system (e.g. a paramedic or a vehicle) are explicitly represented by attributes defining their information [13]. The predefined flow of each entity from state to state through the system is modelled in this world-view[13]. Different views on the system and how to model them emerged in this world-view. An example is the event orientation, in which the system state is changed by a series of instantaneous events where no time passes while executing the event [10]. Another approach is process orientation where passive entities flow through the system by multiple process steps [12]. In contrast to events, time may pass in a process step. Contrary to the top-down view on a system, agent-based modelling uses a bottom-up approach to build the system. Agents are placed in the system, each specified with its behaviour. The agents interact with each other as defined by their behaviour. The interaction of all agents creates the behaviour of the complete system [12]. The approach of event orientation is supposed to be very flexible. Also, it is used in the motivating example simulation IntBIIS and the evaluation system. Therefore, the DES approach is introduced in the following Sec. 2.2.2.

2.2.2 Discrete-Event Simulation

In DES, the system state (i.e. its variables) changes at discrete points in time by the occurrence of instantaneous events [13]. Each event is specified by the modeller and defines its influence on various variables in the system [10]. A discrete-event simulation consists of multiple parts responsible for event management, scheduling, and the resulting simulation. Besides the variables representing the system state, the simulation clock variable represents the current simulation time. For organisation of the events an event-list is employed. It contains the next time each event type occurs. A timing routine selects the next event from the event-list and executes its event-routine. An important aspect of discrete-event simulation is that no time advances while an event is executed [12]. Thus, the timing-routine advances simulation time by updating the simulation clock only after the event-execution is completed. In some modelled activities, time advancement during an event may be allowed or desired. Start and end events have to be defined in such a case [12]. Time can be commonly advanced by using the "next-event time advance" mechanism. At the start of the simulation, the simulation times of future events are determined and the simulation clock is set to zero. Through the execution of the simulation, the timing-routine continuously selects the next event for execution [10]. Whenever the event-routine is called, three actions are possible[10].

1. Update of the system state, i.e. changing its variables
2. Gathering of information about the simulation
3. Extension of the event-list by newly generated times of event-occurrences.

Time advances to the determined time of the next event by updating the simulation clock after the event-routine is finished [10]. This procedure is repeated until a stopping condition is found to be true [10].

2.2.3 Structures of Multi-System Simulations

Some real-world or virtual systems consist of collections of sub-systems rather than a single system. Each of these sub-systems has to be modelled to be simulated as a whole system. Thus each simulation can be seen as a separate (sub-)simulation. For example, a military training simulation can consist of the distinct models for the air-force, the nautical-force and ground-force. Different simulations have to be combined to communicate and interact with each other to produce a simulation of their joined behaviour. Approaches like isolated simulations, co-simulations or integrated simulations are applied to achieve such a combination [14]. In the isolated simulation approach, each simulation is separately executed. Only their results are exchanged after their execution [14]. Integrated simulations are used to let simulations interact with each other, even when they are originally not intended to do so [15]. In co-simulation, multiple simulations interact through a common coordinating entity. This entity provides communication and data-exchange functionality for the simulations.

Software systems or simulations can be designed as either monolithic or modular structures. Monolithic architectures in software engineering contain all capabilities and responsibilities of a software system. The single parts realising the capabilities are tightly coupled through a uniquely developed structure [16]. In the context of simulation, this corresponds to a structure where each simulation model representing a system in a multi-system simulation is located in one unique structure. This approach poses drawbacks also known from monolithic structures in software development. Large monolithic simulation models suffer from low and costly extensibility along with missing reusability in other simulations [17]. Furthermore problem in the usage of monolithic simulations is, that the underlying simulation models are specialised to the monolithic structure of the simulation. If a (sub-) model of the simulation shall be reused, it has to be prior extracted. This requires the understanding of the structure itself. The extraction can then be time extensive and costly so that it can be more efficient to implement the simulation anew [1].

These drawbacks can be approached by the method of modularisation [2] as used in software engineering. In modularisation, a problem is separated in several smaller problems. This approach enables a better understanding of a single problem. Another benefit of modular structures is shown in the reusability of their modules. Reusability is beneficial because of reduced development time and a wider design-alternative-space. The latter is provided by the possible combination of off-the-shelve components and self-created components [18]. The simulation community also researched this technique on their own, which is called *composability.* Composability aims to reuse simulations to describe different larger simulations [18]. This approach is similar to the use of components in component-based software design. However, software components themselves are designed to provide reusable functionality. Also, they are treated as black-boxes (i.e. the developers cannot see their internals) [3]. On the other hand, simulations often contain specific semantic information [19]. Because semantics are difficult to capture, two simulations can have different semantics for the same information. The difference between semantics introduces difficulties in the interaction. Also, simulation composability is said to pose additional difficulties like time management or event generation [18]. However, it is still desired to compose simulations of multiple reusable simulations or models. We call simulations consisting of other simulations or models "composed" or "modular" simulations. In the following, composability and its requirements are further discussed.

2.2.4 Composability of Simulations

A goal of the modular simulation is to select, assemble and re-assemble simulations like components in the component-based software design [20]. In the simulation community, the terms of composability and interoperability are introduced. Composability defines the capability to select, assemble and reassemble different larger simulations to satisfy specific requirements. Prior selected simulation components should be (re-)combinable without larger effort to meet different needs [20]. Interoperability, on the other hand, is concerned about the consistent and meaningful collaboration of multiple simulations to simulate a scenario [21]. The focus lies on the exchange of data and information between the simulations. An assembly of simulations to achieve one goal can be interoperable. However, at the same time, this assembly is not necessarily composable when the single simulations cannot be reused in another context [20]. Interoperability is a prerequisite for achieving composability. However, interoperability alone is not sufficient enough [20]. This problem originates in the reason that interoperability, in the first place, describes the combination of multiple simulations. Composability, on the other hand, is focused on the model and that they fit together in a meaningful way. Interoperability approaches use some form of protocol or coordinating entity to enable interaction between simulations. These simulations must be fitted to the capabilities and structures of the interoperability approach. Thus, the fitted simulations cannot be independently recombined to other larger simulations without further effort [20]. With these definitions Page and Briggs [22] propose three dimensions, or views, to categorise the interconnection of simulations. The composability-view is specified on the models of different simulations. Composability, in this dimension, is given when the objectives and assumptions of the models are properly aligned [22]. Interoperability defines the view on implementation specifics of the model like data type consistency. The last dimension is integrability. This view is concerned with the physical environment of the simulation. In this thesis, only the first two levels are of interest. An attempt to bridge the gap of composability and interoperability is proposed by Tolk et al. [23] to combine implementation focused interoperability approaches with conceptual models. It is structured in reference to the "Levels of Information Systems' Interoperability" [24]. A conceptual model describes the aspects of a system to be represented in a model. These aspects include the limiting assumptions on the model and other assumed capabilities to satisfy a special purpose [15]. As a result, Tolk et al. [23] defines the Levels of Conceptual Interoperability Model (LCIM) [23]. This model consists of five Levels. These levels are defined as follows:

- **Level 0 - System Specific Data**: There exists no interoperability between two systems. The data is system specific and only proprietary usable and not shared. Data only usable and identifiable by the system (e.g. undocumented csv-tables or hard-coded data) is such an example. Thus the data and functionality of the system are only known to already familiar users.

- **Level 1 - Documented Data**: All data is known and documented in a consistent way using a defined protocol. With such documentation and access through interfaces, data can be mapped to external sources.
- **Level 2 - Aligned Static data**: The meaning of the data is unambiguously described and documented with a common reference model based on a common ontology. This level targets the solution of conflicts created by merging different data sources. These conflicts are specified in four conflict classes. These classes are namely semantic, descriptive, heterogeneous and structural conflicts. Semantic conflicts describe that two concepts of the schemata of the models/simulations do not match exactly. Descriptive conflicts are concerned with, for example, different names, attributes for the same concept or synonyms. In structural conflicts, different structures describe the same concept in different models. The last conflict class are heterogeneous conflicts. These conflicts describe different methodologies used to describe concepts.
- **Level 3 - Aligned Dynamic Data**: Additionally to the data in level 2, the behaviour of a single component, including the use of data and resulting state changes, is made visible.
- **Level 4 - Harmonized Data**: For each component, the conceptual model has to be made available. The availability of the conceptual models allows a check for semantic consistency.

The LCIM can be used to enhance the definition of components and their specifications. This improved definition is supposed to achieve easier and better-defined interoperability. Wang et al. [25] enhances the LCIM to incorporate the notion of integrability, interoperability and composability prior mentioned. Seven levels of interoperability (0-6) are proposed as a result. Level 1 is is related to the "physical and technical connections" [17] between the system. The simulation and implementation details are the predominant aspects in levels 2 - 4. These level include the exchange of data. Levels 5 and 6 are aligned with the composability dimension in [22] and refer to the alignment of models. Bartholet et al. [18] points out that true composability is not achieved yet and is currently mostly theoretical. However, progress in the field of interoperability is made by standards like the high-level architecture and the base object model. Thus, co-simulation and the high-level architecture are introduced in-depth in the following section.

2.2.5 Co-Simulation and the High-Level Architecture

An approach to provide capabilities for interaction has to be used to couple multiple simulations. Co-simulation is a principle to combine multiple simulations, the so-called simulation units, to a larger simulation. Running those units on different computers is possible. The simulation units can be seen, as black-box components with different external visible information [26]. An entity responsible for the management and exchange of information between the simulation units is used. It contains capabilities like management of time or the movement of data between the units [26]. This entity is called an coordinator. The result of the connection of multiple simulation units and a coordinator is called co-simulation. The co-simulation itself can be coupled with other simulation units or co-simulations through another coordinator. This results in hierarchical structures [26].

A realisation of this principle is the HLA. Dahmann et al. [27] introduces the HLA as an approach for discrete-event co-simulation, standardized by the Institute of Electrical and Electronics Engineers (IEEE). Based on concepts of Distributed Interactive Simulation (DIS) [28], the HLA provides an architecture consisting of three major parts. These parts are major functional elements, interfaces, and design rules. All of those parts are developed to be feasible for all simulation applications. The HLA specification provides rules and a common framework for the definition of specific system architectures [27]. With this approach, reuse and interoperability of simulations in 2.2.4 is achieved. HLA does not dictate an implementation or programming language due to its description as framework and protocols. The HLA consists of three functional components resulting in a co-simulation, or in HLA called "federation" [27] .

1. The (sub-)simulations are called **federates** and are assembled to a **federation**. Passive data collection and evaluation is enabled through monitors and loggers. They can be applied like general federates. The only requirement a federate must meet is the realisation of the HLA capabilities needed. This prerequisite includes capabilities for the interaction between objects of different simulations.

2. The **Run-time Infrastructure (RTI)** provides services to support the federates in their "federate-to-federate" communication. A second function is the provision of the federation management support. This management includes capabilities like federation management or time management. Only indirect communication between federates over the RTI exists in the HLA.

3. The **runtime interface specification** is a set of implementation and object model independent specifications. They define how federates should interact with the RTI. This definition includes the way to invoke RTI services and how to respond to requests from the RTI.

For the formal definition of the HLA, three components are specified. Furthermore, the HLA itself is formally defined by the *runtime interface specification*, the Object-model Template (OMT) and the HLA rules [27]. The **federate interface specification** describes the services provided by the RTI. This specification subsumes the services in 6 different management categories. Following a short description of each category is given. They are taken from [29], where the whole specification can be read.

- **Federation management** defines capabilities for the management of federation executions, including the creation, modification and deletion. Functionality for the definition of synchronisation points and to save and restore federation states are also included.

- **Declaration management** realises functionality for declaring what information and interactions a federate provides or requires. This declaration must be made before object instances can be registered or manipulated.

- **Object management** specifies how object instances are registered, modified or deleted. Also, the sending and receiving of interactions are defined.
- Each instance of an object can be shared among federates. The **ownership management** handles the ownership status of attributes belonging to an instance. This status allows cooperative modelling of a given object instance in a federation.
- **Time management** provides services and capabilities for coordinating the time between the federates. The time in HLA is seen as a single time axis. The simulation specific time is coordinated on the basis of this axis. All messages and interactions between simulations are coordinated to this single time specification.
- **Data distribution management** may be used to reduce the traffic between the RTI and a federate by defining a filter for irrelevant data.

The OMT is a standard form to document the *FOM* and the Simulation Object Model (SOM) consistently. The SOM defines the simulation data, thus specifies all the data which can be possibly exchanged [27]. The Federate Object Model (FOM), on the other hand, defines which data is shared in the federation [27]. All data used in the FOM is provided in the SOMs of the participating simulations in a federation. This separation facilitates reuse because the SOM is generally valid for its corresponding simulation. Thus, each SOM can be used in the creation of multiple federations [27]. The HLA structures the data of simulations and interactions in an object-oriented principle. Thus, information is structured as classes with attributes and interactions with parameters. Here, classes and attributes are entities persisting over time. Interactions with parameters only exist in one instance of time. The structure of information is classically defined in tabulated form and specified in [30]. However, approaches like the *poRTIco* project [31] use the Extensible Markup Language (XML) for definition of the OMT. In the following, some capabilities of the OMT are further described and taken from [30]. Multiple kinds of information are defined.

- Objects classes
- Attributes of object classes
- Data types
- Interaction classes
- Parameters of interaction classes

The HLA facilitates an inheritance scheme where the child class inherits all information (e.g. attributes or parameters) from its parent class. The inheritance is used for object classes as well as for interaction classes. Object classes must inherit from the root class *HLAobjectRoot*. This class provides additional attributes required to process object classes and attributes by the RTI. The same concept exists by the *HLAinteractionRoot* for interaction classes. Each attribute in an object class has to incorporate a prior specified data type. Data types are defined by a name, size in bits, an interpretation, the endianness and a text field declaring the encoding. Data types are HLA federation specific. Similar to object classes, interaction classes are complemented with parameters, containing a data type and further HLA specific information [30]. The **HLA Rules** provide basic guidelines for federations and single federates. An example of federation rules is that all object representation takes place in the federates and not in the RTI. Another example is that all federates must provide their public information in their SOM using the OMT [32]. Fig. 2.3 depicts the parts of the HLA with functional federates. To enable federates to communicate with

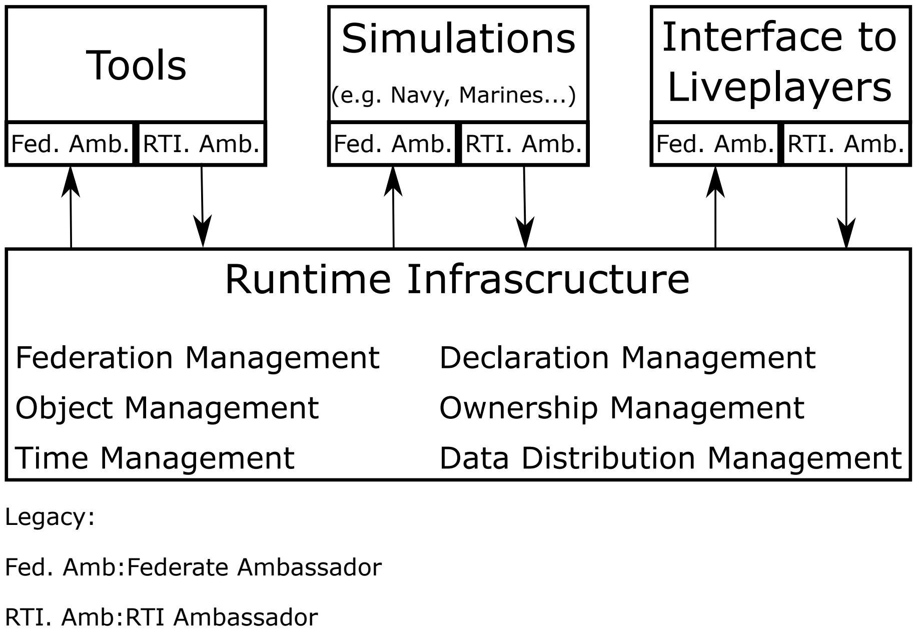


Figure 2.3: View of HLA components [27]

the RTI, implementation entities called *ambassadors* are commonly used. Two types of ambassadors exist. The *RTI Ambassador* provides the capabilities to call federate-initiated methods of the RTI (e.g to check if messages are available for a federate). The *Federate Ambassador* handles callbacks from the RTI to the federate [21].

Aside from the HLA other approaches for interoperability and composability can exist. A standard definition of the data of a model, a so-called object model, is beneficial to support a composition approach [15]. Simulation can be easier composed if they comply to such a model. The OMT describes such an object model, but only describes data. The Base Object Model (BOM) standard takes this idea further. A BOM includes the HLA OMT and complements it with additional information like patterns of interplay or general data definitions [33].

The BOM is a standard supported by the Simulation Interoperability Standards Organisation (SISO). The BOM is meant to facilitate interoperability, reuse and composability of simulations [34]. BOMs document data structured as classes with attributes and interactions with parameters. This structure is based on the HLA OMT specification described in Sec. 2.2.5. Unlike classical FOMs or SOMs, the BOMs describe a complete "model of a simulation interplay activity" [33]. BOMs complement the FOMs and SOMs with meta-data for this purpose. The meta-data can include information like requirements, the conceptual model, sequence diagrams or description of the intended domain [33]. They are designed as small compositional units. Each unit describes a single aspect rather than a complete simulation like the FOM or SOM [33]. Thus a BOM consists of a conceptual model, model mappings and the HLA Object Model [34].

2.3 Approaches for the Description of Simulation Composition

As mentioned in Sec. 2.2.3, modular simulations consist of multiple (sub-)simulations or their models. The description of simulation composition can be useful due to the higher level of abstraction and their benefits mentioned in Sec. 2.1. One example is the validation of consistency between the models. Also, it is possible to generate code out of a description. Furthermore, descriptions can be reused in other modular simulations. This section provides an overview of some approaches to describe modular simulations and simulation composition.

2.3.1 CODES

The CODES approach by Teo and Szabo [35] focuses on the abilities to discover and reuse components. It provides the capability to validate the syntactical and semantic composability of the selected components. Models and their composition are described by an ontology to facilitate syntactical composability [35]. Each model in this ontology consists of black-box components with in-channels and out-channels which describe the provision or requirement of certain data. In addition to the in-channels and out-channels, the component contains descriptions of attributes and their behaviour. The syntactic composability is enforced by rules described with a composition grammar [36]. Semantic composability is checked when the out-channel of one component is connected to the in-channel of another other. Sent and received data are annotated with attributes like origin, destination, time, type and range. An algorithm checks if the connection between in- and out-channels is valid.

2.3.2 Discrete Event Systems Specification

The Discrete Event System Specification (DEVS) approach is proposed by Zeigler [37] and uses formal definitions to specify DES models. Developers can describe the states of a model and transitions between them by this formalism. A DEVS model is defined by seven parameters to provide this capability. The parameters are [37]:

- A set of input events
- A set of output events
- A set of sequential states
- The initial state taken from the set of sequential states

- A time advancement function is used to set the maximal duration the simulation can execute one state
- The internal transition function. This function defines the change of the system's internals when the maximal duration of a state is exceeded
- In the external transition function, the transition behaviour is described when encountering an event
- An output function, determining the output event generation

Zeigler [37] enhances this definition by using sub-components to provide coupled DEVS. Coupled DEVS utilises a hierarchical approach. It consists of atomic DEVS models or other coupled DEVS models. An atomic DEVS model is specified with the structure shown above. Another 8-tuple describes the coupled DEVS, which contains input and output events as well. Additionally, the names of the subcomponents and the used atomic or coupled DEVS models are specified. External input couplings define possible points of interactions between the coupled DEVS and other DEVS models. Internal couplings determine how the contained DEVS models are connected. The external output coupling function determines the output of a coupled DEVS. If two events coincide, a tie-breaking selector function decides which event is selected. Zeigler and Lee [38] also propose an extension of DEVS to be able to describe the capabilities of HLA. The time advance is based on events rather than discrete steps. Thus, events occur only by significant changes in input, state or variables. Zeigler and Lee [38] also inspect the properties and specifics of the term "significant change". An application of the DEVS formalism (more concrete parallel DEVS by Chow et al. [39]) is used by Röhl and Uhrmacher [40]. To achieve composition, they use the notion of components and interfaces as in [41]. The description of these components must be delivered in XML format. The components define public ports. Components can be connected to each other with these ports. Each component and interface is defined in XML format with a specific syntax. A composition is defined by use of a new component. This component defines the ports and naming of each component as well as the connections between them. When the components and the connections are specified, the components internal behaviour needs to be specified with the parallel DEVS formalism. The DEVS model must provide the ports defined in the corresponding component definition. If this structure and requirements are fulfilled, the XML-components along with their DEVS model are transformed into executable models usable by the simulation tool James II [42].

2.3.3 Ptolemy

Ptolemy II is a framework to model and compose hierarchical and heterogeneous simulation models. It is described by Eker et al. [43]. Ptolemy II is based on actor-oriented models. Each actor is a component shares data with each other and is executed concurrently [44]. The communication between actors is defined by the use of ports and passing messages between them. Ptolemy II applies a hierarchical approach. Therefore atomic and composite actors exist [43]. Atomic actors are internally not defined as an actor model. Composite actors are composed of other actors as similar to composite DEVS. Only direct communication between actors in composite components is allowed. Communication to actors outside the composite component is only possible through component ports on the composite components border [44]. On each hierarchical level, composite actors are seen as black box. Therefore, they are treated like atomic actors. Ptolemy II provides implementations with different Model of Computation (MoC) to describe the computational aspects in a component [44]. Each MoC provides the scheduling and communication details for actors. For example, MoC describe the data flow and used time model (e.g. continuous or discrete event). However, this approach can pose the problem that actors with different MoCs cannot process the date of each other. Special actors resolve these problems by translation of data from one MoC to the other [43]. The hierarchical description by actors and the definition of their behaviour by MoCs allows not only modelling but also analysis of the resulting simulation.

2.3.4 OMNeT++

OMNeT++ is an environment to describe and simulate DES and is presented by Varga and Hornig [45]. Its original purpose is to model "communication networks, multiprocessors and other distributed or parallel systems" [45]. However, its general modelling approach allows specification of other distributed simulations. OMNeT++ is written in the programming language C++, but third-party support for the languages Java and C# is available. The basic structure consists of simple and compound modules. Compound modules can be seen analogue to couple DEVS models. Therefore they consist of other compound or simple modules. In their use, compound modules cannot be distinguished from simple modules. The functionality of a compound module can be implemented by a simple module or the other way around. Each module sends messages to other modules by the use of gates. These gates constitute the input and output interfaces of the module [46]. Messages contain data and other supplied attributes like timestamps and are sent between the gates. Therefore different gates are connected with connections. These connections specify which input gate of one module connects the output gate of another module [45]. Through the use of compound modules, OMNeT++ facilitates a hierarchical approach. However, modules of a compound module can only communicate with external modules through the use of a gate at the compound modules border. Thus, no internal module can directly communicate with an outer module [45]. OMNeT++ provides a graphical editor along with an approach to also simulate the described models.

3 Simulation of Software Systems and Business Processes Foundations

Quality characteristics of software systems can determine their success or failure. Quality characteristics are attributes of a system, relevant to a certain stakeholder. Quality characteristics can be categorized in external and internal characteristics according to McConnell [47]. External quality characteristics are those relevant and experienced by the user like the capability of a system to provide its functionality when required (i.e. reliability). Internal quality characteristics are relevant for the developer like the difficulty on how the software can be modified to provide or change capabilities (i.e. maintainability) [47]. Indirect influences, such as unaccounted business processes, can unexpectedly influence the quality characteristics of a software system as well. Thus, early approximations of characteristics and impacts of other influence sources like business processes are desirable. The software system in question can be simulated to obtain early approximations of quality metrics. For this purpose, the software system has to be modelled. The model of a system has to includes all influence sources of the system like the hardware environment or business processes. Another aspect that can be modelled is the structure of the software system.

To provide the capability to model a software system, a DSLs can be used. These languages are constructed to be able to describe a certain domain. DSLs are described in Sec. 3.1. An category of DSLs are Architecture Description Language (ADL)s. These languages are used to describe the architecture of systems and are also described in Sec. 3.1. The foundation of component-based software system simulation is provided in Sec. 3.2. This foundation includes the definition of components in Sec. 3.2.1. This section is followed by the description of Palladio, which is a concrete example of the provision of a component-based software simulation. The simulation engine EventSim can be used by Palladio to execute the models defined by Palladio. This simulation engine is described in Sec. 3.2.3. Furthermore, IntBIIS is described as extension of Palladio to discover the mutual influences between business processes and software systems in Sec. 3.2.3

3.1 Domain-Specific Languages and Architecture Description Languages

A DSL provides the capabilities to describe certain domains. DSLs help developers by improving their productivity as well as enable communication with domain experts [48]. This improvement is achieved by the limited expressiveness of the DSL and a focus on a certain small domain [48]. Thus, domain experts do not have to understand general purpose code of programming languages like C++ or Java. This approach allows the focus on the concepts and the description of the domain instead of the need to learn a general purpose language. Moreover, domain experts can work with structures and terms they are familiar with. An example of a DSL can be seen in the Sprat approach[49]. This approach enables a scientist to describe ecosystems by an ecosystem DSL. The sprat DSL provides the capabilities to specify ecosystems and species with a restricted set of terms used by the domain expert[49]. Another example of a DSL is the Palladio DSL. It is designed to describe component-based software architectures to predict quality characteristics [3]. The Palladio approach provides a DSL to describe the architecture of a system. Such a DSL is also called an ADL. ADLs are formal languages with the purpose of representing the architecture of software-intensive systems [50]. Clemens et al. [50] surveys common ADLs for their capabilities and presents multiple properties a language should have to be an ADL. Some of these capabilities are basic capabilities like abstraction, communication and integrity. Also, the provision of a mapping of behaviour to different architectures can be a capability of ADLs [51]. Furthermore, ADLs should enable the user to create, refine or validate software architectures and also have to provide the elements to describe a software architecture [50].

Considering simulations, McKenzie et al. [52] points out that ADLs are "underutilised" in the modelling and simulation community. This conclusion is reached after surveys of usefulness and effectiveness of general-purpose ADLs to describe and analyse simulation systems. For this purpose, two ADLs are chosen to describe two simulation systems in connection with a HLA approach. McKenzie et al. [52] conclude that the simulation community would benefit from the formal approach of ADLs. Four benefits are named [52]:

1. **Robustness** of simulation architectures (i.e. improved reliability, stability and extensibility) can be reached by the application of the software architecture disciple.

2. Reaching conceptual **composability** is possible by explicit notations of simulation components, connectors and their interfaces. This approach can help simulation architects to see the composition aspect of simulation systems.

3. Better **knowledge transfer** to study good or bad design. Also, ADL descriptions could be a good entry point for simulation architects to become familiar with the system.

4. **Risk reduction** by revealing key aspects of the simulation system and analysing them to find possible problems.

3.2 Component-based Software Simulation

Collection of software quality measures is only possible after certain points in the development process. Measures can only be gathered when testable pieces of software are created in software development without simulation. Thus, parts of the system are already in the creation- and past the design process. It is even possible to that certain measures can only be obtained during the run-time of the system (i.e. after its deployment). The ability to collect data only in a late phase of development poses a problem in the detection of design mistakes. Much time and effort are already invested into design decisions, possibly later found to have a negative impact on the resulting system [3]. In other engineering areas, simulation is a common practice to gather information about systems without building them first [3]. In software engineering, simulation can be used to obtain an estimation of software systems' properties in earlier phases of the development (e.g. of performance). Early estimations enable developers to react to negative influences in the design of software. An early reaction to negative impacts is important because "errors are more expensive the later they are removed" [53]. To be able to simulate the software system, it has to be modelled. The model has to include all factors influencing the software systems. Examples are the architecture, the use of the system and the hardware. With this information, it is possible to gather values for characteristics of the system. The Palladio approach provides capabilities for modelling and simulation of software systems for exactly this purpose. It provides a DSL to model the architectural domain of component-based software systems. The relevant terms of software components has to be introduced due to their usage in the Palladio approach.

3.2.1 Components

Software components are used in software development to build and structure software systems. They are supposed to be used and reused by third parties and to provide the ability to be independently deployable [41]. Thus, the intent of components is for developers to avoid the need to create every part of the software by themselves. This approach enables the possibility to buy readily available software components created by a third party "off-the-shelf". The component approach provides benefits in the field of system development and maintenance. Thus, components provide possibilities for significant savings in time. These saving can be motivated by teams that are creating systems and are not experienced in the domain of a certain functionality [41]. These teams can obtain components for these functionalities. The development of functionality includes high upfront expertise gathering. This upfront gathering of expertise results in higher cost compared to developers with domain experience. Thus, buying a component in this case, avoids the gathering of expertise which results in saved time and with that consequently costs. Also, it can be expected that a component, created by a team specialised in its functionality is highly reliable. Thus, this implies that functionality, implemented by developers with less expertise in the domain, is inferior to a component of a specialised team [41]. The benefit of increased maintainability is reasonable because components are directly maintained and kept up-to-date by its third-party developers. Therefore, the correct implementation of the functionality is outsourced by its users. To facilitate the compositional aspects of components, they have to be contractually specified and self-contained [41]. This specification is realised by the use of one or more interfaces. An interface is understood as "abstract description of units of software" [41] which is defined as "points of interaction between components" [3]. Interfaces specify what a component requires and provides. Pre-conditions and post-conditions with functional but also quality concerns can be stated [41]. Other attributes of components are their black-box nature and their non-observable external states [41]. The former implies that no internal information is accessible by the user except through the defined interfaces. This implication includes that especially no source-code or internal behaviour is visible [3]. A non-observable external state implies that copies of a component can exist but that they are indistinguishable from the original [41]. The feature of non-observable states makes it possible to use one component more than once (e.g. in a load-balancing scheme). Besides their benefits, components also have drawbacks. They must be specified to a certain component approach, resulting in a specialisation of the assembled system. Also two to three times higher upfront cost can be expected for design and specification due to the detailed interface design and good quality attributes. Amortization of these costs is possible if the component is reused more often [3].

3.2.2 The Palladio Approach

The Palladio approach is developed with the goal to gather approximations of quality metrics in early design phases. Examples for quality metrics are performance or reliability. The gathering of quality metrics is achieved by using the DSL of Palladio to model component-based software systems. This DSL allows the user to describe software architectures and their contained knowledge relevant for quality prediction [3]. Palladio can be utilised with these models to simulate the modelled system to gather resulting quality attributes. The architectural domain provides performance and reliability characteristics. Performance covers resource efficiency and timing behaviour indicated by the following measures [3]:

- **Response time** describes the time between passing a request to a system and the return of its computed response.
- **Throughput** measures the units of work that can be done by a system in a unit of time
- **Utilization** indicates the load of a resource over time in relation to the maximal load it can process per time unit.

Reliability in Palladio is indicated by the measures of probability of failure on demand and the failure rate of the system. The former describes the likelihood that a failure occurs in the system when it is used. The latter denotes the rate of failure occurrence in the system [3]. To capture the system properties, which are relevant to performance and reliability, not only the architecture has to be specified but details about the execution environment and the usage profile as well. The generated measures can be used to analyse design alternatives for their impact in performance or reliability or trade-off between each [3]. As an example, two similar components A and B can be used in the system. The system is modelled and simulated with component "A" and also with component "B" to examine if one component provides an advantage in performance or reliability. The result can be surveyed for positive or negative impacts. Through the response time and resource load, problems like bottlenecks can be found [3]. The supplementation of hardware and usage specifications allows to inspect their impact on the system. This insight can be granted in other ways only at runtime when the system is deployed on the hardware and in operation. For example, it is possible to find that the system does not achieve the requirement of a given response time when using a certain hardware. Also, the load generated by users can pose an influencing factor of the success of a system. A component can prove valid when used by ten users, but shows decreased quality when used by 1000 users. Thus the design alternative of a load balancer can be evaluated to remedy this problem [3].

Palladio specifies three viewpoints and four developer roles responsible for modelling the architecture, execution environment and usage profile of the system. They are used to produce a detailed model of the system and also separate the development process into collaborating units. The viewpoints contain different view types and distinguishing between system-independent and system-dependent properties. The first viewpoint provides building blocks like components or hardware specifics which are possibly reusable in other models and systems. The second viewpoint is related to system-specific properties like the assembly of multiple components to a system or the assertion from components to hardware. Reussner et al.[3] describes the viewpoints, view-types and their connection to the roles as follows.

1. The **structural viewpoint** contains the system-independent repository and the system-dependent assembly view-type. The repository contains components, interfaces and data types to create the component-based software architecture. Its content is created by either a **component developer** by direct specification and implementation or a *software architect* through specification of new interfaces required for assembly. In the assembly view-type, components are connected(assembled) by the component developer to composite components or by the system architect to entire systems.

2. With the **behavioural viewpoint** the behaviour in a component, between components and the usage of systems is modelled. The component developer uses Service Effect Specification (SEFF)s to model the intra-component behaviour, which defines the flow of a call through the internals of the component. These SEFFs can be enriched with resource demands for actions and calculations, resulting in Resource Demanding Service Effect Specification (RDSEFF). The **Domain Expert** models the behaviour between the components as well as the usage of the system by its users.

3. The **deployment viewpoint** captures the hardware aspects of the system. The **system deployer** models the system-independent resource environment with the possible or available hardware. This environment is represented by resource containers which can be seen as individual computers each consisting of one or more resources. Currently, processing units (i.e. CPU) and storage (i.e. HDD) are available. When using multiple resource containers, linking resources represent the connection between the resource containers. Components of systems are allocated to those resource-containers in the system-dependent allocation view-type by the system deployer.

4. The **decision viewpoint** spans above all viewpoints. Different design decisions are specified. If the design is changed, violations of a certain decision can be discovered.

The simulation of the modelled system by the different roles can be conducted by the use of multiple simulators available for Palladio. One of these simulators is *SimuCom*, which uses the process interaction technique as well as an event-based technique for different parts of the simulation. It generates code of the provided model through model-to-code transformations. In the process interaction technique, each generated user is mapped to a thread provided by the operating system. This mapping results in performance drawbacks when many users have to be spawned. With this problem in mind, Palladio can be used with the *EventSim* simulator [54]. This simulator employs the event-scheduling technique to cope with higher workload intensities [3]. This simulator is explained in more detail in Sec. 3.2.3. Another simulator usable by Palladio is *SimuLizar*, which enables *Palladio* to model and analyse self-adaptations. The analysis of self-adaptation is achieved through the capability to specify monitoring annotations and a simulation-based solver which interprets the provided models [3]. The schematic structure of the Palladio simulation environment used by *EventSim* and *SimuCom* is shown in Fig. 3.1. The *Abstract SimEngine* constitutes the lowest level of the structure and provides general

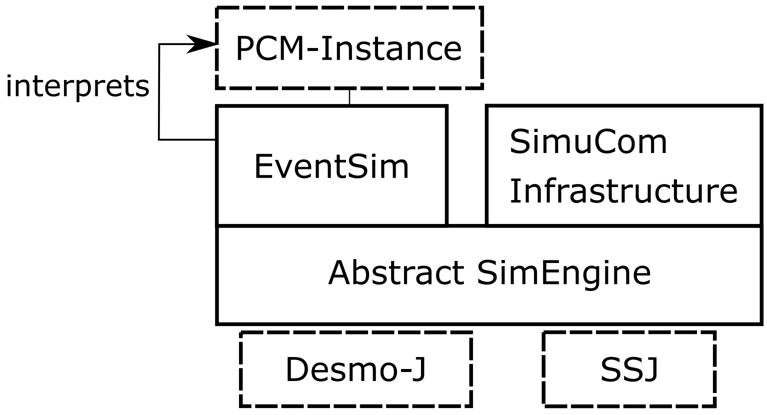


Figure 3.1: Simulator Environment of EventSim and SimuCom

simulation capabilities like an implementation of the process technique or functionality for event-based simulation [54]. Libraries like *Desmo-J* [55] or *SSJ* [56] typically provide these functionalities [54]. The *Abstract SimEngine* was formerly included in *SimuCom* but has been factored out to provide its capabilities to *EventSim* [54]. *SimuCom* and *EventSim* constitute another layer and realise a simulation approach. The last layer is represented by a Palladio component model instance which is interpreted by a supported simulator.

3.2.3 EventSim

To compute performance metrics by use of Palladio models, the discrete-event simulator *SimuCom* is considered as the reference simulator of *Palladio* simulator [3]. It realises the process-interaction simulation technique where every simulated user is mapped to a thread of the operating system. This technique, however, experiences performance drawbacks which are amplified through the use of Java-Threads when complex simulation models (i.e. many users) are processed [54]. These drawbacks result from the "inability to represent simulation processes efficiently" of Java [54]. *EventSim* simulator was developed with the goal to provide an alternative to *SimuCom* and to cope with these problems. It uses the event-scheduling simulation technique explained in 2.2.2. The relevant system parts are realised as entities. Thus, the users of the simulated system (user entity) invoking system requests (request entity) and having demands on resources like processing or storage (resource entity) are defined [54]. Their behaviour is modelled through chains of actions. Therefore, usage scenarios for user entities and RDSEFFs for request entities are used. Every entity is responsible for simulating their behaviour by traversing the action-chain and, in each traversal step, executing the action [54]. Contrary to *SimuCom*, *EventSim* interprets the loaded model rather than generating simulation code or performing model transformations [3]. Events are generated by the corresponding entity to trigger the simulation of component- or usage-behaviour [54]. They contain a call to an interpreter as well as the intended simulation time. Due to the event-based principle, no simulation time passes while an event is executed. After traversal of an action, the successor in the chain is traversed conforming to the event-list described in 2.2.2. Extensibility of *EventSim* was a specific goal in its development. The concept of traversal strategies reflects this extensibility. Here, each strategy describes a special traversal behaviour [4]. The definition of a new traversal strategy or a new action extends *EventSim* with additional simulation capabilities. Heinrich introduces an extension of *Palladio* in [4]. This extension enables the simulation of dependencies between software systems and the new domain of business processes. This extension is further described in the following section.

3.2.4 Business IT impact simulation

A major reason for problems or failures of Information System (IS)s in the industry is the missing alignment between them and the Business Process (BP)s in which they are used [4]. In the development of ISs and *BP*s, often their mutual impact is not considered. Thus, ISs and BPs are designed mainly in parallel rather than dependent on each other [4]. The missing dependence can lead to performance drawbacks at runtime which may not be obvious in the development process. For example, a BP can induce high workloads, which can result in large process execution times or overloaded resources [14]. Quality attributes of an IS can be acceptable in a normal use case, but can rapidly decline due to certain actions specific to a BP.

A IntBIIS is used to find dependencies and shortcomings at early stages of the development [4]. It intertwines BP and IS simulation which enables the prediction measures relevant to discover the performance-impact between a *BP*s and an ISs. For its realisation, the *Palladio* tool-chain and the simulator *EventSim* are used. However, the general model is not specific to *Palladio* and, therefore, can be applied to different BP and IS approaches when similar meta-models and prediction of performance measures are provided [4]. Each BP is modelled by a set of activities which are collections of linked steps and (sub-)activities in itself. Therefore, Heinrich [4] introduces the term "step" as the "smallest unit of work" in a BP [4]. These steps are distinguished between the entity performing them. They are called *actor steps* when completely performed by a human. Performed by an IS, they are called *system steps.* For each step performed by an actor, it can be defined if it is interruptible. Interruption of an actor step can occur if the actor step can be assigned to an actor with higher priority. Also, an actor step can be interrupted if an actor is outside of its working times [4]. A BP is located in an *Organisational Environment (OE)*, which can be seen as parallel to the resource environment of *Palladio* described in 3.2.2. It contains *human actors* as active resources, actively performing actor steps. *Device resources* can be optionally required to perform these steps. The device resources are similar to passive resources in *Palladio* and thus cannot perform steps themselves. Each human actor contains work periods and belongs to one or more organisational roles. The latter is used to group actors "exhibiting a specific set of attributes, qualifications or skills" [4]. Heinrich [4] clarifies that BPs are affected by different factors. Either by overloading/exhaustion of resources through too many actor requests or by the response time of an IS in a single system step. In the first case, the requested resources cause a bottleneck which can result in performance issues. If at least one resource is overloaded by too many requests and cannot take more requests, the BP itself is slowed or even interrupted. In the second case, large response times in the IS for one step can increase the response time of the surrounding actor step or even the whole BP.

To use *Palladio* with IntBIIS, the simulator EventSim was extended by introducing new simulation layers or extending existing ones [14]. These layers are shown in Fig. 3.2. For

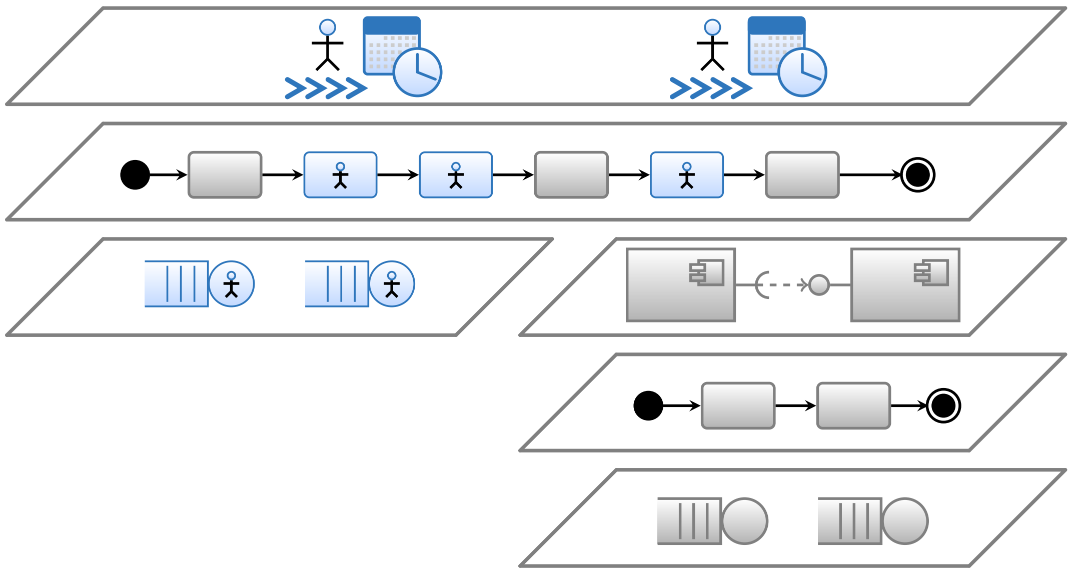


Figure 3.2: Simulation layers and elements in IntBIIS. Grey depicts the existing EventSim elements, blue denotes the elements and layers introduced by IntBIIS [4]

this extension, the mechanism of EventSim is used by supplementing a new traversal strategy in the case an actor step being encountered. The corresponding strategy selects a human actor resource from a set of human actors which meets the following criteria [14]:

- The human actor owns the organisational role
- The human actor has the shortest duration until the actor step can be performed

Non-suspended actors and those with the smallest duration to start the activity are preferred due to these criteria. IntBIIS introduces several new actions to *EventSim*. The *AcquireDeviceResource* and *ReleaseDeviceResource* actions realise an assignment of device resources to actor resources. This separation allows the application of the traversal strategy concept of *EventSim* [4]. This concept is used as explained in 2.2.2 to allow time to proceed even when the resource is assigned. In the traversal strategy of *AcquireDeviceResource*, the availability of the resource is examined. The resource is allocated for processing the corresponding action sequence if its requested amount is available. If the amount exceeds the available resources, the BP instance is blocked. The traversal of *ReleaseDeviceResource* releases acquired resources. To control the processing of actor steps and the suspension of actors the events *ProcessingFinishedEvent* and *SuspendEvent* are supplied. The first event indicates the finishing of an actor step and starts the next scheduling. The second event signals the intention of an actor resource to suspend (e.g. because of a lunch break) [4]. The utilised sensor framework in *EventSim* is extended by sensors measuring BP-related execution times and actor resource utilisation. To generate *BP* instances, the *Process-WorkloadGenerator* is used, which resembles an open workload scheme which generates instances with a certain inter-arrival time. The simulation of these instances are performed by the traversal of its action chain as described in [54]. While the corresponding action chain is traversed, either actor steps or system steps can be encountered. In the case of the former, the traversal strategy described above is applied. The traversal strategy for software-system simulation of *Palladio* is used [4]. Performance measures for theBPs as well as the ISs are collected through the corresponding sensors.

4 Related Work

This chapter provides an overview and discussion of works and approaches related to the topics in this thesis. This overview is partitioned in three superordinate sections. Work about interoperability and composability is discussed, which includes general theoretical work about interoperability in the context of HLA in Sec. 4.1.1. Concrete modelling approaches for HLA are described in 4.1.2. The use of ontologies to describe interoperability and composability are described in Sec. 4.1.3. Also the concrete approaches described in Sec. 2.3 are discussed in 4.1.4. Approaches of decoupling of monolithic simulations is examined in Sec. 4.2. Also, an insight in the availability of ADL for modular simulations is provided in Sec. 4.3.

4.1 Composability and Interoperability

One goal of the DSL used in the our modularisation approach is to provide the capabilities to describe the composition (i.e. the coupling) of modular simulations. One property of the DSL is that it has to be possible to describe the coupling approach of HLA. Because of this goal, motivating theoretical work concerned about interoperability in the context of HLA and model-driven development is presented in the following section.

4.1.1 Challenges in the Interaction of High-Level Architecture Implementations

The major and standardised interoperability approach for discrete simulations is the HLA. A deeper insight in the constructs and definitions of HLA is given in Sec. 2.2.5. The DSL aims to describe multiple approaches to connect simulations, but also the capability to describe the HLA. However, the HLA only describes the capabilities but provides no reference implementation. Because of this missing implementation, multiple independent realisations are produced. For each realisation, different programming languages with different Application Programming Interface (API) as well as unique communication protocols are used. Therefore, each federate must be implemented in the implementation-specific language and use a specific API of a certain realisation.

The implementation in a specific language poses the problem of the missing capability of each HLA realisation to interact with one another without further effort. Granowetter [57] calls this problem the "Interoperability Barrier" [57]. Along with the explanation of this problem, Granowetter [57] proposes approaches to reduce the impact of this barrier. These approaches are barrier elimination, work around or lowering the barrier. In barrier elimination, "wire standards" [57] like the DIS [28] are used.

Every implementation should decode and encode its network data according to a standardised definition in this approach. Therefore, all federates only needs to understand the standardised messages. This standardisation eliminates the need for every federate to be written in a specific language and to translate different encodings. At the same time, Granowetter [57] argues that this approach is a bad idea through too many restrictions to the HLA implementations. Also, this approach would dictate a lot of HLA functionality like, for example, time management.

In the idea of bridging two RTI, a RTI-to-RTI-bridge joins two federation executions as federate. Through this approach, data can be forwarded from one RTI to another. However, some HLA information like interactions or reaction to states cannot be forwarded in this way. Granowetter [57] therefore proposes the lowering of the barrier "through standardisation and cooperation of the RTI vendors, federate developers and federation program managers" [57].

Tolk [58] proposes the use of model-driven engineering approaches to end the "interoperability war" [58]. Through a model-oriented approach, the developers can be "supported in the design, implementation and execution phases" [58]. Tolk emphasises the positive aspect of metamodels through the effective "mapping and migration management" [58] of different solutions. The positive aspects are possible through the capability to transform modelled approaches into another, different realisations of already defined models. As one example Tolk [58] mentions the approach of the MOF. The proposed model-driven approach by Tolk [58] is another way to lower the barrier proposed by Granowetter [57]. Thus, one part of this thesis is to describe the data focused interaction (i.e., the interoperability) between multiple simulations through the use of model-driven technologies.

Parr and Keith-Magee [59] propose to apply Model-Driven Architecture (MDA) to HLA as well. The use of MDA is attractive due to benefits like improved reuse quality and interoperability when a standardised component model is used. At the same time, MDA decreases development effort, maintenance and costs. The need for a component model for HLA is stressed to provide reusable information and descriptions. As an example, Parr and Keith-Magee [59] use the Common Object Request Broker Architecture as a component-based approach in the domain of software engineering. One focus of the this approach is the notion of interfaces. The federates (i.e. the component) should provide or require an interface and therefore publish or subscribe the content of the interface [59].

Also, Parr and Keith-Magee [59] describe the use of the Unified Modeling Language (UML) for modelling and stress the need for a specific UML profile. UML should be utilised to fully describe the HLA capabilities with, for example, object and interaction classes and publication/subscription of object classes. Furthermore, HLA management specifics like time management or object management should be of concern to be described with UML. The idea of modelling simulations as components and the use of interfaces is supported in the DSL. However, only the data-centric capabilities are of prior concern and thus only the description of object- and interaction classes are provided. Also, the DSL provides the capability to model the services and functions of the RTI.

4.1.2 Modelling Approaches for The High-Level Architecture

Topçu et al. [21] show an approach of modelling the behaviour and data of a HLA federations. In this approach, the SOM, the FOM as well as the behaviour are modelled with the help of editors. A generator produces preliminary code using the models. This code can then be edited to produce executable code. This approach by Topçu et al. [21] is a powerful approach to model HLA federations and their behaviour. The approach provides capabilities for analysis and simulation. However, the limitation to HLA poses problems with simulations demanding other requirements and thus are not applicable to the functions and specifications of HLA. One example is the need for distributed simulation with agent-based models of Scerri et al. [60]. In the approach of Scerri et al. [60] the requirement of shared variables is stated. This requirement is approached with an additional service called "conflict resolver". Such a service is not planned in the HLA specification and thus, not realised in a model-driven approach with the sole focus on HLA. Because of such alternate solutions for distributed simulation, the DSL does not only focus on HLA. The DSL tries to provide the flexibility to describe different capabilities within a HLA-like structure.

Bocciarelli et al. [61] provides a model-driven framework to produce distributed simulations of autonomous systems. For this purpose the modelling language *SysML* [62] is used. The additional profile *SysML4HLA* enhances *SysML* with the capabilities to describe HLA like simulation structures. In the process of the approach of Bocciarelli et al. [61], autonomous systems are designed with *SysML* capabilities. The resulting model is then annotated with stereotypes of the *SysML4HLA* profile. Through a model-to-model transformation, the automated system is transformed into an UML model with HLA structure. This UML model is then transformed into Java code conforming to the *pitch RTI* implementation [63]. However, mostly code-stubs are generated and need to be filled with the final code. This approach shows a use-case of model-driven development to HLA in the application of *SysML*. However, the proposed functionality is only usable for HLA code and autonomous simulation. Also, *SysML* in this context does not provide the focus on the assembly of simulation as desired in the DSL. Thus, this approach does not meet the requirement to provide a general-purpose modelling approach to model the coupling of a modular simulation.

Neema [64] provides an integrated approach to model large-scale distributed simulations. This approach enables the modelling of simulations and integrates many simulators like *OMNeT++* and *MATLAB/Simulink*. The HLA is used to coordinate time advancement and data routing for common interaction between the described simulations. Because of the application of HLA, Neema provides a metamodel for the modelling of federations and federates. This metamodel includes the ability to model interaction classes and object classes as given by the SOM. For each federate, the publish and subscribe relationship can be defined. Neema [64] also provides a metamodel for deployment and execution information. This description is utilised by a model interpreter to move all generated scripts and files to the execution destination. One difficulty encountered by Neema [64] was the use of simulations with their self-described data model. The difference in the data model of each simulation causes the problem of incompatible described data. Thus, a simulation can receive data from another simulation, but cannot interpret it. In HLA this happens in the approach of Neema [64] if two simulation engines use different FOMs or SOMs. Neema [64] gives examples for such problems. One example are different names for the same data like "Hello" and "Bonjour". Another example is an object class modelled by two simulations. Here the attribute "ID" is represented in one simulation as Integer and in the other simulation as String. Neema [64] provides a mapper as a new federation to remedy this problem. This mapper is responsible for transforming data. Neema [64] provides a metamodel to describe mappings between information. If more complex transformations are required as described in the examples above, the user can insert Java-like code. Due to the similarity of the approach of Neema [64] to the one we create, the differences need to be discussed. Neema [64] provides an integrated approach using HLA. The HLA capabilities are implicitly used but not explicitly specified. Also, the data is modelled with the attributes of HLA. The sole focus on HLA poses the problem of use in other simulation domains. This problem was shown in the example of Scerri et al. [60] in Sec. 4.1.2. Our DSL is designed with the use to model not only HLA capabilities, but other possible approaches. Therefore also functionality used for the interoperability can be defined. The DSL provides also focus on composability and reusability. The reusability is facilitated by the use of independent interfaces. Furthermore, the metamodel is designed to a independent description of simulations and coordinators. These capabilities are complemented with the ability to describe the assembly/composition of simulations. The DSL provides a guideline for roles to specify the simulations and assembly. One common aspect is the bridging of different data models with mappers. The mapping integration in the approach Neema [64] is realised ready to be used. Nevertheless, the DSL provides reuse of the adaptations in different models. Additionally, approaches to structure the mapping of data are provided to reduce the number of descriptions necessary. Furthermore, the metamodel is designed to enhance the capabilities of adaptation further.

4.1.3 Using Ontologies for Simulation Composition and Interoperability

An approach in the field of composability is provided by Benjamin et al. [65] through the use of ontologies. An ontology defines entities in a domain, together with their properties and relationships. The approach of Benjamin et al. [65] enables ontologies to describe components and their properties and place them into a repository. When components are required, they can be chosen through their requirements. Benjamin and Akella [66] extract ontology models for each simulation application. The sources for these models are text sources like requirement documents, design documents and source code. The metamodel of each application is described through the ontologies to facilitate interoperability. An ontology-driven translation is carried out by mappings, which define "equivalence between concepts" [66] in the simulation application.

Gutierrez and Leone [67] use an ontology network for building distributed simulations for supply chain management. The aspects of composition and interoperation are realised through the application of the BOM and HLA. The ontology network consists of four ontologies: *BOMOnto*, *FEDOnto*, *SCOnto* and *EMOnto*. The *FEDOnto* provides the concepts of HLA to generate the conceptual model of a federation. Semantic information enriches this model through the *BOMOnto*. This ontology describes the information contained in BOMs. A federation realises the supply chain concepts, which are described by the

SCOnto ontology. The implementation of the business process within the supply chain is modelled with the *EMOnto* ontology. To use the approach of Gutierrez and Leone [67], each federation member must agree upon a FOM definition. Followed by this agreement, already created SOMs can be uploaded to build the resulting ontology. This approach poses a possible drawback because the participants of the simulation need a previously created SOM. Through the need for an existing SOM, the approach of Guiterrez and Leone [67] is only applicable after the design process. However, building a model of the federate is desirable along the design process of a simulation. This enables the developer to select fitting simulations to achieve a composed capability. Özdikis et al.[68] provide an application of an ontology with respect to HLA. In this approach, an ontology is used for a transformation to a HLA object model. For this purpose, a tool is provided to transform the proposed ontology to OMT constructs. The ontology is designed to represent multiple capabilities not only those specific to HLA. However, in the approach by Özdikis et al. [68], the HLA specifics are directly provided in the process of ontology-to-HLA transformation. This ontology can be seen as a part of the whole approach. However, no composability features are provided.

Ontologies can express many aspects needed to describe assembled simulations. However, the underlying ontology must be carefully designed. Also, it is possible that it can express properties not intended to. The creation of a metamodel provides a more detailed restriction of such properties. Because of this, a model-driven approach using the Eclipse Modelling Framework (EMF) is used to provide the DSL

4.1.4 Simulation Composition Approaches

In 2.3, approaches to define or model the composition of simulation models are presented. In this section, the reasons for not selecting one of these approaches is discussed.

CODES provides many features to describe distributed simulation and ensures simulation composability and interoperability. The use of black-box components provides the possibility to exchange a component by a different one with the same capabilities. The behaviour description allows to capture simulations in the aspect of their complete behaviour. Also, the check for syntactic composability ensures the valid coupling of the simulation. However, to provide all of these capabilities, the whole simulation development must adhere to the structure provided by *CODES*. Also, other simulation structures like the use of HLA are not possible. In contrast to this restriction, the DSL is designed on a higher design abstraction level. This allows the provision of a tool for describing simulation without specifying concrete underlying structures and mechanism.

The *DEVS* approach enables developers to describe a large amount of properties of composed simulations, such as a hierarchical structure, input couplings, output couplings as well as encapsulation of atomic models. A formalised approach also enables the application of formal methods to check for consistency. However, it can be assumed that domain experts are not familiar with the formal description of DEVS. Thus, they have to learn and understand this formalism before they can design their system. Also, DEVS does not provide reuse in the sense of interfaces definitions. In the DSL, the notion of hierarchical composition and the coupling is taken but provided more abstractly. The use of a DSL can provide applications for different developer roles to enable a better understanding and specialisation to certain aspects of simulation development. DEVS is also used in the approach by Röhl and Uhrmacher [40]. The DEVS formalism and the XML format is used to provide the reusability of components. This approach includes the reuse of interfaces to describe different simulation capabilities. Also, hierarchical compositions can be defined. However, the model must be described in the DEVS formalism. Also, the direct use of the simulation tool *James II* prohibits the use of other simulators. Furthermore, the description of object-oriented simulation models and descriptions of composition approaches is not possible. In the DSL, the notion of components and interfaces is used. A hierarchical composition is also provided to achieve similar results. The model-driven approach, however, should help developers to compose simulations without knowing a specific XML syntax.

As described in Sec. 2.3.3, *Ptolemy II* allows the analysis of its described composed simulation. An analysis is especially useful when the simulation is in the design process to gather information about it. The hierarchical description of simulations is provided to enable a finer granularity of the design. The use of the black-box principle enables the exchange of actors with different MoCs but the same functionality. However, due to the approach of *Ptolemy II*, each simulation has to be described with its tool-set. Thus, existing simulations must be designed with the tools of *Ptolemy II* again. Also, the possibility to describe reusable interface is not given either. These interfaces could be reused in other simulations to describe the data needed for the same simulation aspect. This notion of defined reusable interfaces is however given in the DSL. Also, the DSL provides the capability to describe multiple interoperability approaches like HLA, which is not given in *Ptolemy II*

OMNeT++ also provides the desired capability to describe atomic and composed components. The gates provide particular points of interaction between components. Also, the black-box view on simulations provides the capability to exchange simulations by others. However, the *OMNeT++* approach restricts the user on its specific capabilities as well as to its prescribed structure. In the DSL, the encapsulation of simulations in larger simulations and the passing of information is also provided. However, *OMNeT++* does not provide a model-based approach to describe additional information like management functionality or functions to be realised by the simulations. The restriction of *OMNeT++* to specific languages provides another drawback to describe composed simulations. Thus, the DSLs model-driven approach uses a higher level of abstraction. There, models with their provided and required data can be defined without keeping the final realisation in mind.

4.2 Decoupling in the Context of Monolithic Simulations

All prior works in this chapter are concerned about the creation of simulations according to composability approaches. However, simulation models can still be contained in monolithic simulations. The underlying simulation models contained in a monolithic simulation need to be extracted to be reused individually. Regrettably, the works on the extraction of simulation models confined in monolithic simulations are found to be scarce. Papadopoulos et al. [69] use dynamic decoupling to partition a complex equation-based object-oriented model into sub-models. This approach is used on physical systems which run on continuous or discrete time-scales. Dynamic decoupling is done in a two-phased approach. In the analysis phase, the structure and the time-scales in the system model are analysed. The decoupled integration phase then tries to improve simulation efficiency. The proposed approach is specific to equation-based systems and therefore can only be used in this domain. The DSL tries to provide a more general approach to find a way to decouple simulations. Fish and Chen [70] provide another decoupling approach of monolithic simulation in the field of multiphysics. However, this approach is specific to equation-based piezoelectricity physics. Thus, this work is not usable to the goal to propose a more general decoupling approach in the form of guidelines.

Other related sources are inspected due to the lack of literature on the topic of decoupling in the context of simulations. Some sources exist in the domain of software engineering. One goal in the decoupling of software is to identify separate components. To achieve this goal, Kim and Chang [71] presents a step-wise UML-based approach to identify components. The identification is achieved through the application of clustering algorithms, metrics, heuristics and decision rules. Metrics are used in the first step to analyse functional dependencies between use cases. In the second step, the determined metrics are applied to a clustering algorithm. This step allows to find and combine use-cases with the same values. Step three allocates classes to components. The allocation is executed through the inspection of sequence diagrams which belong to the clustered use-cases. If a class is assigned to more than one component, metrics are calculated between classes and components. Following, the conflicting components are compared using these metrics. In the fourth step, the optimal selection of components is conducted. A value to determine the number of components and the granularity is proposed. The components are chosen so that this value is optimal. However, such an approach can allocate functionality valid in the sense of a software component. However, simulations consist of semantic systems. If only a functionality based approach is applied to the code of a simulation, it is possible that semantically connected code pieces are separated. Thus, in the separation of simulations, not only functionality is of importance but also the semantic of the functionality.

Choi and Cho [72] provide an approach to identify components by employing use-case diagrams and sequence diagrams. Choi and Cho [72] provide definitions for static and dynamic dependencies between classes. For static dependencies, relationships between classes are exploited in a class diagram. Those relationships are composition, inheritance and association [72]. Composition describes the containment of a class in another, so that writing to one class directly influences the other. For example, the creation of one class includes the creation of the other. According to Cho and Choi [72], "classes in a composition tend to be operated as a functional unit" [72]. The inheritance relationship provides high cohesion because the child classes inherit properties of the parent class. If those classes were distributed in multiple components, the cohesion between those components would be equally high. Cho and Choi [72] state, that composition and inheritance relationship provide a "strong bond" [72] between classes. Therefore these classes have to be part of the same component and cannot be used independently. In the association relationship, one class can send messages or invoke functions of another. This relationship does not provide direct influence on the other class. Nevertheless, a high degree of dependency exists. Dynamic dependencies are analysed in the form of call types between classes.

For this purpose, Cho and Choi [72] define five classes of calls between classes and their direction. The direction is either uni-directional (i.e. one class calls methods of another) or bi-directional (both classes call methods of each other). With these characteristics, Cho and Choi [72] propose multiple criteria to identify business/system components. Following, a step by step approach is proposed to identify these components. Requirements and use-cases are utilized to identify candidate components. Call graphs are used to group the found classes. The approach by Cho and Choi [72] uses a descriptive way to identify business and system components. The notions of relations between classes are beneficial and the provision of a step by step approach provides a solid basis to be used. However, this approach is specified on software systems. Simulations also have to incorporate semantics to be valid which are not part of the approach of Cho and Choi [72].

Dehghani [73] gives descriptive guidelines on how to decouple monolithic software into micro-services. An example is to minimise the dependencies of the decoupled micro-services to the monolith. This guideline can be used in the context of simulation to decide if a simulation model should be further divided. Also, the idea to decouple the software system step by step and not in a "big bang" approach is applicable to simulations. The step by step approach enables the simulation in use to remain operational with the decoupled parts and the monolithic system along the decoupling process. Lots of the other guidelines given by Dehghani [73] are however more specific to software systems. Because of this, they are only usable as ideas in the simulation decoupling.

Sarkar et al. [74] provide a case study and a modularisation approach to a large-scale business application. In this approach, monolithic systems partitioned into domain modules for which interfaces are found. This partitioning is done by identification of domains and their domain-specific business operation represented in the software system. Through heuristics, the domain's files are identified and assigned to the domain modules. With these files, sub-modules are created to express specific functionality. The intermodule interaction is inspected by using static code-analysis tools. Especially calls between modules and receiving functions are of interest. For the functions between domain modules, interfaces are defined. The described approach is again specific to software systems. However, the separation of domains can be used on larger simulation models. Then interfaces between the contained simulation models can be found. The interaction between modules can be seen as interaction between simulation models. So, even if this approach cannot be used directly, some ideas can be used in the creation of the decoupling approach.

Software decoupling is a problematic topic in the field of software engineering. Taibi et al. [75] show this predicament in an empirical study about the migration from monolithic architectures to micro-service structures. In this study, the results from interviews of 21 practitioners concerning this migration are discussed. One of the main issues is the "complexity to decouple from the monolithic system" [75]. An approach to this topic is to reimplementing the capabilities of the system as micro-services "from scratch" [75]. Another approach is to only implement new software features as micro-services. In simulation development, this is analogue to only implement new simulation models with a coupling approach. All work concerning the decoupling of software systems is only used as guidelines. However, decoupling approaches in software engineering often miss semantic aspects of simulation models. For example if in software engineering, two functionalities can be separated. However, in simulation, these functionalities may belong tightly together to generate a semantic relationship. Also, no concrete step by step support of decoupling has been found.

Few relevant solutions are found, which leads to the conclusion that works on the decoupling of simulation are hard to find or scarcely researched.

4.3 Architecture Description Languages for Modular Simulations

In software engineering, ADLs are used to describe the structure of software systems. However, they allow not only the description of software systems but also their analysis. In the domain of distributed simulation, Coen-Porisini and Baresi [76] introduce the Simulation Architecture Description Language (SADL). With SADL, each artefact during the different design phases of a distributed simulation can be described. UML is enhanced in SADL to be able to describe assembled simulations specifically. This approach also incorporates the creation of a metamodel. It can provide "a formal description of all the entities that are relevant in the design process" [76]. In the first design phase, the information model is described. Each information model consists of reusable elements with attributes. For each element input and output, gates are defined. Each piece of information can only flow through points defined by these gates. In the second phase, the developer creates the system architecture. One or more objects are created in the process of system architecture specification. Each object is an instance of an element and consists of assigned values. Those values correspond to instantiations of the referenced elements attributes. In the simulation architecture definition phase, each object is assigned at least one simulation component. The simulation component has input interfaces and output interfaces. Links connect each interface type. Filter components are provided in addition to the simulation components. Each filter component transforms data to enable simulation models to exchange information if their data representation does not match. Another form of a component is the activator, which can alter the control flow of a simulation. Each simulation architecture also has to specify a data flow description (i.e. the information that is exchanged among components). Also, the control flow description has to be supplied, defining how the components interact. The simulation deployment can be described with the resulting simulation architecture model. In this design step, the components of a simulation architecture are distributed on "nodes". Each node represents a process or processor. SADL also provides consistency checks, to signal errors in the modelling process to the user. SADL also provides an approach to model and analyse simulation compositions. However, SADL is rather restrictive in the way simulations interact. The metamodel of SADL does not enable a developer to describe management functionality used in distributed simulations as, for example, RTI-management-capabilities. However, this capability facilitates a broad and defined use and enables to describe the capabilities for the management of multiple simulations in a specific approach. The notion of the filter component is required for a integration of independently created simulations. A similar approach is used in the DSL but with a different design to enable a broad reuse of the filters and mappings.

Besides SADL there are not many other full-fledged ADLs specialised in simulations. This problem is also recognised by McKenzie et al. [52]. Instead, the software ADLs *Rapide* [77] and *Acme* [78] are used to describe and analyse distributed simulations. The architectural descriptions are based on HLA and two federation architectures. Through successful application of the ADLs, McKenzie et al. [52] concluded that general purpose ADLs are applicable to distributed simulation. However, McKenzie et al.[52] also stress the benefits of a standardised simulation ADL specific for the simulation community.

5 Extraction of Simulation Models from Monolithic Simulations

This chapter provides information to support the extraction of simulation models out of monolithic simulations. For this purpose, the monolithic simulations IntBIIS and *HumanSim* is inspected. Therefore, the provided information are only related these two monolithic simulations but it is possible that they are applicable in other simulations with similar structures as well. The scope of this description chapter is restricted to DES and process oriented simulations as described in Sec. 2.2.1. With the identified information entry-points to inspect the connection between the simulation models are provided. The found connections can then be used to extract the simulation model and describe required and provided informations of each extracted model.

First, Sec. 5.1 describes structure of a monolithic simulation as created of its features. Sec. 5.2 then provides conceptual elements in these modules to be important for the extraction of simulations out of a monolithic simulation. Sec. 5.3 describes how these elements can be used to define aspects to concern when extracting simulation. Finally, some problems confined in different aspects in the extraction of simulations such as the synchronisation of simulations or tooling approaches, are discussed in Sec. 5.4

5.1 Simulation Features of Monolithic Simulations

Monolithic simulations contain one or several simulation models to represent the system to be analysed (see Sec. 2.2.3). These simulation models are designed to correspond to the specific structure of a simulation. In the process of modularisation, the simulation itself can be divided into its features. These features can then be extracted to be reused. These extracted features are called "simulation features" because each feature constitutes a simulation which represents a feature of the former simulation. An extracted feature itself does not have to contain only one simulation model. For example the Palladio feature can contain simulation models for reliability and performance. If Palladio is extracted out of IntBIIS, these models can be extracted together as one feature. The business process model provides another feature. The separation of a monolithic simulation in its features has to be done by application of contextual information. This information is especially important, because simulations are claimed to be developed on context-sensitive assumptions [18]. Either a developer knowing these assumptions can separate the simulation models or the developer has to obtain knowledge about the simulation first. This gathering of knowledge, however, can be cost and time intensive.

5.2 Simulation Information to be Identified for Extraction

If a feature is identified by context information, its confined elements can be found. Three categories are found to define information in simulations. First, entities as shortly described in Sec. 2.2 used in the simulation system can be identified. These entities are used to to potentially progress the system state by interactions with other entities. Other kinds of information is the flow of the simulation execution (hereafter called "execution flow") and the flow of data between features. Entities and the flows can be used in the extraction of models and their code out of monolithic simulations. Therefore, they are defined in the following subsections.

5.2.1 Simulation Entities

Entities flow through the system and interact together to possible change the state of the system [11]. These entities and their use in each simulation feature have to be identified for the extraction. The categorisation of Banks et al. [11] is used to categorize these entities in two classes. The first class are *dynamic entities*. These entities are used to actively interact with other entities and can flow through the system. This interaction can change the state of the system and generates the systems behaviour. In IntBIIS, *Actor* instances are dynamic entities. They use device resources to change the state of the simulation (e.g. by obtaining device resource capacity to proceed with their execution or to block other actors). The second class of entities are static entities. These entities are used by dynamic entities to support the execution of the simulation behaviour. Static entities themselves do not interact with other entities themselves. In IntBIIS, static entities are device resources. These entities are used by actor resources to generate the desired behaviour of the simulation model. The specification of dynamic and static simulation entities is dependent on the simulation model they are used in. Therefore, the static entity can become an dynamic entity in an evolutionary scenario. For example, device resources in IntBIIS are currently only used by actors. Their occupation regulates the simulation flow (e.g. by blocking the process of actor resources when no capacity is available). The device resources themselves do not provide any further behaviour. If, for instance, the model is extended and device resources can interact with each other (e.g. to provide networked behaviour), the device resources become dynamic entities.

5.2.2 Flows in Monolithic Simulations

The information flow and execution flow in monolithic simulation are identified to be significant for extraction of simulation features. The execution flow of a simulation is expressed by the traversal of events in DES or the execution of the process steps in process oriented simulations. Multiple execution flows can exist in a monolithic simulation. The dynamic entities are used in this flow to interact with other entities and the system itself to change its state. Entities are related to execution flows. For example, each user of a user-scenario in Palladio has its own execution flow. The state of the simulation features (i.e. state variables) can alter the execution flow of a module. Also, the execution flow can be dependent on entity attributes. For example the state variable of a component in Palladio can describe which event is called as next. The execution flow and the entities in this flow also induce the second type of flow, the data flow, in a simulation. The information- (i.e. data-) flow defines the transfer of information between simulation entities. An information flow exists when dynamic entities change attributes of other entities or call one of their functionalities. The information flow can be direct between an dynamic entity and dynamic or static entity. Also an indirect flow exists where information is transferred between dynamic entities over an static entity. The information flow is of importance when extracting models to specify the data used by the extracted simulation. It is possible that these information can be optional. The data flow between entities can alter execution flows when they are dependent on state variables.

5.3 Usage of the Flows and Entities

The application of the proposed information confined in simulation modules described in Sec. 5.2 is provided in this section. The execution flow can be used to identify the entities confined in a simulation feature. The interactions of entities define the interfaces for interaction between simulation features. Furthermore, direct interaction between the execution flows can be identified. Therefore, connection properties in the case of an extraction can be found when inspecting the entities and the execution flows. For this purpose, the identification aspect of entities in simulation features is discussed in this section. Furthermore insight is provided for inspecting requirements and provisions of simulation features. In addition to this inspection, the need for replication of entities in the modules is discussed. Additionally, different types of waiting schemes in modular simulations is explained.

5.3.1 Identification of Entities by Simulation Features

In a modular simulation, every entity has to be identifiable to be usable by other simulation features. This identification allows a entity of one simulation feature to interact with an entity of another simulation feature. If no identification is possible, the entities cannot interact. The interaction between actor resources and software systems can be taken as example. In the current monolithic system of IntBIIS, the traversal of the usage scenario is executed by traversal of the corresponding modelled actions by references. In a modular simulation, IntBIIS can be split in a Palladio simulation feature and a business process simulation feature. A specific software system cannot be specified to be accessed by an actor step through a reference due to this separation. Because multiple software systems could be realised, the business process simulation feature has to define which system has to be traversed. If no identification would be provided the usage of a certain software system is not possible

5.3.2 Determination of Requiring and Providing Data

To determine the requirements or provision of data, the information flows between execution flows of the simulation features have to be found. In a direct information flow between two active entities, the simulation feature containing the entity changing the variable can be declared as "providing". The simulation feature containing the entity with the changed value can be declared as "requiring". This declaration has to be done for every transferred data. It is possible that the data-flow between two entities is not direct. Static entities can be used to exchange information. Here, the dynamic entities accessing the static entity have to be found. However, the required and provided properties themselves are the same as in the simple case. Furthermore, the events or process steps themselves can change or call functionality or change data of entities in other simulation features. These changes of data and calls of functionality have also to be declared as "provided". It is possible that a entity is conceptually assigned to two or more simulation features. These simulation features can utilise and define the attributes of one entity. In this case, the required and provided information cannot be found by analysis of the data-flows themselves. This property has to be found by inspecting all execution flows for usage of a dynamic entity. However, such a inspection can be time consuming and context information is helpful. If this constellation is found, the attributes utilised in one simulation feature and modified in the other simulation feature have to be required. This specification can result in the constellation, that simulation features both require and provide values for attributes of one entity.

5.3.3 Replication of Entities

The flow of data between two modules realises a change of state. In an entity driven simulation, the information about entities used to interact with has to be stored in every simulation feature. The necessity for representation of entities from other features can be identified by the dynamic entities in the execution flow of a simulation feature. Every entity in other simulation features accessed by an dynamic entity in the analysed simulation feature has to be represented. The entities can be seen as the interfaces between the simulation features. To replicate an entity it suffices to represent only the properties to be accessed or changed (e.g. attributes or functions to access). This allows to only provide a limited representation of the entities of other simulations and reduces overhead in the implementation. It can even suffice to only store the identifying elements on an entity (e.g. a name or identifier) The need for replication is not bi-directional. Therefore, if one entity has to be replicated in one module, the other module does not necessarily has to also contain the entity. If entities are conceptually assigned to two or more simulation feature as described in Sec. 5.3.2, the entities have to be represented both of them. Here, all attributes used in the execution flow of the simulation features have to be represented.

5.3.4 Types of Execution Flows in a Waiting Scenario

The execution flow is especially important for the representation of waiting for an event. With a execution flow, conditional and unconditional waiting can be expressed. In conditional waiting, the execution flow builds loops and checks for a change in state (i.e. change in the value of an entity). Here, another execution flow has to alter the state (e.g. alter state variables or attribute values of an entity) to let the waiting flow proceed. However, this flow also allows to escape or change the behaviour if the condition is not reached. Palladio and IntBIIS does not provide such an approach. Therefore, we use the example of the movement of a troop and a supply transport. The troop approaches a point and enters the waiting state. The execution flow of the troop reschedules the wait event until a variable is changed (e.g. a "train arrived" value on their position). If the transport arrives, the value is changed and the execution flow proceeds with the next event. The rescheduling can also be "broken" by providing a escape condition (e.g. a certain time has passed). Unconditional waiting on the other hand is represented by the end of an execution flow. In this approach the execution flow (e.g. scheduling of events) is interrupted. The only way to proceed with the conceptual flow is by the resuming through another execution flow. As an example in IntBIIS, we conceptually separate the actor step traversal and the traversal of software systems. When a actor step is executed, a point is reached where the software system traversal is scheduled. The traversal of the actor step is ended but the conceptual flow (e.g. the work day) is only paused. Then traversal of the software system is executed. At its end, the execution flow of the software system resumes the execution flow of the actor step. A more distinct example is the one stated above. However, this time, the troop execution flow is not rescheduled in the loop. The execution flow is only resumed by the arrived transport. With unconditional waiting, an explicit dependency on another flow can be produced. Without the interaction of this flow, the execution of the model is not proceeded. Thus this approach states a true requirement for the interaction with another simulation.

5.4 Challenges in the Extraction of Simulations

The extraction of simulations provides some challenges in different areas of development. Some problems are presented in this section.

5.4.1 Tooling

We use simulations written in Java. In the process of this thesis, only few tools could be found to provide valid information for decoupling. The static structure of a simulation can be viewed with *Structure101* [79] or *SonarGraph* [80]. The static structure can show isolated models or references between identified dynamic or static entities. To find the data and control flow, data flow analysis or program slicing can be used. However, in the domain of the Eclipse IDE, we found no tools working with our eclipse distributions (Eclipse Modelling Tools Neon.3 (V4.6.3)). The decoupling should not be hindered by a unsuccessful or time-intensive setup of tools. This would increase the cost of the decoupling and can make it infeasible from the perspective of stakeholders.

5.4.2 Duplicated Code

One problem in the decoupling of a system is duplicated code. This duplication is necessary for dynamic and static entities used by multiple simulations as described in Sec. 5.3.3. For each simulation, the entity has to be internally represented. Either a full fledged representation or only by representable stubs. Another problem arises with the simulation engine. No problems arise if the monolithic simulation already uses different simulation engines for the execution of each underlying simulation model. However, this is seldom the case as seen for example in IntBIIS where EventSim is used for the Palladio as for the business process simulation model. Therefore, if these two simulation models are placed in their own simulation, the code of EventSim has to be used in both simulations. Duplicated code is a problem for maintainability. The code of the simulation engine has to be maintained separately. This problem can be approached by using a plug-in system. This is the case for EventSim. A plug-in system enables the central maintenance of the simulation engine.

5.4.3 Synchronisation of Simulation Time with Coupling Approaches

To use extracted simulations, an coupling approach has to be used to enable the creation of the desired behaviour of a modular simulation. The goal of extracting a simulation feature out of a monolithic simulation is to reuse it as a single simulations and in modular simulations. Therefore it is important that extracted simulations can be reassembled in a modular simulation to provide the monoliths behaviour. Coupling approaches can contain the ability to synchronise the time for the simulation features interacting with it. Monolithic simulation however, have to provide their own time mechanism which is confined in the used simulation engine. Two approaches are identified in the extraction of a simulation features exists to use the time-line of the coupling approaches. The first approach is to use the monoliths simulation engine and its time-mechanism in the extracted feature as well. In this approach, the time-line of the simulation engines of the features and the time-line of the coupling approach have to be synchronised. This synchronisation has to be carefully executed so that no information is lost. This can pose great difficulty dependent on the simulation engine and the underlying use of the simulation features models. If, for example, in one simulation feature only one execution flow and entity exists, its synchronisation can be directly to the time-line of the coupling approach. However, if in one simulation feature multiple execution flows for entities exist, then one execution flow can schedule a time advance after that of another entity with a greater time advance. In such a case one of these time-advances can be lost. This can result in incorrect interaction with the other simulation features. Therefore, the time-mechanism of the simulation engine has to be adapted to synchronise the execution flows themselves and also to synchronise them in regard to the time-line of the coupling approach. The other approach is to utilize only the time-mechanism of the coupling approach. Therefore, only the simulation model is used but not the monolithic simulations engine. This however results in two major drawbacks. First, simulation engines provide capabilities for simulation approaches like DES (i.e. event-scheduling-mechanisms). These features have to be either be replicated or other simulation approaches have to be used (e.g. process oriented simulation instead of DES). Second, the simulation features is fitted to a certain coupling approach and cannot be reused with other approaches.

6 A Description Language for Simulation Coupling

In this chapter, a DSL is introduced to describe the coupling between simulation features. This DSL is designed to support the modularisation of monolithic simulations or the design of new modular simulations. The modular simulation can only use the information of the contained simulations. Simulations of a modular simulation can be other modular simulations themselves and simulation features.

The interaction between simulations consists of an exchange of information. The term information of simulations includes the data of simulations (e.g. attribute values) and notification of simulations about certain events they can react on of other simulations. The modular simulation has to use a coupling approach to enable interactions between its used simulations. The coupling approach is called Modular Simulation Environment (MSE) in the DSL. The simulations and the MSEs have to be combined to create a working modular simulation. This combination is realised in a modular simulation assembly. Therefore, the term modular simulation and modular simulation assembly is used interchangeably in this chapter. The DSL is designed to provide the capabilities for the description of the static architectural and information-driven design of modular simulations. The static architectural description is related to the used simulation and their connection through the MSE. The information driven design capabilities are used to describe the exchange of information between simulations. This design poses some limitation on the use of the DSL. One limitation is the inability to describe the aspect of a modular simulations' behaviour.

The DSL is created with model-driven technologies. Therefore the DSL provides a metamodel for the description of the modular simulation and its content (e.g. simulation modules). The EMF [81] is used to design the DSLs' metamodel. The capabilities of the DSL have to be extensible for possible modifications in the future. For this purpose, metamodel elements are provided to be used as entry-points for extension. The DSL is designed with the following goals:

- The DSL has to contain the capabilities to define the structural parts of modular simulations. These capabilities include the definition of simulation features and the MSEs. The applicant of the DSL must also be able to connect these definitions to modular simulations.

- It has to be possible to describe modular simulations as a combination of other modular simulations and simulation features.

- The information contained in a simulation must be describable. This description enables the use of the information in the assembly of a modular simulation.

- Exchanged information between simulations has to be describable. This description is done by the specification of required and provided information.
- The DSL has to provide the capabilities to model simulation features and MSEs, independently. This independent design includes that no knowledge of the other used simulations or MSEs of a certain modular simulation assembly is needed. This goal enables the use of models of third party developers because no assumptions about other simulation features or MSEs can be made.
- The information contained in simulations and MSEs must be represented to be usable by computers. Also, the DSL has to be able to model an object-oriented scheme for the representation of information contained in simulations and MSEs.
- It must be possible to describe MSEs to facilitate capabilities for interaction between simulations. This description includes possible callable functionalities and additional information needed to provide these capabilities.
- The metamodel of the DSL has to allow the description of at least the capabilities and aspects of the HLA. However, not only the HLA has to be describable. Defining more, less or different capabilities must be possible.
- Incompatibilities can arise due to the independent design of simulations and the MSEs. The main reason for these incompatibilities is the different definition of information in the independent description. Therefore, the DSL has to provide an approach to mitigate the incompatibilities of the described information.
- The models created with the DSL shall be reusable in different modular simulations. This reusability includes the descriptions of simulations and MSEs. Also, the content of the approach to mitigate incompatibilities of information have to be reusable. Additionally, the modular simulations themselves have to be reusable.

This chapter is structured by first giving a more precise definition of our understanding of modular simulations in Sec. 6.1. This definition also includes the constituting parts of modular simulations. The solution approach to mitigate incompatibilities of information is described in Sec. 6.2. An introduction of the DSLs' metamodel is given after the description of this approach. Sec. 6.3 provides an overview and discussion about the package structure of the metamodel. The detailed description of the elements contained in the metamodel is given after that. This description starts with the introduction of basic metamodel elements to enable an identification of model elements Sec. 6.4. Sec 6.5 describes the metamodel elements used to specify information in simulation features and MSEs. Sec. 6.5.1 includes the description of data types used in information to make them usable by computers. This description is followed by the explanation of the metamodel content to model notifications in Sec 6.5.2. These elements are required to define the information contained in simulations and MSE. Sec. 6.5.3 describes the representation of information in the simulation. The information in the DSL is represented by an object-oriented structure. Sec. 6.5.4 provides a discussion about the chosen design of the metamodels object-oriented structures. Sec. 6.6.1 describes the elements to model simulation features. The metamodel parts to describe

MSEs are provided in Sec 6.6.2. Sec. 6.2 describes the approach to resolve incompatibilities between information of independently designed simulations and coordinators. This description includes a proposition for a possible process to use this approach in Sec. 6.2.2. Likewise, the identified areas (or types) of the solution approach are explained in Sec. 6.2.1. The metamodel content for the created solution is described in Sec. 6.7. Sec. 6.8 contains the definition of metamodel elements to model the assembly of a modular simulation. Sec. 6.8.2 describes the provided elements to enable the use of the independent designed information in the modular simulation. These elements include the capability to specify additional context information for data and notifications contained in simulations. This specification enables processing of information by the MSE. The required and provided data in the modular simulation has to be specified for each simulation. This specification is realised by mappings to interfaces described in Sec. 6.8.3. The approach to resolve incompatibilities of simulation is defined to be reusable in different modular simulations. Therefore, the created models have to be kept abstract to to be applicable in multiple modular simulation models. The abstractly defined parts have to be connected to the information in a modular simulation to enable their usage. The metamodel elements realising this concrete specification are described in Sec. 6.8.4. The simulations and MSE parts have to be statically connected to describe possible information flows. The description of the metamodel elements used to connect the content of modular simulations are given in Sec. 6.8.5.

Also different developer roles are proposed for the use of the DSL. Different roles enable a specialisation of developers into different aspects in the creation of modular simulation. For example, one developer specialises in the development of MSEs and another in that of simulation modules. This specialisation can result in more reliable content due to the specialisation. Also, parallel and interleaved solution of the tasks in the development of modular simulations is possible. The roles are described in Sec. 6.9

6.1 Modular Simulations in the DSL

Modular simulations in the DSL consist of several simulations to provide desired capabilities and information. These capabilities are achieved by the interaction of the contained simulations. For this purpose, the contained simulations can provide data to other simulations or require data from others. The information required by a simulation is necessary for correct processing of its underlying model. For example, the response time of software systems is required in the business process simulation to calculate the overall execution of an actor in IntBIIS. Also, simulations can require notifications from other simulations to react to them. In DES these notifications typically trigger a particular event. For example, Palladio requires the notification to start the simulation (i.e. the traversal) of a software system. In some scenarios, notifications are not only required for correct execution of the simulation. It is possible that a simulation does not proceed with calculations if it does not get a specific notification. The required information has to be provided by another simulation. Palladio provides the response times of software systems to other simulations (e.g. to the business process simulation) in the above examples. IntBIIS provides the notification to Palladio to start the simulation of a software system. Notifications can include additional pieces of information to provide a better specification. For example, IntBIIS could provide an id of the software system to be simulated. Functionality has to be provided to enable interaction between simulations contained in a modular simulation. The functionality is defined in a MSE in the DSL. The MSE defines capabilities to realise correct interaction between the simulations. This definition includes capabilities to manage several common aspects in simulations. For example, the exchanged information of two simulations or the representation and management of simulation time. Multiple approaches for MSEs can exist. One example of a possible approach is to provide these capabilities directly in each simulation of a modular simulation. In this approach, the simulations have to synchronise with each other. Another approach is to use a centralised coordination unit which provides these capabilities. This unit is called a coordinator. The coordinator provides capabilities for a common interaction between simulations. These capabilities include the management of simulation aspects. Examples for capabilities are the distribution of information, management of time or notification of simulations about events. A MSE can specify context information to the data and notifications exchanged between simulations. The context information is needed for simulation information to be processable by the coordinator. An example for a context information is the order how updates to a datum have to be delivered. The MSE also has to specify notifications to and from simulations. These notifications enable the use of the coordinator capabilities. For example notifications to call for an advancement of time. The coordinator is currently the only realised approach in the DSL. Nevertheless, we specifically distinguish here between coordinator and MSE. The MSE is understood as the conceptual part used in a modular simulation. The coordinator, on the other hand, is a concrete approach. For each concrete approach, the metamodel has to be enhanced by describing elements. An example of a coordinator is the HLA RTI. A RTI can be represented explicitly by elements in the metamodel. Representation of information contained in simulations and MSEs need to be described to be usable in a modular simulation. The exchange of information could not be specified without such representation. The description of available and required information in a simulation is necessary for simulation modules. The statement of required information is of particular importance. This statement provides insight into the information needed to execute the simulation module correctly.

The information of simulations and MSEs have to be modelled based on computer interpretable data types. This approach enables usage of the models in the implementation of the modular simulation. The specification of the data types must be flexible enough to represent all information contained in a simulation or MSE. This information can be semantically enriched to specify certain aspects in a simulation. Semantically enriched information is, for example, units of time or distance. Another example is the representation of the available working time spans in IntBIIS. An object-oriented scheme is employed in the DSL to represent information of simulations. Data of simulations are represented by classes and their contained attributes due to this scheme. In the DSL, classes are called "object classes", and attributes are called "properties". Also, the notifications of simulations and MSEs have to be defined. These notifications are performed by functions (in the DSL called "operations") and their parameters. Parameters specify additional information transported in an operation. The name "operation" for notifications is chosen because they can be seen as similar to a call of a function in a program. One goal of the DSL is to provide reusable descriptions of simulations and MSEs. That goal is reflected in the design of the DSL to create descriptions of simulations and MSEs independently. This description entails that no knowledge of other simulations or MSEs used in the modular simulation is needed for their design. Therefore, the information and data types confined in simulations or MSEs are also specified independently. Three reasons motivate the property of independent development. The first reason stems from the extraction of simulation modules out of monolithic simulations. Each simulation module is still using the schemes of the monolithic simulation (e.g. individual data types). Furthermore, the goal the independent specification increase the reusability of models created by third parties. The designed models are easier applicable to other modular simulation, when they are created independent of the currently used MSE or other simulations in a modular simulation. The third reason stems from parallel and specialised creation and use by different developers and teams. Different developers can focus on their field of expertise (e.g. creation of simulations, MSE or assembly of modular simulation). Other developers can use the developed content without the implementation of the content by themselves. If simulations were directly based on a MSE, the developer would have to fit its products on the MSE. The independence in development provides the benefit of sole responsibility as it is assumed with components described in Sec. 3.2.1.

The goal of the independent creation of simulations and MSEs induces the potential problem of incompatible or conflicting information. This problem originates from the possible different design decisions for information contained in simulations and MSE. For example, two simulations incorporate the same concept (e.g. speed). Because of different naming conventions or design decisions, different names are used for this concept (e.g. speed and velocity). If one simulation receives the information of this concept with another name (e.g. values), it cannot be processed. Another example is the different representation of "percent". One simulation can use an Integer (values: 0 to 100) and the other a Double (values: 0.00 to 1.00) to represent percent. If both simulations are chosen to interact, they cannot (correctly) interpret the received value. The selection of only compatible descriptions would greatly reduce the number of possible selectable simulations and MSEs for a modular simulation. Therefore, the DSL provides the approach of "adaptation" to mitigate the effect of conflicting or incompatible information. The adaptation approach can be used to enable interaction between simulations of a modular simulation containing differently designed information. The assembly of a modular simulation can, therefore, be described with the content described above.

The simulation modules of a modular simulation can either be simulation features or modular simulations. Simulation features do not consist of other simulations. The use of other modular simulations creates a hierarchic structure. Simulations and MSE approaches are confined in components to be used in the DSL. The components of simulation features and modular simulations do not appear different in the assembly of another modular component. This property enables the use of simulation features or modular simulations interchangeably. A simulation module can consist of one or multiple models. Thus, a simulation can represent a single-model system or a monolithic system (i.e. multiple models confined in one system). This design supports an evolutionary style. The monolithic simulation can be used as a component. A modular simulation can be created when the (sub-)models of the monolithic simulation are decoupled and confined in independent simulations. This modular simulation can then be exchanged. The description of the required and provided information of a component also supports the ability to exchange components. The DSL employs an interface scheme for the definition of a components provided and required information in the modular simulation. Interfaces define information to be exchanged. A component can mark an interface as provided or required. This mark signals that the component provides or requires the information specified by the interface. Every information to be exchanged in a modular simulation has to be defined in an interface first. This enables to exchange simulations in a modular simulation with the same required and provided interfaces. The modular simulation has to define the information contained in unsatisfied required interfaces as required by itself. This definition allows the deference of the provision of the information to a simulation outside of the modular simulation. Also, the information the modular simulation can provide has to be described. This description allows to use of the modular simulation for information provision in another modular simulation.

Fig. 6.1 exemplary shows the containment of the simulation features Palladio and business process in the modular simulation IntBIIS with provided and required interfaces Components can also contain tools. Tools are represented like simulation features with

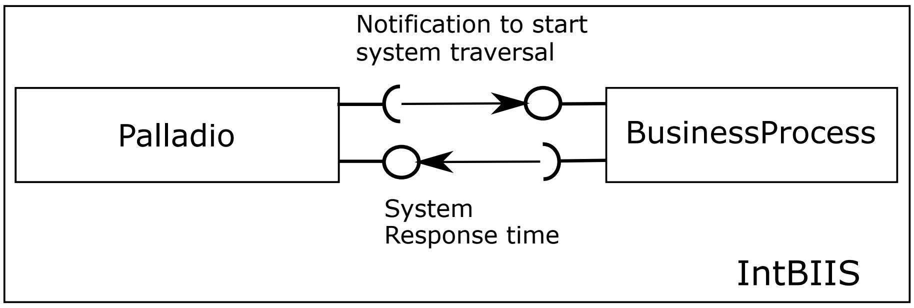


Figure 6.1: The representation of the modular simulation IntBIIS. The half-circles represent provided information according to interfaces. The circles represent the required information according to interfaces. The connection between two elements define the connection between the corresponding information interfaces

required and provided information. Tools can, for example, be data or time logging tools. Due to their confinement in a simulation component, they also have to apply the requirement and provision scheme of simulations. The nested structure of modular simulations creates a hierarchy. One logical consequence is that the lowest level of the hierarchy only consists of simulation features.

The DSL also has to provide the capabilities to define the structural aspects of modular simulations. This structure consists of information on how the components in the modular simulation are connected. This connection includes how the information in the modular simulation flows. Two types of connections between components are differentiated in the DSL. One connection specifies the relationship between the required and provided interfaces. A required interface of one component can be connected to a provided interface of another component when they use the same abstract interface. This connection allows checking of whether each required interface is satisfied by a provided interface. If a required interface cannot be satisfied, a connection to the modular simulation itself has to be specified. This is called a delegation connection in the DSL. This specification enables the definition of the required information by the modular simulation itself. Another connection is the flow of information in the modular simulation. Each component contains connectors to be connectible with other components. In an implementation, these connectors contain the capability to transfer and receive information (e.g. by references or by network capabilities). These connectors define the interaction points between the simulation components and MSE components. Only one connection between the underlying model (i.e. simulation or MSE) of a component and the component itself has to be implemented. Also, only the component has to provide the logic for interaction between the modular simulation and the independently developed containments. Therefore, the component is the interface between independent designed elements and the assembly of the modular simulation. Through this design, the independently developed elements are separated from the modular simulation. The notifications to access the capabilities of MSEs have to be specified in the connector. This specification includes notifications from simulations to the MSE. Also, the notifications from the MSE to the simulations have to be specified. This specification enables the simulations to access the capabilities of the MSE and the MSE to provide information to the simulations (e.g. updates or other notification). Each simulation has to establish a connection to the coordinator to enable interaction. This connection is realised between the connectors of the components.

For every modular simulation at least one MSE has to be used to enable the interaction between simulations. The same MSE can be used in different modular simulations. The definition of the DSL also allows to define and use different MSEs in one modular simulation. An exemplary structure of a hierarchical modular simulation is shown in Fig. 6.2. Here, the connections between components and connectors specify the information flow. They do not describe provided and required information. Before the metamodel is discussed in

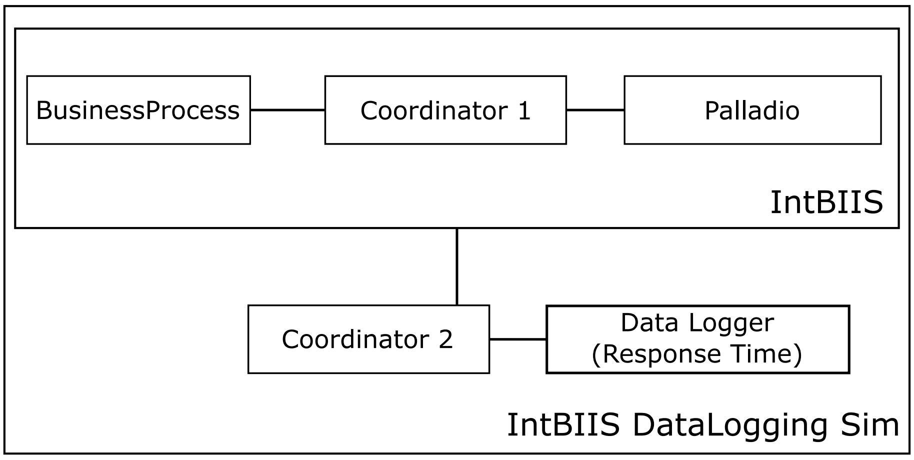


Figure 6.2: The structure of a hierarchical modular simulation. The lines represent the connections between components and define the information flow

detail, the approach of "adaptation" is provided in the following section.

6.2 Mitigation of Information Incompatibilities through Adaptation

One obstacle in the assembly and reassembly of simulations are differences in the exchanged information. There are several approaches to eliminate or reduce such differences. For example, Tolk [23] approaches this problem by presenting multiple levels of interoperability as described in Sec. 2.2.4. One level is related to structural, declarative and semantic conflicts. The use of a reference model solves the first two conflicts. Another approach is to provide a common object model for a modular simulation. Every simulation has to use the content of this object model. The drawback of this approach is that all simulations in a modular simulation have to conform to this object model along with its included representation of information.

The DSL provides an approach to mitigate incompatibilities between information of simulations in a modular simulation. For this approach, the DSL utilises the idea of adapters. The approach is similar to the mapping approach of Neema [64]. The DSL provides an independent specification of abstract description for adaptation rules. The descriptions specify what information has to be adapted and how the adaptation is executed and are called *adaptation descriptions*. Abstract names represent the information to adapt. These names are called "markers" in the DSL. For example, two interacting simulations exchange values of time. However, time is represented as minutes in one simulation and as seconds in the other. Both simulations would interpret the exchanged values for time wrong. For this purpose, two markers are created. One marker contains the name "minute" and the other "second". Additionally, a developer specifies a rule for how to transform the described information. We call this an "adaptation conversion". A conversion stating to multiply or divide the value by 60 would be sufficient in the time-example. This example is pretty fundamental and could be resolved by other means. Nevertheless, it is used to clarify the idea of the approach. The DSL provides capabilities to structure the abstract markers by use of specialised *adaptation descriptions*. These structures described by *adaptation descriptions* are supposed to reduce the number of descriptions in relation to a definition of every adaptation as a one-to-one relation. Different identified structures are predefined and provided by the DSL. One example is the structure of SI-Units. SI-Units define a "base unit" for each represented quantity (e.g. second for time units). Every other related unit is defined as to be calculated from this unit. This structure is represented in the DSL.

To enable the adaptation in the DSL, a component in a modular simulation can describe contained entities to execute the described adaptations. These entities are called *adapter services*. The location of these *adapter services* in a component reduces the dependencies between an independent designed simulation or MSE to the modular simulation. The location in a component enhances the reusability of independent designed content in multiple modular simulations. The location of an adapter service in the components enables adaptations to be usable centralised in a coordinator, or locally in a simulation. These locations are different from the used mapper in the approach by Neema [64]. Here, the mapper is an independent tool, connected to a RTI. For a tooling approach, the information has to be transferred to the mapper. Neema [64] used this design to provide an approach without altering or encapsulating a HLA. An encapsulation is however already modelled in the DSL. So, we decided to design the adaptation approach without transferring all information to a component. This design is also based on the reason of possible network communication. If all information in a modular simulation has to be transferred to one component performance problems can arise. Also, if the component fails due to some reason (e.g. the tool experiences and erroneous state), the whole modular simulation cannot be adapted. In an information exchange, the adapter service verifies if an adaptation description can be applied in the exchange of information. It also executes the conversion.

A goal of the DSL is to facilitate the independent specification of the adaptation descriptions. Therefore, the descriptions are designed independently of concrete information in modular simulations. This independence is realised by the abstract markers defined before. In the independent form, the abstract description cannot be applied in a modular simulation. Therefore, the descriptions have to be connected to the adapter services to be usable in modular simulations. While connected to a particular adapter service, the information contained by the described simulations and MSEs are connected to the markers. The identified types of adaptations are presented in the following Sec. 6.2.1. A proposition of an exemplary realisation of a step-wise adaptation process is provided in Sec. 6.2.2

6.2.1 Adapter Types

Adaptation is used to resolve conflicts in the information of different simulations. The term "conflict" is taken from Tolk and Muguira [23] where information differences are assigned to four conflict classes. Tolk and Muguira [23] state that a common reference model can resolve descriptive and structural conflicts. However, such a reference model cannot be assumed by independently developed simulations or modules extracted of different monolithic simulations. Therefore, the adaptation approach uses these two classes. Descriptive conflicts describe different names for the same concepts and exchanged information [23]. The provided adaptation approach can solve this conflict. Markers for the names of the information for each adapted concept are provided. A conversion can be described to translate the names used in the simulation specified by the prior define markers. One example is the use of different languages (e.g. "hello" in different languages). The second conflict class refers to structural conflicts. This concept incorporates two different problems. One is the use of different structures. One underlying structure can be transformed into another in the adaptation (e.g. a collection "list" to an array). The other problem is stated, that one concept uses an attribute, the other a reference to another concept. This problem can be resolved by describing a requirement or provision of another information. Furthermore, a conflict class is identified as differences in information by their underlying representation. Here, one simulation uses a different data type to describe a concept than another simulation. An example for this is given in Sec. 6.2 with the representation of "percent". Adaptation could also be used to adapt the representation of time in several simulations. This adaptation can be applied when simulations use different representations of simulation time like continuous, discrete or mixed time. Nutaro [82] shows, that it is possible to transform equation based continuous time models into discrete time or discrete states. However, Nutaro [82] also explains that "discrete event simulation of continuous systems is an active area of research". Nevertheless, because of the possibility of such an approach, adaptations could transform the continuous represented information into discrete representation. The DSL could provide an adaptation description to identify the time scheme of a simulation. For this time scheme, the approach of continuous time discretisation can be described. However, this is a theoretical assumption, and further research has to be provided in future works on this topic. Filtering or transformation of information can also be a adaptation type. This type can be used to convert the information contained in a simulation to another type. For example, one simulation needs the number of active actors in IntBIIS. IntBIIS provides only the actors confined in a list. An adaptation description can be realised, where the conversion describes the calling of the .size() method of the list type. Another application is the transformation of not transferable information by the MSE scheme. For example, only data types like String or Integer can be exchanged between simulation. One simulation uses a collection of objects of a specific class. The adaptation approach could be used to transform the objects into string representation. Another kind of filtering is the provision of information with certain underlying conditions. These conditions can possible not be represented by other schemes in the modular simulation. For example, a simulation only requires actors with an active state. This requirement cannot be expressed by the current scheme of expressing required information. Therefore, an adaptation could be specified to provide only the active actors.

6.2.2 Adaptation Process

Adapter services and adaptation descriptions can be used after their specification and connection to the modular simulation. We propose the following conceptual approach for the adaptation as depicted in Fig. 6.3. The process starts when information is exchanged. A process detects one or several correspondences in information according to the adaptation descriptions. Found correspondences are resolved with the application of the conversion in the description. It has to be checked if new correspondences appeared due to the transformations. For example, stunde (English for hour) is transformed to hour. In the simulation, all hours are represented as seconds. Therefore, hour has to be transformed to seconds. The process is finished for one information when no more correspondences are found. This procedure allows a chained transformation of the information received from the source to the one expected in the destination. We provide no reference implementation of this process realisation. Each implementation of adapter services can be individually performed. This approach enables the implementation of the concepts in a design of the developers choice.

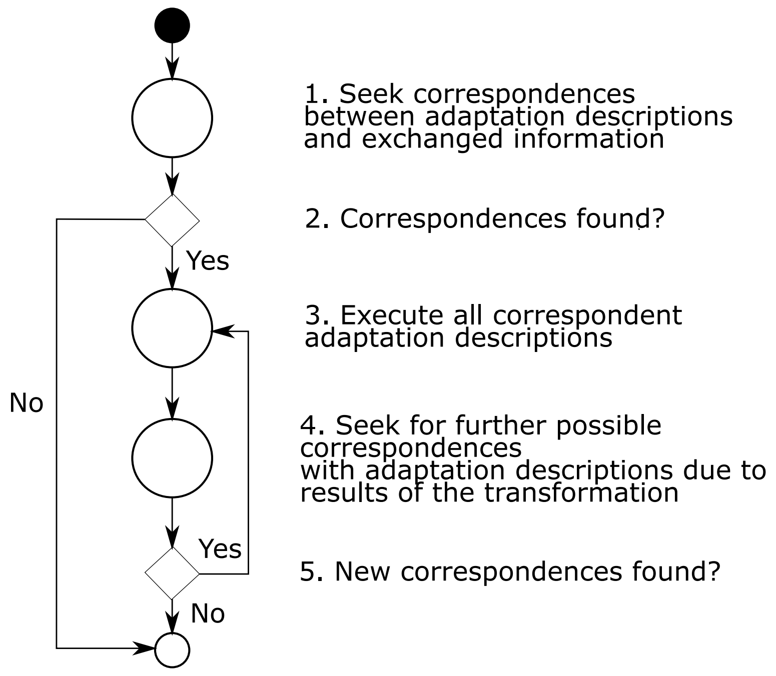

1. Seek correspondences between adaptation descriptions and exchanged information
2. Correspondences found?
Yes
No
3. Execute all correspondent adaptation descriptions
4. Seek for further possible correspondences with adaptation descriptions due to results of the transformation
Yes
5. New correspondences found?
No

Figure 6.3: The proposed conceptual process of adaptation. Large round circles are actions, arrows mark the control flow. The diamonds (rotated squares) depict decisions. The small black circle depicts the start and the small white circle signals the end

6.3 Package Structure of the Metamodel

The metamodel of the DSL is used to describe modular simulations. This metamodel is structured by multiple superordinate packages. Every superordinate package except of the *basic* package in the DSL, realises elements and capabilities of modular simulations as described in Sec. 6.1. The *basic* package provides basic capabilities to identify model elements. It is taken off the Karlsruhe Architecture Maintainability Prediction (KAMP) [83] metamodel. In the remainder of this Sec. 6.3 and its subsections, the package structure of the metamodel is presented. The DSL provides the capabilities to describe simulation features in the *SimulationFeature* package. MSEs can be described by elements in the *ModularEnvironment* package. Both packages contain elements to the *DataRepresentation* to define their data. The package contains the elements to describe data types and the information contained in basic simulations. This description includes the capabilities to describe operations and data. The metamodel elements to describe the independent parts of the adaptation approach are located in the *Adaptation* package and its sub-packages. The *ModularSimulationAssembly* package and its sub-packages provide all capabilities to describe an assembly of modular simulation. The elements in these packages use the models created with the prior mentioned packages. A fine granular package structure allows the provision of an overview of the topics in the DSL to each metamodel element. Also, the packages confine highly semantically or logically coupled elements. This structure makes extraction of the responsibilities of the packages into separate projects easier. Extraction into different projects enables the reuse and separate evolution of the package contents. For example, the packages *Adaptation* package could be reused in a project to provide more exact descriptions of conversion. Also, this project could be maintained by one developer team without possibly breaking the use in the DSL. The inclusion in other projects is a case for its reuse.

The inspection of the package structure is provided as block-diagrams. The dependencies between each package are marked. Two types of dependencies can describe a relation. One dependency is the sub-typing of an element of another package (triangle-arrow with white head). The other dependency signals the usage of an element of another package. This using-dependency is marked by a black arrow. A bidirectional (marked with two arrows) connection between two packages signals a mutual dependency between the packages. If the direction is unidirectional, the target package does not need elements of the opposite package. The structure of superordinate packages is described including their directly contained packages in the following subsections. This description is used to provide a general overview. The dependencies of the packages are marked. However, in the overview, no specification of the number of dependencies is provided. Packages not containing inner packages are marked in grey. After the provision of the overview, the packages are, and different design decisions are provided. Additionally, alternative packaging approaches are discussed when deemed useful. Only the package structure is discussed in this section. A finer description of the meta model is provided in Sec. 6.4 to Sec. 6.8.6

6.3.1 Superordinate Package Structure

Fig. 6.4 shows the structure of the superordinate packages including their directly contained sub-packages. Every subordinate package contains sub-typing elements of the *basic* package. This shows the utility characteristics of the *basic package*. The independence between the packages *ModularEnvironment* and *SimulationFeature* is visible. Their contained elements directly reference (i.e. use) the content of the *DataRepresentation* package. This relation is marked by the black arrows. Therefore, the content of *ModularEnvironment* and *SimulationFeature* can be used independently of each other. The elements of *ModularEnvironment* and *SimulationFeature* directly use the content of *DataRepresentation*. This relationship shows a dependency of capabilities to define information. This dependency, however, is due to the information-centric design of the DSL. The elements of *DataRepresentation* do not contain elements of other packages. Only sub-typing relations to other packages exist. The contained elements only sub-type elements of *Adaptation* and *ModularSimulationAssembly*. The sub-typing is related to the *Adaptation* package, because the aims to adapt information. The *Adaptation* provides one element to be sub-typed by all adaptable classes to realise this ability conveniently in the metamodel. This element explains the sub-types relation. The design of the *DataRepresentation* shows strong independence from all other packages. Therefore, this package could be a good candidate for extraction into its project. This extraction would allow to maintain, evolve and reuse this package in a more isolated fashion. Also the *Adaptation* package does not have outgoing references except to the *basic* package. This property illustrates the independent use of the *Adaptation* approach. The design of *Adaptation* package allows

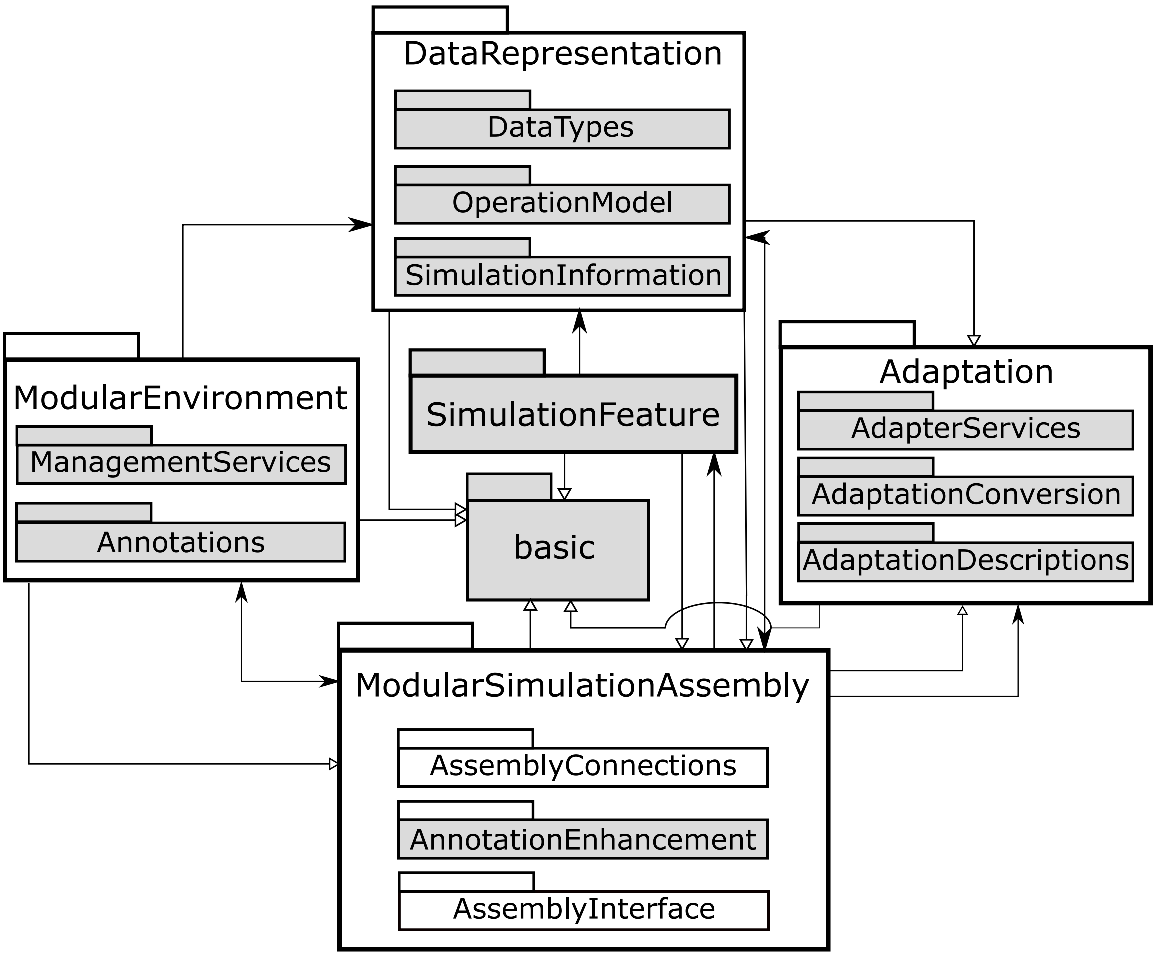


Figure 6.4: Superordinate package structure of the DSL. Packages are depicted including the directly contained packages of the superordinate packages. Boxes containing text are packages and their names. Grey box filling signals no further packages containment. White box filling signals further contained packages.

extracting this package for better reuse. This package is a good candidate to reuse in other metamodels to provide the adaptation capabilities. Also, an evolution independent of the DSL is possible. The assembly character of the *ModularSimulationAssembly* package is shown in the superordinate view. Its contained elements directly use other elements of all four remaining packages in the *DSL* metamodel. Sub-type references exist to the *basic* and the *Adaptation* package. The high coupling of the *ModularSimulationAssembly* package to the other packages shows that its content is explicitly designed for the use of the content of the other packages. The *ModularSimulationAssembly* package cannot be used in another metamodel without the other packages. This property stresses the assembly capability of this package.

In the following subsections, further dependencies between different packages are discussed. For this purpose, the dependencies are described in finer granularity. Dependencies between directly contained elements of the superordinate package to elements of inner packages are also marked in the figures of the following subsections. This description is done by a connection (i.e. arrows) of the border of the package symbol to the contained package.

6.3.2 Dependencies between the Packages SimulationFeature, ModularEnvironment and DataRepresentation

The elements of the *ModularEnvironment* package and the *SimulationFeature* package use the elements of the *DataRepresentation* package. This usage is emphasizes the information providing nature of the MSE and simulation features. In this section, the inner-package-dependencies of the packages are inspected. Also the inter-package-dependencies between *ModularEnvironment*, *SimulationFeature* and *DataRepresentation* are discussed. Furthermore, possible alternatives are discussed. On this basis, it is explained why the current structure is selected. The package structure of these three superordinate packages is presented in Fig. 6.5. In this figure, the inner- and inter-package dependencies are visible.

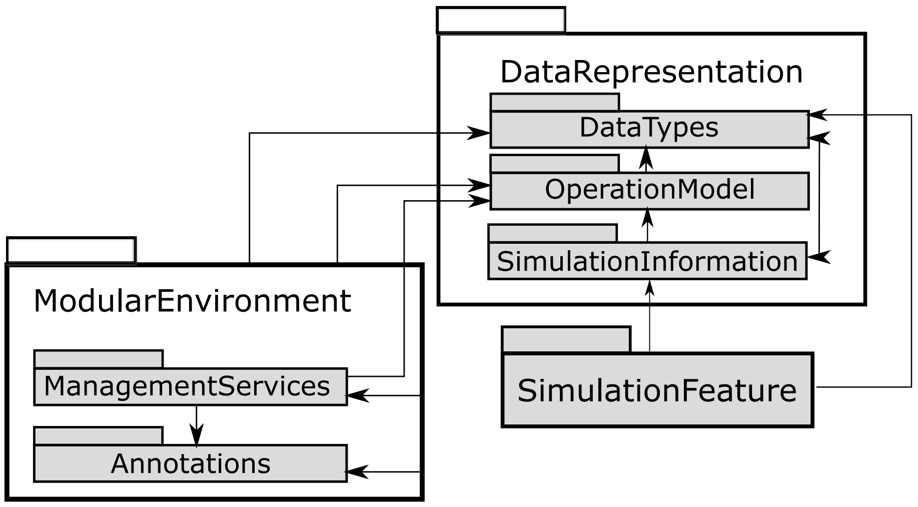


Figure 6.5: Package structure of the three superordinate packages *ModularEnvironment*, *SimulationFeature* and *DataRepresentation*. Dependencies between the elements contained in a superordinate package and the contained packages are marked by a connection between the package borders

Because *SimulationFeature* does not have inner packages, no inner-package dependencies can be shown. Every package has its distinct use in the DSL. The absence of further packages in *SimulationFeature* illustrates the data-centric nature of this package. The package content is used to design simulation features. The current goal of the DSL placed on the information contained in a simulation. Therefore, the content of *SimulationFeature* only contains elements to describe a simulation together with its contained information. This approach is also visible by inspecting the dependencies between the *SimulationFeature* and *DataRepresentation* package. Elements of the *SimulationFeature* package use elements of the *DataTypes* and *SimulationInformation* packages. The *DataTypes* package provides the capability to define the types of information. This capability is needed to build models of information from data types interpretable by a computer. The *SimulationInformation* package contains elements to define information contained in basic simulations. The *SimulationInformation* uses the *DataTypes* to be able to describe data on the basis of data types to be understandable by computers. The elements of the *Operations* package enable the modeller to represent notifications which the simulations and MSEs can send and receive. The *DataTypes* package has to be used because of the independent development aspect of basic simulations. The instances of the elements contained in *DataTypes* have to be specified for every simulation feature to be usable. For this purpose, the *SimulationFeature* package uses elements of the *DataTypes* package. Also, the *SimulationFeature* packages uses elements of the *SimulationInformation* package. The elements of this package are utilised to represent information like object classes with properties and operations with parameters. The independent description of a MSE creates the need to specify own data types too. Therefore, the *ModularEnvironment* package uses content of the *DataTypes* package. The model elements directly contained in the *ModularEnvironment* package use elements of the inner *ManagementServices* and *Annotation* packages. The *ManagementServices* package provides elements to describe the capabilities of a MSE. The elements of the *Annotation* package can be used to specify the context information required from simulation data and notifications. The *ModularEnvironment* package as well as the *ManagementServices* package use elements of the *OperationModel* package. This usage signals, that one package has to create element instances and one package references it.

6.3.3 Overview of the Adaptation Package Structure

The *Adaptation* package contains all elements to provide the realisation of the adaptation approach. The package structure is depicted in Fig. 6.6 It is evident that the *Adaptation* package contains centralising elements in this approach. There is a bidirectional dependency between the *Adaptation* and the *AdaptationDescriptions* package. This dependency signals a definition of elements in one package and usage in the other. Also, the *Adaptation*

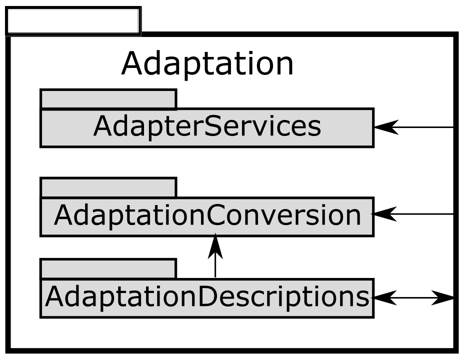


Figure 6.6: Package structure of the superordinate *Adaptation* package

package and the *AdaptationDescriptions* package use elements of the *AdaptationConversion* package. The purpose of these references is for one package to define the elements and for the other to use them. The *Adaptation* package also references the *AdapterServices* package. Therefore, a connecting capability is signalled by using all three contained packages. Due to the use of all packages, the *Adaptation* package is self-contained and therefore can be extracted for a separated use.

6.3.4 Package Structure of ModularSimulationAssembly

The modular simulation assembly is designed to combine all models created with the other packages in the metamodel to a modular simulation. Therefore a high coupling to the content of the other packages can be expected. The dependencies related to the *ModularSimulationAssembly* package are shown in Fig. 6.7. The figure shows the inner packages and intra-package dependencies. Also, the other packages directly used by the *ModularSimulationAssembly* package or its inner-packages are depicted.

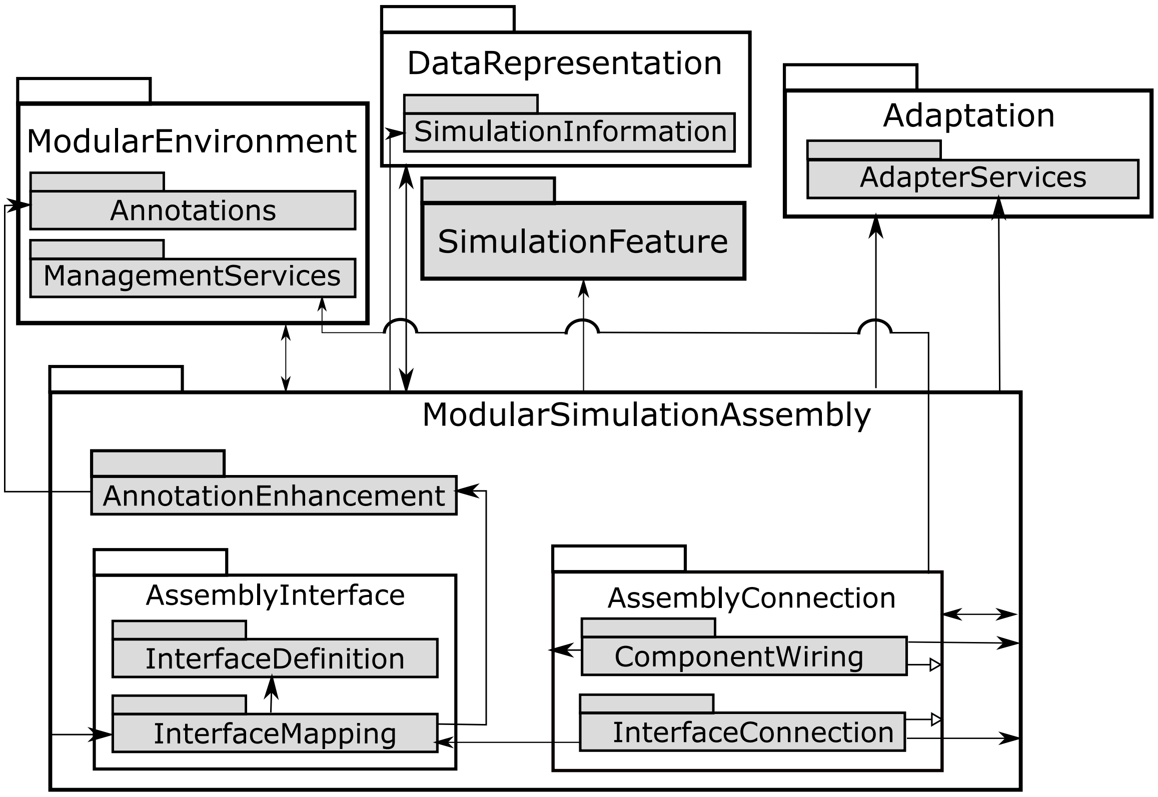


Figure 6.7: All packages and dependencies related to the *ModularSimulationAssembly* package and its sub-packages

ModularSimulationAssembly shows high dependencies on every other package in the DSL. These dependencies are expected due to its assembly nature. The *ModularSimulationAssembly* package itself directly contains elements. This containment can be seen by dependencies of the *ModularSimulationAssembly* package to its inner packages. The elements of *ModularSimulationAssembly* package directly use elements of the *ModularEnvironment* package and *SimulationFeature* package. Also these packages use elements of the *ModularSimulationAssembly* package themselves. This bidirectional relation provides a hint of a bidirectional relationship of elements in the packages. This usage can be explained by the component approach employed by the modular simulation. Also elements of the

Adaptation package and *AdapterServices* package are used. The reasons for this relation can be found in the explanation of the adaptation approach in Sec. 6.2. The containment of adapter services explains the dependency of *AdapterService* package to components defined in the *ModularSimulationAssembly*. Also the adaptation descriptions have to be bound to the adapter services and the information of the modular simulation. Therefore, the package content of *Adaptation* is used. The *AnnotationEnhancement* package is used to enrich independent designed data and notifications of simulation features with context information. The MSE defines the required context information. Therefore, this package uses the elements of the *Annotations* package. The elements of the *SimulationInformation* package is used to specify the context information for the data and notifications provided by basic simulations. Contextually annotated information is seen as available in the modular simulation. The *AssemblyInterface* package is used to define and use interfaces to represent provided and required simulation information in the modular simulation. The definition of interfaces is realised with the content of *InterfaceDefinition*. The *InterfaceDefinition* package does not contain outgoing dependencies because this definition is realised independent of concrete information. The content of the *InterfaceMapping* package is used to define interfaces as provided or required for simulations in the modular simulation. Also, it maps the information to the interfaces. Therefore, the mapping has to use the content of *InterfaceDefinition* to specify the applied interface. It depends on elements of *AnnotationEnhancement* to map the available information of simulations in the modular simulation to the interfaces. The content of *InterfaceMapping* is used in components to define their required and provided interfaces. Therefore a dependency of the *ModularSimulationAssembly* to the *InterfaceMapping* exists. The *AssemblyConnection* package provides capabilities to define different types of connection in the modular simulation. This includes the definition of connectors and connections. A connector has to include the information what notifications can be sent and received from and by the MSEs. Therefore, the *AssemblyConnection* has a connection to the *ModularEnvironment* package. In this package collections of the operations realising the capabilities are provided. The bidirectional dependency between *ModularSimulationAssembly* and *AssemblyConnection* signals the use of a element connecting the modular simulation and the content of the *AssemblyConnection* package. This is emphasized by missing dependencies of *ModularSimulationAssembly* to the inner packages of the *AssemblyConnection* package. Also the inheritance dependencies of the inner packages to elements directly contained in *AssemblyConnection* indicates this relation as well. The *ComponentWiring* package is used to describe the information flow connection between components. Thus, the possible information flow between components can be modelled. The definition of components is done in the *ModularSimulationAssembly* package. Therefore, the *ComponentWiring* package uses the connecting capabilities of *AssemblyConnection* and the *ModularSimulationAssembly* package to model such connections. The *InterfaceConnection* contains the elements to describe the connection between required- and provided-interface definitions. This connection describes which required interface is satisfied by which provided interface. The elements of the *InterfaceMapping* package have to be used to specify the required and provided interfaces. Also, the components used in the interface mapping have to be marked. This relation also explains the dependency to the *ModularSimulationAssembly* package.

6.4 Basic Metamodel Classes

Model elements often require to be identifiable by the the system or users. The provision of super classes with identification capabilities (e.g. as attributes) avoids the need for replication in each new metamodel element. Super classes also provide a way to design reusable capabilities. Also such classes enable a centralized maintenance. These classes can then be sub-typed by all metamodel elements that require these capabilities. The classes for provision of identification capabilities for model elements in the DSL are located in the *basic* package. The package content is depicted in Fig. 6.8.

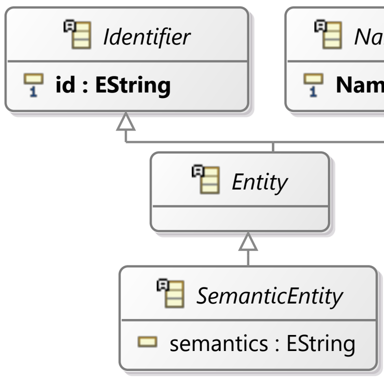


Figure 6.8: Content of the metamodels *basic* package

The abstract class *Identifier* contains the attribute *id:EString*. The value of *id* is set to a unique string when a subtype of *Identifier* is instantiated. This unique string allows to uniquely identify each model element. The generation of this unique id is realized by application of the code-line $setId.(ECore.generateUUID())$ in the metamodels implementation source code. The abstract class *NamendElement* contains the attribute *name:EString*. The *name*-string allows the provision of a human-readable self-defined name to a model element. Human-defined names let the modellers better identify the model element than a random string. The abstract *Entity* class sub-types *Identifier* and *NamendElement*. This sub-typing provides the model element with attributes to describe a name and a unique identifier. These two attributes allow the definition of model elements with both identifiable capabilities. These capability are necessary because the *name* cannot be restricted to be unique. The *id* allows differentiation between model elements even when they have the same assigned value for *name*. The abstract class *SemanticEntity* sub-types *Entity*. It extends the capabilities of *Entity* by the attribute *semantics:EString*. The *semantics* attribute allows to provide semantic information about an identifiable entity in a simulation. The *Identifier*, NamendElement and *Entity* classes are reused elements of the KAMP metamodel [83]. *SemanticEntity* is an addition for a semantic description of the entity.

Many of the elements in the DSL-metamodel either sub-type *Entity*, *SemanticEntity* or *Identifier*. The metamodel of the DSL is presented and described in the following section. Mentioning of these super-types would increase the text-volume of this chapter unnecessarily. Nevertheless, knowledge of the of attributes and capabilities of a metamodel element is necessary. Therefore, tags are used in the text to annotate the elements sub-typing one of the classes of the *basic* package. The tag [I] stands for *Identifier*, [NE] for *NamedElement*, [E] for *Entity* and [SE] for *SemanticEntity*

6.5 Representation of Information

The DSLs' models of simulations and MSEs have to describe their contained information to be usable in an implementation of a computer program. Therefore, the information has to be represented to be understandable by programs. Two types of information are required in the DSL. The first type is representing the state of a simulation (hereafter called data). This information is exchanged and modified by other simulations. The second type enables components in the modular simulation to send and exchange notifications to each other. Notifications do not persist over time. Therefore, when a notification is received, it cannot be its contained information cannot be reused. Simulations and MSEs can react on the notifications (e.g. schedule events or execute capabilities). The underlying types of data have to be described to be representable in a computer. The metamodel capabilities of the DSL enabling the description of data is presented in the remainder of this section.

6.5.1 Data types

Data types are used in the DSL to represent the information of simulations and MSEs on the basis of types processable by a computer. Sec. 6.1 provides reasons why it is necessary to enable an independent modelling of simulation features and MSEs. For this purpose, every model of a simulation or MSE has to describe its used data types. Data types in the DSL are not only understood as types like Integer or String. Data types can be semantically enriched types on the basis of such types like Integer or String (e.g. Units like second or a type to limit capacity of a device resource). The data types used in MSEs or simulations can be described in the DSL by the elements of the package *DataType*. Its structure is depicted in Fig. 6.9

Each data type is modelled as an element of the abstract class *DataType*[SE]. The *DataType* instances are containments of the *DataTypeContainer*[SE] metamodel class. This collected definition enables centralised maintenance and collected reuse of its elements. The reuse enables developers to use the model of *DataTypeContainer* with the same *DataType* instances in different models. Only the *DataTypeContainer* description has to be migrated for its reuse. An example for the application of this approach is the creation of a product line for a modular simulation. The reusability removes the necessity to model the *DataType* instances again for each product. *DataType* is sub-classed by the abstract class *ClassicalDataType* and the *Unit* class. The abstract *ClassicalDataType* class represents data types and other containments found in programming languages. The term containments is used for structures of data types like maps or collections. The super type *ClassicalDataType* enables an easier definition of interleaved data types (e.g. to model a collection of collections). To define an interleaved type, the subtype of *ClassicalDataType* only has reference or create instances of the *ClassicalDataType* class. Assumptions on the existing basic data types of programming languages have to be made. These assumptions

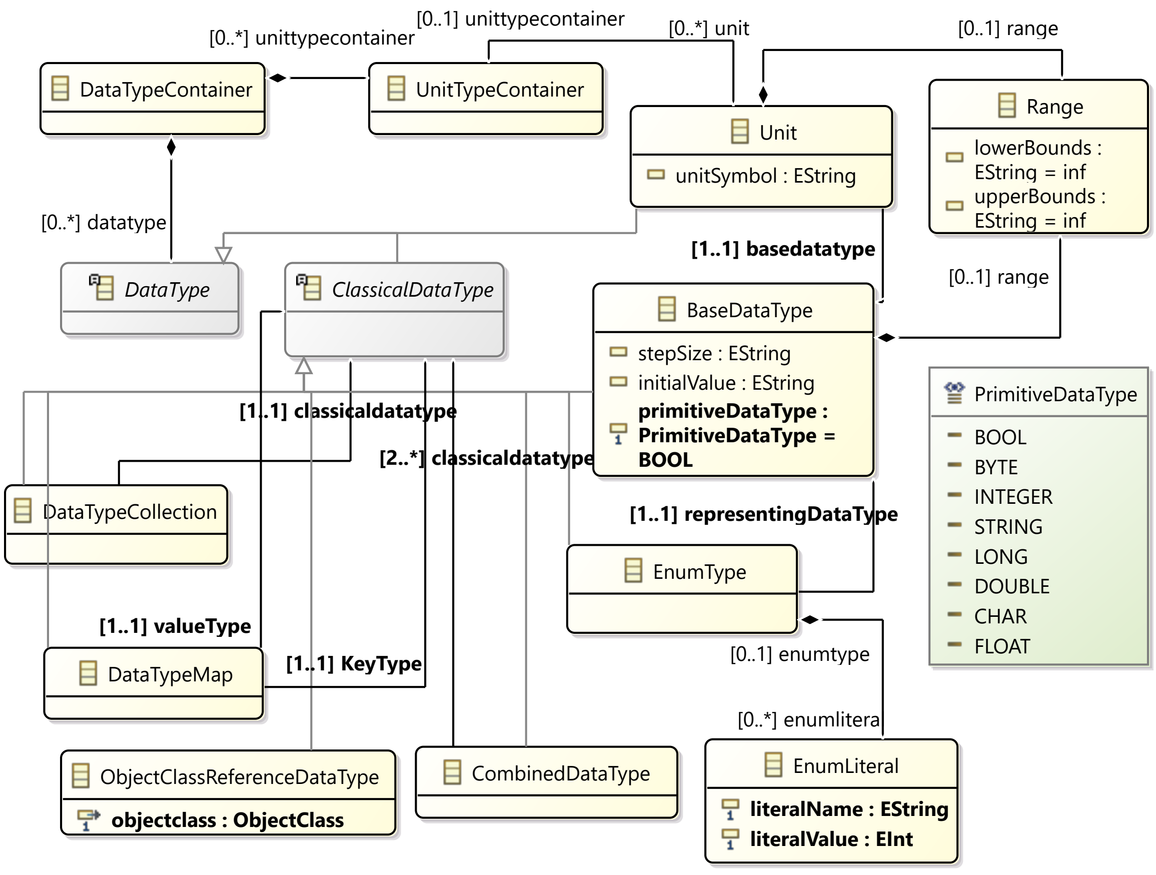


Figure 6.9: Classes and Relations Contained in the *DataTypes* Package

are necessary because not all programming languages support the same data types. An alternative would be to identify a minimal set of data types available in all programming languages. This identification would, however, restrict the DSLs convenience. We call the selected types "primitive data types" (hereafter called primitive types). These selected primitive types are defined as literals in the *PrimitiveDataType* Enum element. The primitive types are Byte, Integer, Boolean, Long, Float, Double, Char and String. These primitive types are used to model *BasicDataType* instances. This class subtypes *ClassicalDataType*. *BasicDataType* is used to provide the basis for the representation of all information in the DSL. The *BasicDataType* contains the enum attribute *primitveDataType:PrimitiveDataType* to signal its underlying primitive type. The assumption of primitive types results in the problem that some simulations cannot use certain types. This problem exists because some assumed primitive types are not natively build-in in specific programming languages. One example is the boolean data type in the context of the programming language C. The DSL provides capabilities to model such data types on the basis of other primitive types. One of these capabilities is found in the description of the *BasicDataType* itself. The DSL provides the ability to specify values a *BaseDataType* can assume. This is realized by the attributes *initialValue:EString*, *stepSize:Estring* and the *Range* metamodel class. *initialValue* defines the value a *BasicDataType* represents, when no value is specified (e.g. 0 for Integer). The *stepSize* attributes describes the value the *BasicDataType* assumes by an "increment by one" (e.g. 1 for Integer). Only textual values can be provided for the attributes *initialValue* and *stepSize* because the primitive type of a *BasicDataType* cannot be directly interpreted in the metamodel of the DSL. It is possible to use an editor for the *DSL* or by application of the Object Constraint Language to restrict the content of these attributes. This would allow to restrict the String of *initialValue* and *stepSize* to fit the selected *PrimitiveDataType* enum literal. However, this approach is currently not realised.The *Range* element contains the attributes of *lowerBounds:EString* and *upperBounds:EString*. The lowest and highest values of a *BasicDataType* can be described with these attributes. The definition of *initialValue*, *stepSize* and *Range* allows to describe data types not built-in in a programming language with available primitive types. As an example the data type of Boolean in the programming language C can be modelled. If a simulation is written in C, the modeller can model a Boolean on the basis of an Integer. For example by creating a *BaseDataType* with the *name* attribute set to "Boolean". The *primitivDataType* attribute can be set to *INTEGER*, the *stepSize* to "1" and the *initialValue* to 0. Then a *Range* instance can be created with the value "0" for the *lowerBounds* attribute and "1" for *upperBounds* attributes. The semantics of *BaseDataType* can state: "0 stands for false, 1 for true". The restriction of values can also be used to create semantic enriched data types based on primitive types. One example is the restriction of values for the capacity attribute of the possible work durations for *actor resources* in IntBIIS. For example a Double data type can represent the work durations with a value range from 8.5 to 12.

Data types in programming languages are described by their representation in a computer. Such information is, for example, the size in bits or their endian. The modelling of such information would allow a more defined handling of each *BasicDataType*. This information currently cannot be modelled in the scope of the DSL due to its architecture based perspective. Nevertheless, such information provides ideas for further enhancement of the DSL to enhance its application capabilities. Another type usable in programs is the Enum data type. The Enum type enables a selection of identifiers better readable by humans (e.g. to present the state of a system). Enums provide textual-literals based on values. The human can read the literals, and the computer interprets the values. The application of Enums also allows restricting the values of data. For this purpose, the DSL provides the capability to define Enums. The Enum data type is represented by an extra metamodel element in the DSL. Enums are modelled by the *EnumType* class. This class contains instances of the *EnumLiteral*[I] element. Each instance of *EnumLiteral* defines the attributes *literalName:EString* and a *literalValue:EInt*. The *literalName* attribute represents the literals presented name. These literals enable an easier identification and understandability by modellers. The *literalValue* attribute represents the underlying value. It must be assured by the modeller, that multiple instances of *EnumLiteral* never have the same value for *LiteralValue* in a *EnumType* instance. Every *EnumType* has to reference one *BaseDataType* to provide information about the underlying representation of its defined *literalValue* instances. A capability of a type-system in object-oriented computer programs is to define references to other object classes. In the DSL, a reference to information of a simulation can be used twofold. One possible use is the indirection to another object classes can be used to provide additional information to be transferred. Another possible use of indirection is the potential use of adaptation scenarios (e.g. to translate the data into another format). References to other data is enabled by the DSLs *ObjectClassReference-*

DataType class. This element references *ObjectClass* instances whose explanation is given later in this chapter. Data can be structured in different ways like collections or maps in computer programs. These structures allow the storage and exchange of structured information. The description of such structures is necessary if simulations do not want to exchange a single but similar connected information at once. Collections represent one of these structures. In the DSL the term collection is used for every structure providing multiple instances of a data type at once. Example of such structures are lists or queues but also arrays. A collection is modelled in the DSL by the *DataTypeCollection* metamodel element. This element describes a collection of another *ClassicalDataType* by referencing one of its instances. For example a collection of a *BasicDataType.* The reference of a *ClassicalDataType* also enables to model collections of collections (e.g. 2D arrays). Programming languages also provide capabilities like Structs or Tuples to structure multiple different data types. This structural approach is especially useful to define a connection between two data types. The *CombinedDataType* class represents a tuple-like connection of multiple *ClassicalDataTypes.* This class references two or more *ClassicalDataType* instances for this purpose. Maps can be used in the DSL to enable the connection between elements of two *ClassicalDataType* instances. This design enables the modeller to represent maps, used in simulations, to be represented. One semantic representation of information in simulations are units (e.g meters or seconds). Units provide known and common semantics to their values. These semantics provide a basis for reasoning and interpreting the values (e.g. comparing response times). Because units are commonly used in simulations to provide defined semantics, they are also represented in the DSL. Units are modelled as subclass of *DataType* by the *Unit* class. Real-life units are assigned to a unit symbol for abbreviation (e.g. second -> s). This symbol can be defined by the *UnitSymbol:EString* attribute. In reality, units are expressed with numerical values. The application of units gives these values a semantic meaning. To be represented in a computer, units have to be based on a data type. The using of a data type is realized in the DSL by a reference to a *BaseDataType* instance in the *Unit* class. The *BaseDataType* specifies the underlying representation of a unit (e.g. Integer or Double). For example, the unit second could be expressed as Double (to also define fractions of seconds) or Integer. The *Range* instance and the attributes of *BaseDataType* define the values the unit can assume. Some units, however, are more limited in their range of values due to their semantics. For example, the values of "degree Celsius" range from -273,15 to infinity. The *BaseDataType* instance may have a greater range. For example the referenced *BaseDataType* expresses a general purpose Integer with a range from infinity to infinity. The use of one general purpose data type can be motivated to reduce the data type instances in the model. The range of a unit can be further specified due to this potential difference. The restriction of the values is realized by a containment of zero or one *Range* instance in the *Unit* class. *Unit* instances can be referenced by a *UnitTypeContainer*[SE]. These containers facilitate concise semantics and the connection between units. For example, the units second, minute and hour can be collected in a duration *UnitTypeContainer.*

6.5.2 Operations

The constituting parts of a modular simulation (i.e. the simulations and MSEs) have to exchange information to create a concise behaviour. One of these exchanges is a notification of a simulation or MSE. A notification does not persist over time. Thus the notification is sent and received in an instance of time. Notifications can be designed for the MSE to state the desire to access its capabilities. Also, a notification to a simulation can express the occurrence of a certain event. This event can result in a reaction of that simulation (e.g. change of state or scheduling of an event itself).

In a process-oriented simulation, this event is represented by a call to a function. The notification about an event can be used to schedule an event in a simulation in a DES. The term of operation is used in the DSL for notification. This naming is chosen because of the similarity to starting a process by a call to an function or operation. An example in IntBIIS is the notification that an actor-step uses a software system. The use of a software system is realised by scheduling an event to start its simulation. In a process-oriented approach, this could be a call to a "startTraversal" operation. The DSL provides the capabilities to describe operations by metamodel elements in the *OperationModel* package. The content of the packages is depicted in Fig. 6.10 Developers have to know specifics about the

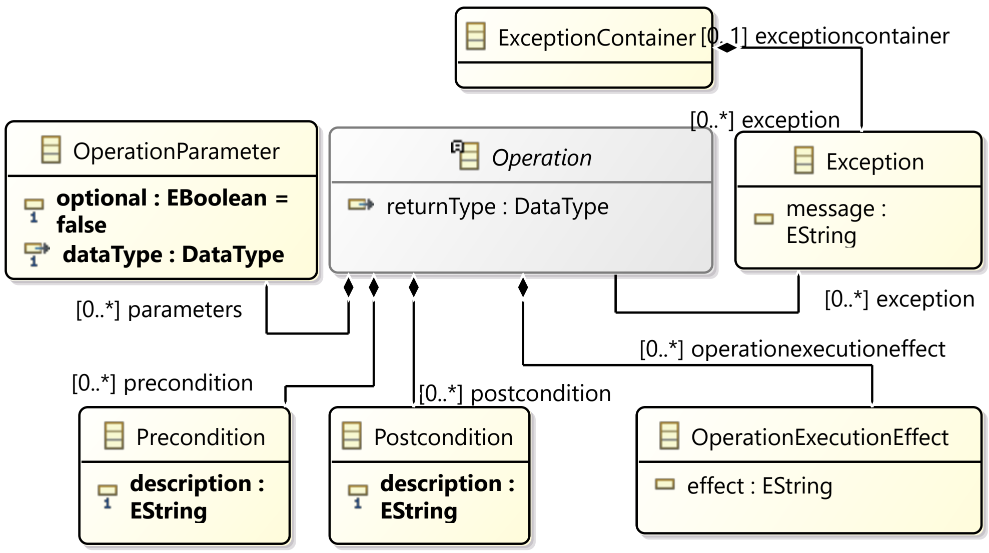


Figure 6.10: Classes and relations contained in the *OperationModel* package

operation in an independent development scenario. This specific definition is required because the operation can be used by a third party in an modular simulation. The specifics description of an operation allows guidelines for its designed application. Therefore, the operation has to defined what it expects from the callee and what can be expected of its execution. Conditions have to be defined to be met to execute the operation (precondition) for this purpose. The operation guarantees a particular state or reaction after the execution (postconditions) if these conditions are met. It is possible that the callee can or must provide additional information to execute the operation. The parameters of the operation provide the additional information. For example, in Palladio, the notification about the start of a software system could entail an identifier. Only this identifier enables Palladio to simulate the correct system. A parameter can also be specified to be optional. This optionality allows alternative executions of an operation when additional information is provided. It is also possible that a problem in the execution of the operation occurs. Such a problem can be in the prior example, that the software system does not exist in Palladio. Exceptions for problems can be stated to enable the callee to react in a defined way The explicit statement of these exceptions enables the callee to react to certain erroneous situations (e.g. to re-sent a correct id).

Operations are described in the metamodel of the DSL by the *Operation*[SE] class. The information to be specified in an operation are realised as containments in this class. Each *Operation* contains zero or more model elements of the *Preconditions* and *Postconditions* classes. The attribute *description:EString* textually describes a condition. Each *Operation* can model the effect of an execution by an instance of *OperationExecutionEffect*. This instance enables a developer to identify the operation with the desired effect. The *OperationExecutionEffect* can be used when the name of the operation is not descriptive enough. The attribute *effect:EString* enables the modeller to describe the effect. The provision of a textual description is necessary because the real effect is heavily implementation dependent. The DSL, however, is not concerned with implementation details. Each operation can have multiple parameters to represent additional required information. Each of those parameters contains an identifying name. To represent these parameters, the *Operation* class contains zero or more instances of the *OperationParameter*[SE] class. The *OperationParameter* class references one *DataType* instance to define the expected type of the parameter. By the definition of the type, the expected values are indirectly described for this *OperationParameter* as well. A *OperationParameter* can be described as optional to reduce the number of model instances. The attribute optional:EBoolean describes this optionality. Each operation can reference zero or more exceptions to be encountered when executed. These exceptions are represented by the *Exception*[SE] metamodel class. The exception instances themselves are created in a *ExceptionContainer*[SE] instance. The centralised definition and management of *Exception* instances in a *ExceptionContainer* element enables a collected reuse. It is possible to export the complete *ExceptionContainer* instance instead of creating or exporting every *Exception* anew in another model. Also, the collection of *Exception* instances and referencing enables centralised maintenance. If an *Exception* instance has to be changed (e.g. another description or another name for better recognition) it only has to be done in the container. If *Exception* instances for every operation would be created, the problem of replication could arise when one exception is used in multiple operations. When this problem occurs, each replication would have to be manually changed.

6.5.3 Representation of Information of Simulation Features

The information of simulations has to be represented in order to use them in a modular simulation. This representation is necessary for a defined structure. The structure enables a shared understanding of the relation between the information. The object-oriented structure is selected due to its application in the field of simulations. Object orientation can express entities in a simulation. The scheme is also applied by several high-level languages like Java, C++ and C#. Because of this design decision, the DSL describes provided and required information of a simulation in an object-oriented structure.

The *SimulationInformation* package of the DSL provides the ability to model required and available information of simulation features in an object oriented structure. Its contents are shown in Fig. 6.11 The entry-point to describe information of simulations is the

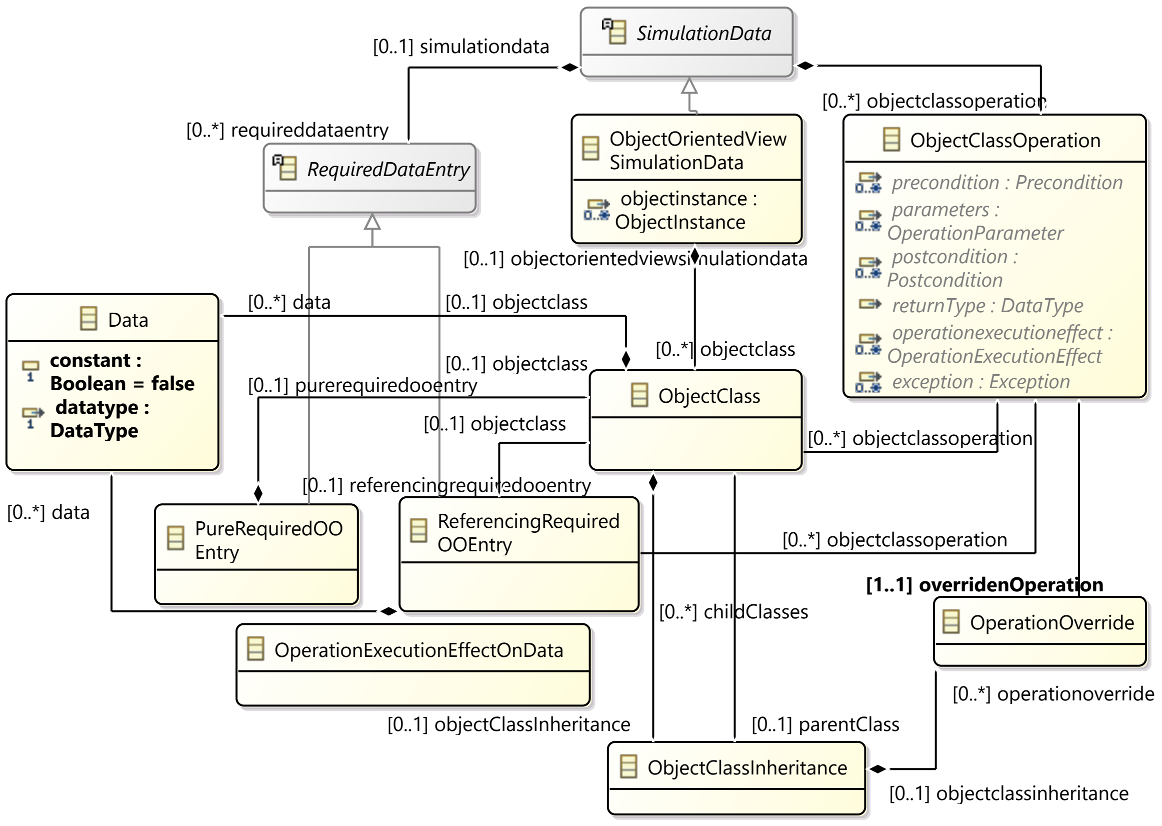

SimulationData
[0..1] simulationdata
[0..*] objectclassoperation
[0..*] requireddataentry
RequiredDataEntry
ObjectOrientedView SimulationData
objectinstance : ObjectInstance
ObjectClassOperation
precondition : Precondition
parameters : OperationParameter
postcondition : Postcondition
returnType : DataType
operationexecutioneffect : OperationExecutionEffect
exception : Exception
[0..1] objectorientedviewsimulationdata
Data
constant : Boolean = false
datatype : DataType
[0..*] data
[0..1] objectclass
[0..*] objectclass
[0..1] objectclass
[0..1] purerequiredooentry
ObjectClass
[0..1] objectclass
[0..*] objectclassoperation
[0..1] referencingrequiredooentry
[0..*] data
PureRequiredOO Entry
ReferencingRequired OOEntry
[0..*] objectclassoperation
[1..1] overridenOperation
[0..*] childClasses
OperationOverride
OperationExecutionEffectOnData
[0..*] operationoverride
[0..1] objectClassInheritance
[0..1] parentClass
ObjectClassInheritance
[0..1] objectclassinheritance

Figure 6.11: Classes and relations contained in the *SimulationInformation* package

abstract class *SimulationData*[SE]. The abstraction of this class enables the DSL to be enhanced by other approaches than the object-oriented structure. The current metamodel provides an object-oriented structure by the use of the *ObjectOrientedViewSimulationData* metamodel class. The definition of available and required information is essential to enable simulations to exchange data. It must be known which information a simulation can provide or requires. Also, the developer who creates a modular simulation must know of the information required to assemble the simulation correctly. This property is of importance because the modular simulation has to be designed to enable the fulfilment of these requirements. This fulfilment provides the means for the simulation to execute its model correctly.

Each simulation has to define its available information to enable collaboration with other simulations. This definition enables a developer to select the component to provided information for another simulation in a modular simulation. The term "available" includes all information a simulation is designed to be able to provide in its execution. The description of not only provided but all available information for one simulation scenario enables greater reuse of the model. For example, the model of Palladio is created with the DSL. Palladio can produce information about response time and reliability of a system. Palladio could be modelled only for the scenario of IntBIIS with the response time of the system. However, this modelling approach would reduce the use of the independent Palladio model only to modular simulations using response times. When the reliability is described as available, the model could also be used in other modular simulations.

In the DSL, the information of simulations is structured around an object class according to the object-oriented structure. Every object class contains properties and operations and represents the entities in a simulation. The properties specify the data contained in an entities. This data is used to define their state. An object class with its data is used over a long duration in the simulation time. All object classes with their states in a simulation constitute the state of the entire simulation. This availability over time is opposed to operations. Operations in an object class define the possibility to notify the simulation about an event or request a state change. The containment of an operation in an object class provides more specific information about the targeted entity in a simulation. The call of an operation can trigger events in a simulation using this entity. Nevertheless, the DSL allows also to define operations not confined in object classes. This definition enables a general purpose application of notifications. The *ObjectOrientedViewSimulationData* realizes a zero to many containment to the *ObjectClass*[SE] metaclass. Thus, zero or more instances can be created in a simulation. Each instance contains zero or more *Data*[SE] and *ObjectClassOperation* instances. The *Data* class describes data contained in an object class. Data in a simulation is represented to provide values for a certain purpose (e.g. to specify capacity in a device resource in IntBIIS). This capability is the reason why not only data types are referenced in an object class. The *Data* class allows a semantic definition of its data in the context of the *ObjectClass*. For this purpose, the *semantics* attribute of the *SemanticEntity* superclass is used. Data in the DSL is based on a data type. This basis enables the interpretation of data by a computer. The *Data* element references one instance of *DataType* to describe the data type of the data. An *ObjectClassOperation* inherits from the super-type *Operation* of the *OperationModel* package. An enhancement of the operations *OperationExecutionEffect* is provided by the sub-type of *OperationExecutionEffectOnData*. The definition of the specialized *ObjectClassOperation* and *OperationExecutionEffectOnData* enables a more precise description of the operation. This precise description entails the effect on the referenced *Data* when the operation is called (e.g. this operation decrements this data by one).

Particular attention has to be placed to information required by the simulation. A requirement specifies the need for specific information to correctly execute. Requirements describe the need for specific data (e.g. the response times of components in IntBIIS). Also, a requirement of an operation represents the handling of notifications (e.g. a component in Palladio needs to be called to start its traversal). Therefore, each simulation has to be able to define the required information to be able to execute its simulation. This definition is optional because it is possible that nothing is required. Two cases are differentiated for the description of the required information. In the first case, the simulation already partially describes an object class. For example, the simulation describes the troop movement direction and the size of the troop. The speed of the troop is required to execute the movement model. Therefore, a requirement to an existing object class exists. The required information has to be described in addition to the existing one. This specification allows another simulation to describe a troop entity which calculates the movement of the troop by its size. The Second case is the definition of information not related to any *ObjectClass* in the simulation. For example, the business process of IntBIIS only needs the response time of a system from Palladio. No information of software system is represented in the business process model. The abstract class *RequiredDataEntry*[SE] provides the capability to describe the required information in the DSL. Zero or more instances of sub-types of this abstract type can be created in an *ObjectOrientedViewSimulationData* instance. The *PureRequiredOOEntry* model element realizes the requirement if no fitting *ObjectClass* exist (case two). Its metaclass incorporates a containment to create one *ObjectClass* for this purpose. Thus, a new *ObjectClass* is created. The *ReferencingRequiredOOEntry* realizes the requirement of information related to an existing *ObjectClass* (case one). Therefore this entry references the existing *ObjectClass* instance. The required information (i.e. data and operations) are modelled as containments of the *ReferencingRequiredOOEntry*. Another approach to describe available and required information would be to annotate every class by an enum. This enum then consists of the literals "available", "required" or "both". However, for usage in the assembly, the correctness had to be checked by the editor. The checking in the editor should be minimised in the DSL. Thus, the prior described structure is chosen.

6.5.4 Discussion of the Metamodels' Object Oriented Structure Design

Each usable information is attached to an (object) class in an object-oriented structure. This class contains properties (i.e. the data usable) and operations (i.e. functions callable to change information or state). It is possible to represent this scheme by different metamodel design approaches in the DSL.

One approach is the one currently used. A metamodel element for each type of information is created for this approach. All elements containing potential information are understood as "type of information". Thus, in the DSL, these types are object classes and their properties. Also, information types are operations with their parameters. The root element to describe information is the object class. This element contains properties (i.e. the data) and operation elements. For each information type, a separate metamodel class is used. This design allows a well defined static structure of the object-oriented approach to describe information. Classes in the metamodel can explicitly reference a particular information type by their class. Collected reuse of the model elements is possible due to the direct containment of the information (i.e. data and operations) in the object class. The static, well-defined structure, however, is also a drawback of this approach when object classes, properties and operation have to be encapsulated in the DSL. In this case, an additional element for each information type has to provided. Also, the structure between these elements has to be replicated. This approach currently adds at least four elements (for object class, property, operation, operation parameter) for each object-oriented structure representation in the metamodel. The approach also has a negative impact on maintainability. This negative impact is of importance when the object-oriented structure has to be changed or extended (e.g. by another information type). This change has to be realised for each object-oriented structure in the metamodel.

An approach to remedy these drawbacks is the definition of a loosely coupled description. Here, one "simulation information" metamodel element is used. An enum contains literals for each information type (i.e. object class, property, operation or parameter). The "simulation information" element contains a literal of this information type enum. Instances of related "simulation information" are required to be together in a "simulation information" container. This container ensures that the contained instances are only used together. An object-oriented structure is realised by an "is-contained-in" or "contains" relation. Instances of a corresponding metamodel element specify the containment in a "simulation information" metamodel element. The "is-contained-in" instance references one "simulation information" instance. This approach would provide greater flexibility and reuse of all instances. Also, it reduces the metamodel elements for representation of the object-oriented structure. Additionally, the enhancement of possible types of information is easily done by the addition of another literal to the "simulation information kind" enum. Nevertheless, this approach has several drawbacks. One of the drawbacks is possible undesired "is-contained-in" relations. For example, an operation can be contained in a property or a parameter in an object class. An editor has to be created to enforce the object-oriented structure. Thus, the editor has to prevent invalid relations. This approach would defer the correctness to the editor. The DSL is designed with the goal to reduce the outsourcing of checks to an editor. Because in each creation of an editor for the DSL, the correctness of checks must be ensured. Thus, besides all drawbacks of the strict object-oriented structure approach, it is used in the DSL. Additionally, in the strict object-oriented structure, object classes, properties, operations and parameters can inherit or references its defining information. This capability is not possible in the flexible approach. Here, common possible super classes would have to be designed to reference the desired capabilities. Such an approach would also increase the number of checks that have to be made by an editor. Both structures are shown in Fig. 6.12

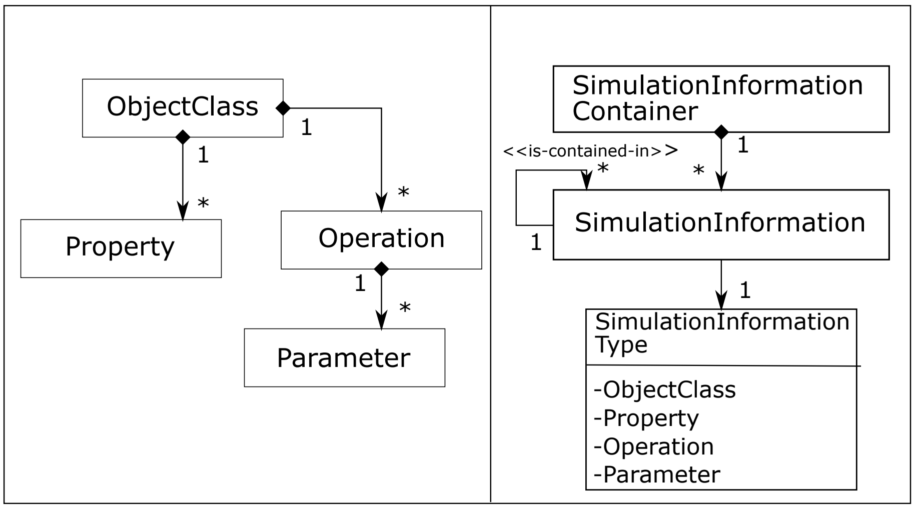


Figure 6.12: Two possible designs of object oriented structures in the DSL. Left: A strict defined structure (realised in the DSL). Right: A loosely coupled approach to enable a object oriented structure. Arrows with black rectangles are containments. Arrows alone are references

6.5.5 Object Instances

Object are instances of object classes. Object classes provide the information contained and the object (instance) is an entity in the simulation. Therefore, each object (instance) defines the state of a simulation. The *ObjectInstance* package provides capabilities to describe object instances. To assign values to the information contained in *ObjectClass* elements, the *metamodel* uses key-value-maps as elements. Each key references a *Data* instance. To provide a more typed capabilities, the *Data* class would have to be sub-typed (e.g. for explicit handing of the *EnumType*). This capability currently is only provided by the *DataSpecification* package in a limited way. The DSL is designed to describe the static architectural and information focused aspects of a modular simulation. Objects are created and deleted in the execution of simulations. This creation and deletion of objects is determined in the execution of the simulation. Thus, objects are related to the dynamic aspects of modular simulations. Therefore, the DSL currently does not use the contents of the *ObjectInstance* and *DataSpecification* package.

6.6 Description of Simulation Features and Modular Simulation Environments

The metamodel elements described in Sec. 6.5 can be used to define the information contained in simulation features and MSEs. This section describes the modelling approach of the DSL to describe simulation features and MSEs

6.6.1 Modelling of Simulation Features

Modular simulations consist of multiple simulation modules. One type contained in a modular simulation is the simulation feature. Simulation features contain only one simulation. However, it does not matter if this simulation consists of one single model or a monolithic model (i.e. multiple models confined in one). In the DSL, one goal is to describe simulation features independently. Therefore, it is mandatory to provide metamodel elements that are not dependent on instances of other simulation feature, modular simulation or MSE models. The elements to describe simulation features with the DSL are available in the *SimulationFeature* package. Its content is depicted in Fig. 6.13 The description of simulations is mainly focused on the data they require and can make available. To explicitly state the required information is essential. Required information is understood to be necessary for a (correct) execution of the simulation. One goal of the independent description of simulations is to be usable in different modular simulations and by different developers. The description of required and available information can be not enough to adequately describe the semantics of the simulations. The semantics of a simulation, however, are essential for their use in modular simulation. The semantic definition of an simulation enables other developers to acquire information about the simulation. Therefore, they can decide if the simulation provides the desired information for their modular simulation. For this purpose, a further semantic definition of the simulation has to be provided. The *SimulationFeature*[SE] class forms the entry-point to model a simulation

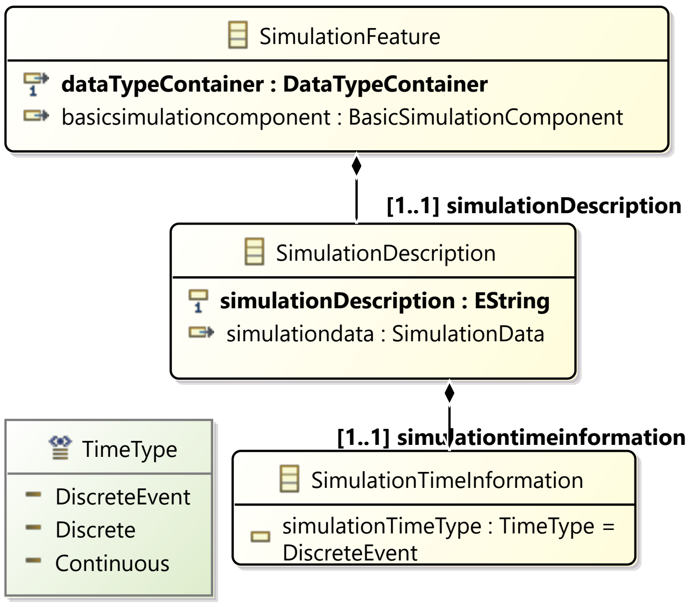

SimulationFeature
dataTypeContainer : DataTypeContainer
basicsimulationcomponent : BasicSimulationComponent
[1..1] simulationDescription
SimulationDescription
simulationDescription : EString
simulationdata : SimulationData
[1..1] simulationtimeinformation
TimeType
DiscreteEvent
Discrete
Continuous
SimulationTimeInformation
simulationTimeType : TimeType = DiscreteEvent

Figure 6.13: Classes and relations contained in the *SimulationFeature* package

feature with the DSL. Code can be generated for each modelled simulation to be used in an implementation. If this ability of a model is used, the implementation of the described simulation has to be contained in a *SimulationFeature* instance. The data types used in the simulation have to be individually specified to enable its independent definition. The definition of data types is used to represent the information contained in simulations to be understandable by computers. For this purpose the *SimulationFeature* class defines a instance of *DataTypeContainer*. The possible data types used by simulations are defined in Sec. 6.5.1. For a reusable description of the simulation, the *SimulationDescription* class is available. A definition of the time-scheme used in a simulation (e.g. discrete-event simulation) supports the understanding of the simulation by other developers. If a MSE is specialised on capabilities of DES it is possible to find incompatibilities of the simulation with a continuous time scheme. This problem can be relaxed by the experimental time adaptation, described in Sec. 6.2.1. To express the simulations time-scheme used in the simulation, the *SimulationDescription* contains a containment of the rudimentary *SimulationTimeInformation* class. In this class, the time-scheme is represented as an attribute of the *TimeType* enum. Currently the enum includes the literals *DiscreteEvent*, *DescreteTime* and *Continuous*. The *SimulationTimeInformation* is currently not further used in the DSL. Nevertheless, it provides an entry-point for enhancement of the DSL in regards to the time-scheme (e.g. time-adaptation discussed in Sec. 6.2.1.

The *SimulationDescription* includes the attribute *simulationDescription:EString* to enable a textual description of the simulation. This attribute provides a human-readable description of the simulation. Simulations are used in a modular simulation to provide information and behaviour in cooperation with other simulations. Therefore, the available information of the simulation has to be provided to be usable. A containment of *SimulationData* in the *SimulationDescription* provides capabilities to model the required and available information in the simulation. Developers can choose the models for their modular simulation on the basis of the textual description and the available and required information. The most important content of a simulation feature description in the DSL is currently the specification of required and available information. This aspect emphasises the current information-driven focus of the DSL. This relation can be seen in Fig. 6.13. Two of three classes in the package contain references to elements of *DataRepresentation* package.

6.6.2 Modelling of Modular Simulation Environment

A MSE provides the capabilities to enable interaction between simulations. Every simulation interacting through a MSE has to adhere to its defined scheme of interaction. The MSE also has to include capabilities for managing aspects of simulations to provide a common understanding of the created modular simulation. An example of this is the management of a common understanding of time. Here, every simulation must adhere to or at least know of a common time line If no common time-line is provided, simulations cannot update or get values conforming in a correct order. This situation is created because this order has to be according to the simulation time the information sent or changed. However, without a common time line, each simulation would interpret the information according to its time line This action could result in invalid states. Therefore, the management of some simulation aspects is necessary to create a successful interaction between simulations. As mentioned in Sec. 6.1, a centralised interaction-scheme is used in the DSL. The decision originates mainly from the popularity of HLA. To be able to evaluate the approaches of this thesis, an interaction approach (i.e. a MSE) was required. The most commonly found and easy to access MSEs are HLA implementations. The information of simulation is passed over a common coordinating unit, the coordinator, to facilitate communication between simulations. Besides the MSE capabilities, the coordinator needs knowledge about the information required and provided by the simulations. This information consists of data and operations of other simulations. The knowledge about this information enables the controlled modification of simulation information in the modular simulation. Also, it enables the coordinator to pass the updated information correctly to the simulations. The time in a simulation is synchronised with those of the other simulations. A MSE is described by the capabilities it provides. The capabilities related to one managed aspect are defined in a service. In the DSL these services are called "management services". The relation between one capability of a MSE and one management service enables its reuse in another MSE. With this scheme, the single aspects can be reused by only extracting and using the management service. Also, capabilities can be centrally maintained and are easier found due to their collected definition. The MSE can define required context information to handle the data and operations of the simulation. The data and operations can be processed by the coordinator only if they specify the context information of the MSE. Context information can be, for example, the scheduling-strategy for updating the value of data. The explicit representation of such context information in the model is necessary. The DSL is designed to allow different MSEs for the use in a modular simulation. This design implies that the context information cannot be defined for simulations in their independent description. A specification of the context information in the independent description of simulations would violate the goal of the independent design of MSEs and simulation features of the DSL. The decision to assume the fixed set of context information of HLA was declined because of two reasons. A designed MSE in the DSL could also contain only a subset of the functionality of HLA. Therefore, not all information would be needed. For example, the MSE uses a scheme, where simulations directly have to enforce the information exchange. Here, only the simulation time is coordinated by the MSE. In this approach, the context information for the distribution of information is not usable. Also, the use of a fixed set of context information would prohibit developers to develop another approach besides HLA. The DSL however, is designed to provide the ability to develop a broad range of modular simulation approaches. For example the approach of Scerri et al. [60] uses agent-based simulation and an approach similar to HLA. The capabilities are extended to allow shared ownership of information. Such approaches could come with own required context information. The flexible definition of the context information in the DSL also allows the description of such approaches.

The *MSE_Entity*[SE] class is the entry-point in the *ModularSimulationEnvironment* package. This class is also used as extension point to provide different kinds of MSE. Concrete elements to model a MSE have to inherit this class. This inheritance enables an easier enhancement of the DSLs capability to model a MSE. The metamodel components and their relation of the *ModularEnvironment* package is shown in Fig. 6.14 The *Coordinator*

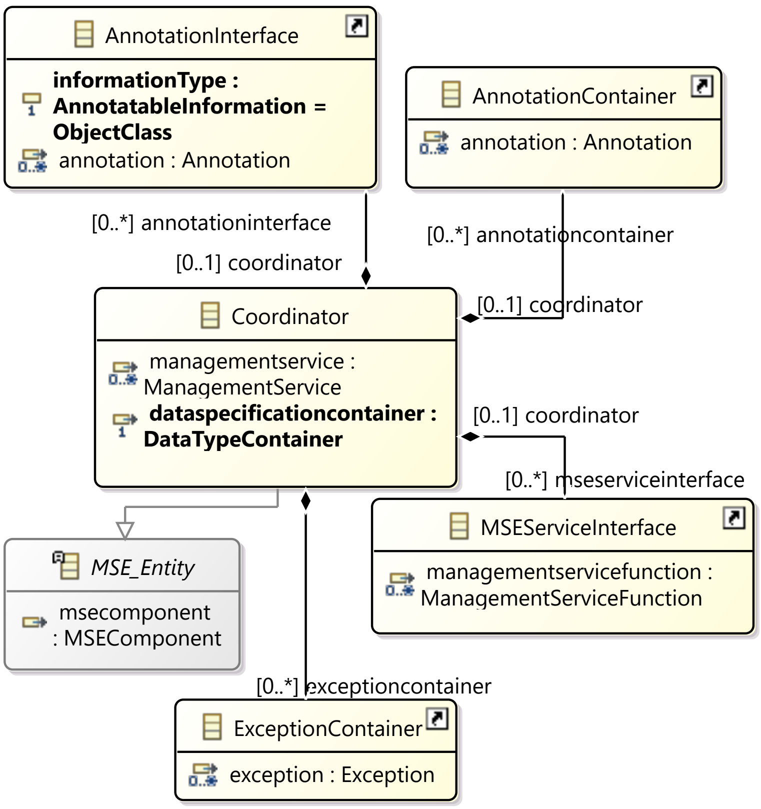
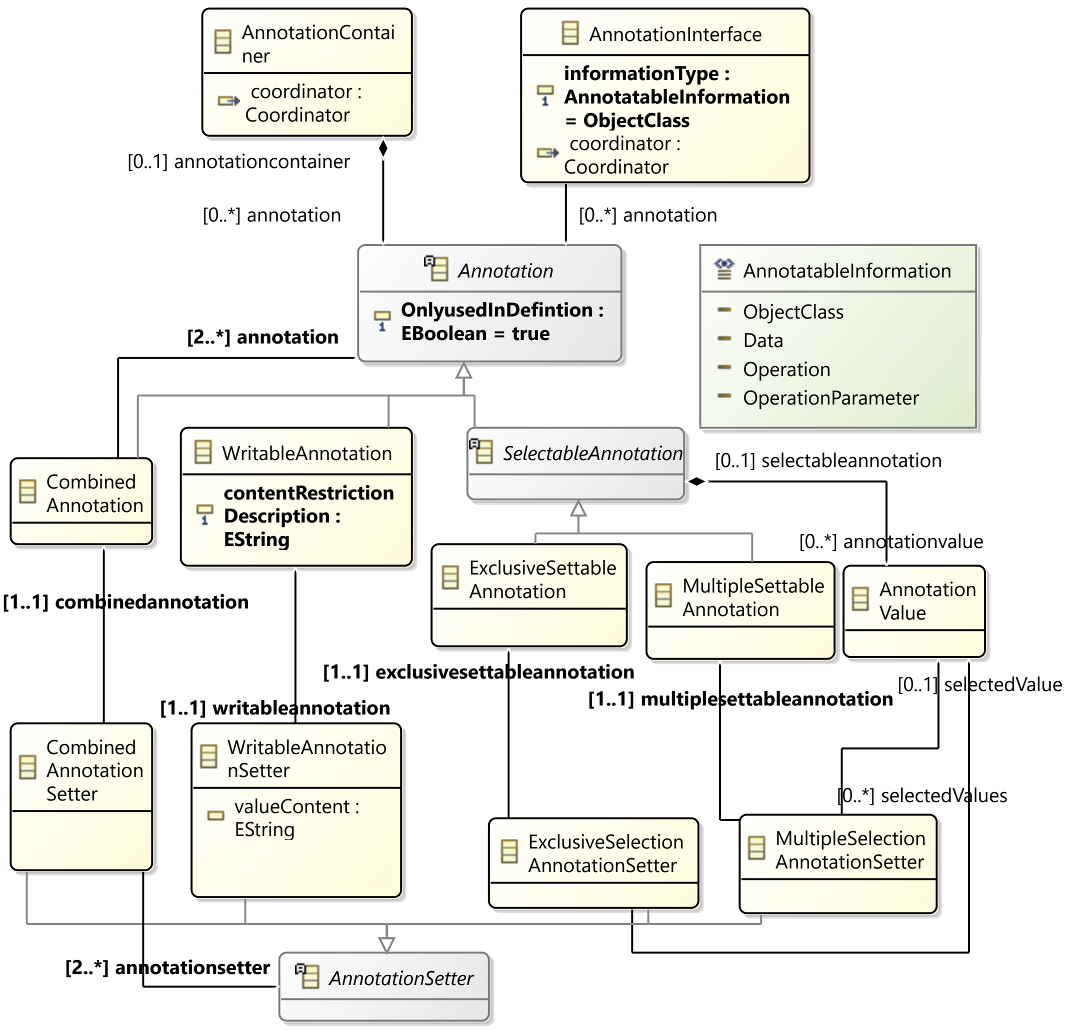

AnnotationInterface
informationType : AnnotatableInformation = ObjectClass
annotation : Annotation
AnnotationContainer
annotation : Annotation
[0..*] annotationinterface
[0..1] coordinator
[0..*] annotationcontainer
[0..1] coordinator
Coordinator
managementservice : ManagementService
dataspecificationcontainer : DataTypeContainer
[0..1] coordinator
[0..*] mseserviceinterface
MSEServiceInterface
managementservicefunction : ManagementServiceFunction
MSE_Entity
msecomponent : MSEComponent
[0..*] exceptioncontainer
ExceptionContainer
exception : Exception

Figure 6.14: Classes and relations contained in the *ModularEnvironment* package

metamodel class is used to define a MSE with centralised capabilities. The *Coordinator* class itself is mainly used to provide a collected definition of the MSE capabilities. The following subsections describe the content of the MSE and the *Coordinator* class.

6.6.2.1 Definition of Context Information

The context information required for processing information provided by simulations is called *annotation* in the DSL. The *Coordinator* contains zero or more *AnnotationContainer*[SE] instances. Each container contains instances of the *Annotation*[SE] class. Both metamodel elements are located in the *Annotations* package. Its content is depicted in Fig. 6.15. The containment in the *AnnotationContainer*[SE] instances facilitates central



AnnotationContainer
coordinator : Coordinator
AnnotationInterface
informationType : AnnotatableInformation = ObjectClass
coordinator : Coordinator
[0..1] annotationcontainer
[0..*] annotation
[0..*] annotation
Annotation
OnlyusedInDefintion : EBoolean = true
AnnotatableInformation
ObjectClass
Data
Operation
OperationParameter
[2..*] annotation
Combined Annotation
WritableAnnotation
contentRestriction Description : EString
SelectableAnnotation
[0..1] selectableannotation
[0..*] annotationvalue
ExclusiveSettable Annotation
MultipleSettable Annotation
Annotation Value
[1..1] combinedannotation
[1..1] exclusivesettableannotation
[1..1] multiplesettableannotation
[0..1] selectedValue
[1..1] writableannotation
Combined Annotation Setter
WritableAnnotationSetter
valueContent : EString
[0..*] selectedValues
ExclusiveSelection AnnotationSetter
MultipleSelection AnnotationSetter
[2..*] annotationsetter
AnnotationSetter

Figure 6.15: Classes and relations contained in the *Annotations* package

management of multiple instances of the *Annotation* class. Also, it enables reuse and provides better maintainability. These benefits are provided because there is only a centralised location for the definition of *Annotation* instances.

It is possible to define different sets of *Annotation* instances. These sets specify the context information to be provided by information types. The separation between information types allows a definition of the meaning of context information in the MSE. For example, an object class itself cannot be updated in any way in the scheme of the current provided DSLs information structure. The update of an object class is realised by updating the attributes. Therefore, there is no meaning in defining an update-policy for the object classes itself. The sets of *Annotation* instances for information types are defined by instances of the *AnnotationInterface*[SE] class. Each instance contains the *informationType* attribute. The values of *informationType* are literals of the *AnnotatableInformationTypes* enum. The current literals of the enum are "ObjectClass", "Data", "Operation" and "OperationParameter". This approach enables fewer metamodel classes than the creation of a metamodel class for each information type. In an enhancement of the DSL, information types can be added only by a new literal in the *AnnotatableInformationTypes* enum. This design contradicts the decision made for the object-oriented structures discussed in Sec. 6.5.4. However, the annotation information can only be used with the same model elements. Contrary to this approach, the structure of object orientation can be used in different locations of the metamodel. Therefore structural integrity has to be checked more often on different elements than that of the *AnnotationInterface*. The *AnnotationInterface* references one or more *Annotation* instances confined in a *AnnotationContainer*. These references realise the assertion of *Annotation* instances each information type has to define.

As stated, *Annotation* instances are used to define context information required to process information. In a modular simulation, the required information of a simulation only uses the information another simulation provides. In the DSL the providing simulation defines the available information in the modular simulation. The requiring simulation only uses the information. Some context information has only to be provided in the definition of information. An example is the strategy to update the values of properties of an object class when they are changed. On the other hand, some context information can be used in the request of the information as well. The definition of an information may state if a certain context information value is allowed. For example, if other simulations can own the defined information. The requiring simulation itself can state with the same context values if this capability is assumed or not. For example, the requiring simulation specifies a potential request to own the information. To model this capability of an *Annotation*, the attribute *OnlyUsedInDefinition:EBoolean* is provided. The value "true" models if the definition of this *Annotation* is allowed by definitions of requirements. Therefore, when the attribute is set to true for a *Annotation* instance, it can be ignored when modelling required information.

Each sub-type of *Annotation*, defines the kind of value an *Annotation* can assume. To assign a the specific value, the abstract class *AnnotationSetter* with its specialised sub-class is available. For each concrete sub-class of *Annotation*, a sub-class of *AnnotationSetter* exists. This structure is chosen due to the different values and cardinalities of selectable values a subtype of *Annotation* represents. A more generic structure would limit the possibilities to describe and later set values. A way to counter this approach would be to provide an editor containing the logic to express the current structure. This, however, is not desired in the conception of the DSL. The subclasses of *Annotation* and their capabilities are:

- The *WritableAnnotation* provides the capability to provide a free-text-writeable *Annotation*. The content of the text field is restricted by a textually description in the *contentRestrictionDescription:EString* attribute. The *WritableAnnotationSetter* references one *WritableAnnotation* and specifies the content by the attribute *valueContent:EString*.

- The abstract *SelectableAnnotations* provides an element to define *Annotations* with values selectable by the corresponding *AnnotationSetter*. These values are defined by one or more contained *AnnotationValues*[SE]. The value is represented by the *name* attribute. The classes *MultipleSettableAnnotation* and *ExclusiveSettableAnnotation* sub-type *SelectableAnnotations*. These sub-types differentiate between different cardinalities of *annotationValue* the corresponding *AnnotationSetter* sub-classes can reference. For the *MultipleSettableAddtion*, the corresponding *MultipleSelectionAnnotationSetter* references one *MultipleSettableAnnotation* and one or more of its defined *AnnotationValue*. Parallel to this, the *ExclusiveSelectionAnnotationSetter* references one *ExclusiveSettableAnnotation* and zero or one *AnnotationValues*. The separate definition of these metamodel elements allows to explicitly model the cardinalities the setters can set for each *SelectableAnnotation* subclass.

- The *CombinedAnnotation* subtype provides the ability to contextually describe a connection between two or more *Annotation* instances by a referencing relation. Parallel to this approach, the *CombinedAnnotationSetter* contains two or more references to *AnnotationSetter*. This allows to define connected annotations. A connection is necessary if one annotation is related to another. For example if one annotation provides a "condition" value, a *WritableAnnotation* can be used to specify the condition. The *CombinedAnnotation* allows a specification of this relation.

6.6.2.2 Representation of MSE Capabilities by ManagementServices

The functionality and internal processes of a coordinator itself are contained in a *ManagementService*[SE] class. For example, the capability to manage and coordinate simulation time is confined in a *ManagementService*. This approach enables a reusable design of the *ManagementService*. Each *ManagementService* can be extracted of the MSE representation and reused in another to provide certain capabilities. Also, it allows the more concrete definition of the capability represented by a *ManagementService* itself. A *ManagementService* defines notifications to provide access and interaction between simulations and the MSE. As stated in Sec. 6.5.2, these notifications are called operation in the DSL. Each management service therefore defines operations responsible for realising the access to represented capabilities. The entry-point for modelling of a management service is the *ManagementService* class in the *ManagementServices* package. Its contents is shown in Fig. 6.16. As it can be seen in this figure, a *ManagementService* is mainly defined by its purpose and the provided operations. This way of modelling results due to the strong implementation dependency of *ManagementServices*. The implementation of functionality of the MSE is done in each *ManagementService*.

Each *ManagementService* contains zero or more instances of *ManagementServiceSupportEntitiy*. The attribute *purpose:EString* of the MSE describes the supporting entities functionality. No further modelling is possible, due to their strong implementation specific context. However, the knowledge of such entities is important when another developer

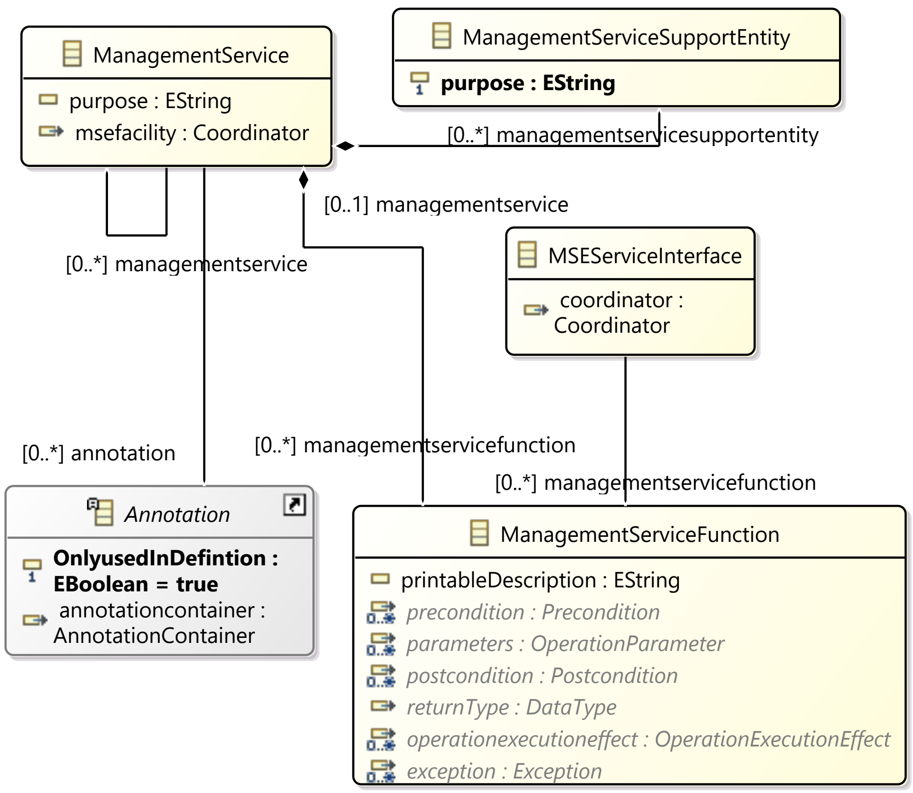

ManagementService
purpose : EString
msefacility : Coordinator
ManagementServiceSupportEntity
purpose : EString
[0..*] managementservicesupportentity
[0..1] managementservice
[0..*] managementservice
MSEServiceInterface
coordinator : Coordinator
[0..*] annotation
[0..*] managementservicefunction
[0..*] managementservicefunction
Annotation
OnlyusedInDefintion : EBoolean = true
annotationcontainer : AnnotationContainer
ManagementServiceFunction
printableDescription : EString
precondition : Precondition
parameters : OperationParameter
postcondition : Postcondition
returnType : DataType
operationexecutioneffect : OperationExecutionEffect
exception : Exception

Figure 6.16: Classes and relations contained in the *ManagementService* package

reuses the *ManagementService* model. This developer can be a third-party developer not entirely familiar with the implementation details of the *ManagementService*. The modeller can provide guidelines for the needed implementation content by the *ManagementService-SupportEntitiy*. A *ManagementService* defines its available functionality by the containment of zero or more *ManagementServiceFunction* instances. They enhance the *Operation* class and also supply the *printableDescription:EString* attribute. The modeller can describe the purpose of the functionality of each operation instance in more detail with this attribute. Each *ManagementServiceFunction* can be seen in the implementation as one function in the code. Multiple management services can interact to provide a certain capability. This relation can be modelled by the self-reference contained in the *ManagementService* class. A *ManagementService* needs certain context information of information to provide their capabilities correctly. For example, the context information specifying how to update data in relation to simulation time is needed in the management service. This context can be related to the common simulation time of the time management service. The *ManagementService* class contains a zero or more reference relation to *Annotation* to signal their requirement. The referenced *Annotation* instances define what context information is required in the case of the reuse of the *ManagementService*.

6.6.2.3 Interfaces for Interaction with the MSE

The *ManagementService* instances of a MSE model define the capabilities a MSE possesses. Also, the operations to use these capabilities are defined. In the DSL, the provision of capabilities is seen only as definitions of operation. This design enables the reuse and exchange of different *ManagementService* instances equal operations. The operations are collected in zero or more *MSEServiceInterface* instances to declare them to be usable in the interaction between simulations and MSEs. These instances reference zero or more *ManagementServiceFunction* instances. With the definition of multiple *MSEServiceInterface* instances the operations to represent a capability of one *ManagementService* can be collected. Thus, the *ManagementService* and *MSEServiceInterface* can be reused together. Additionally, the defined functionality can be used for different purposes. A purpose can entail operations to be called by simulations to interact with the coordinator (e.g. requesting a certain capability). Also, interfaces can be defined to be realised by simulations to be called by the coordinator. This realisation enables the coordinator to send updates to the simulations. This avoids the need for simulations to "poll" functionalities of the coordinator for updates. *Exception* instances are used to define information of possible errors that could happen in the execution of the capabilities of a MSE. The callee of the operation can then directly react to an exception. Instances of *ExceptionContainer*[SE] can be created to describe possible exceptions in the MSE. In an *ExceptionContainer*, multiple *Exception* instances can be defined. These exceptions can be referenced by *Operation* instances. The centralisation of exceptions enables better maintainability of the exceptions themselves. If each operation would define its contained exceptions, the same instance could not be reused. Thus, when for example the exception name changes, each instance would have to be changed manually. With the centralised approach, the *Exception* instance in the *ExceptionContainer* has to be changed.

6.7 Metamodel of the Adaptation Approach

The necessity and conceptual realisation of the adaptation approach is described in Sec. 6.2. This section describes the realisation of the adaptation approach in the DSL. The decisions behind certain aspects are also discussed. The DSL incorporates two concepts to achieve an independent description of the adaptation approach. One concept are abstract adaptation description describing what information to adapt and how to adapt it. The other concept are the adapter services executing the adaptation by the use of the adaptation descriptions. These two parts are further described in this section.

6.7.1 Adapter Services

One concept of the adaptation approach in the DSL is the executing unit of adaptations - the adapter service. The description of this unit is realized in the DSL by the abstract *AdapterService*[E] class. The instances of *AdapterService* represent the executing entities of the adaptation approach. This class is located in the *AdapterServices* package in the *Adaptation* package. Its content is displayed in Fig. 6.17 *AdapterService* instances are located in the components containing the simulations or coordinators of the modular

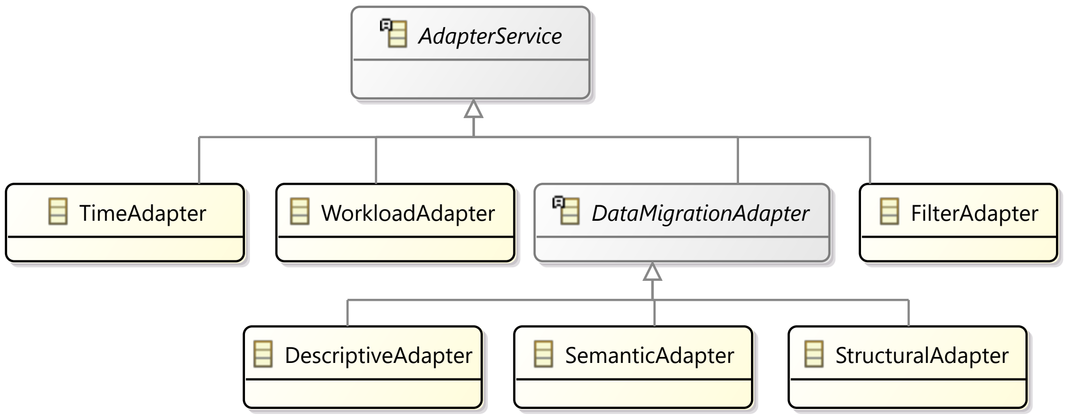


Figure 6.17: Classes and relations contained in the *AdapterServices* package

simulation. Not placing of the adapter services in the independent models of simulation features or MSEs allows minimising the dependencies between the independent designable models and the modular simulation. Another possible approach is the definition of the *AdapterService* as a separate tool in the modular simulation. Thus, it has to require all data provided by all simulations. This approach is applied by Neema [64]. The intention of the tool approach is to avoid modification of the used RTI. Its application to the DSL requires knowledge of the modeller about all simulations used in the assembly. This prerequisite, however, is contradictory to the notion of independent development of simulations and tools. However, an extension of the DSL is possible to provide individual description structures for tools.

The DSL provides different sub-types of the *AdapterService* metamodel element. These sub-types are used to express different topics of adaptation like declarative and structural adaptations as mentioned in Sec. 6.2.1. For each type, a corresponding metamodel subclass of *AdapterService*. The available classes are *FilterAdapter*, *TimeAdapter*, *DescriptiveAdapter*, *SemanticAdapter* and *StructuralAdapter*. Currently, there is no difference in the metamodel elements of the *AdapterService* sub-types. Nevertheless these elements are provided to provide further extension points for the adaptation approach in the DSL.

6.7.2 Adaptation Descriptions

The other concept of the adaptation approach is the description to identify the information to adapt. Additionally, it must be specified how the *AdapterService* has to adapt the information. An *AdaptationDefinitionRepository*[SE] instance contains zero or more instances of *AdaptationDescription*. The *AdaptationDefinitionRepository* class is directly contained in the *Adaptation* package. Its content is depicted in Fig. 6.18. The collection of related *AdaptationDescription* instances in a *AdaptationDefinitionRepository* facilitates reuse of related adaptation descriptions. The *AdaptationDefinitionRepository* can be reused as one model element if the descriptions are required in another modular simulation assembly. For example, it can be expected, that the adaptation of SI-Units is required in more than one simulation. The description of adaptations for SI-Units has only be modelled once in a

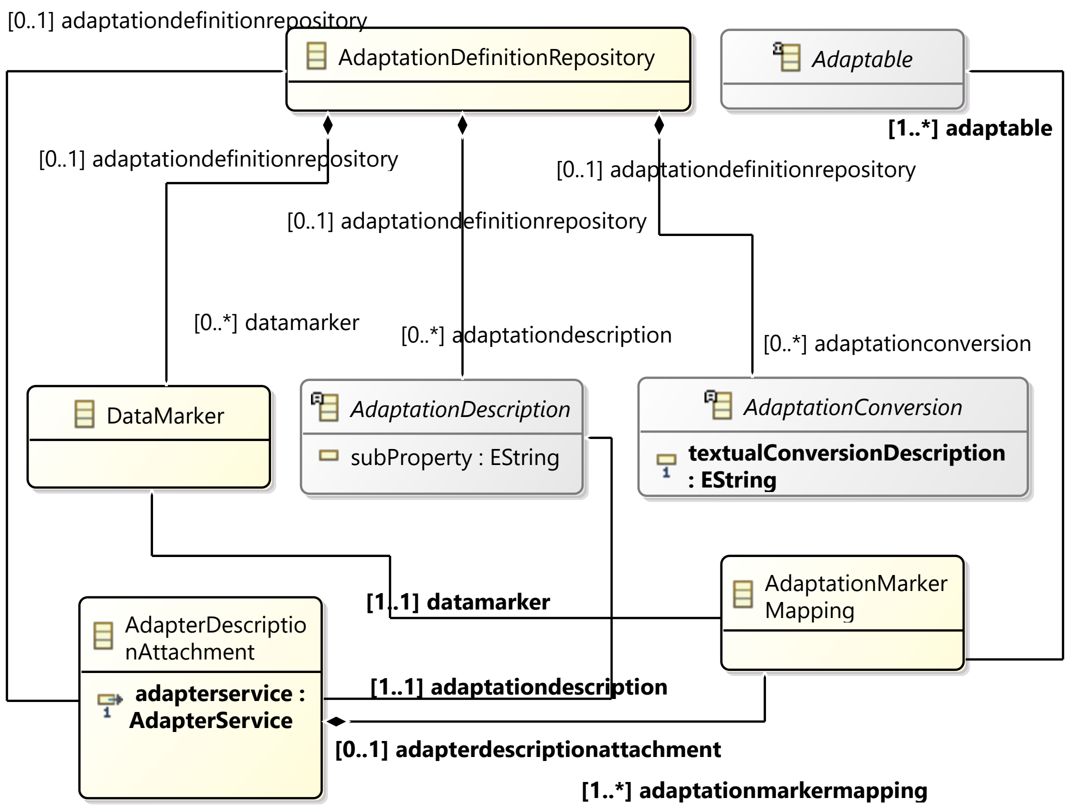

[0..1] adaptationdefinitionrepository
AdaptationDefinitionRepository
Adaptable
[1..*] adaptable
[0..1] adaptationdefinitionrepository
[0..1] adaptationdefinitionrepository
[0..1] adaptationdefinitionrepository
[0..*] datamarker
[0..*] adaptationdescription
[0..*] adaptationconversion
DataMarker
AdaptationDescription
subProperty : EString
AdaptationConversion
textualConversionDescription : EString
AdaptationMarker Mapping
[1..1] datamarker
AdapterDescriptionAttachment
adapterservice : AdapterService
[1..1] adaptationdescription
[0..1] adapterdescriptionattachment
[1..*] adaptationmarkermapping

Figure 6.18: Classes and relations directly contained in the *Adaptation* package

AdaptationDefinitionRepository. This repository can then be migrated to models of other modular simulations

To enable the independent description of adaptations, the abstract class *DataMarker*[SE] is used. The *DataMarker* abstractly defines information to be adapted when encountered. These markers are referenced in *AdaptationDescription* instances. The *DataMarker* instances abstractly represent the information defined in a description. Zero or more instances of *DataMarker* are contained in an *AdaptationDefinitionRepository*. This enables the reuse of *DataMarker* instances in multiple *AdaptationDescriptions* without modelling them multiple times. Also better maintainability of *DataMarker* instances can be expected. If *AdaptationDescription* instances would define own *DataMarker*, it is possible that multiple instances would relate to the same information. If this information is renamed, all related *AdaptationDescription* instances have to be scanned. In a centralised organisation, only the single *DataMarker* in the *AdaptationDefinitionRepository* has to be changed.

The DSL provides the abstract class *AdaptationConversion*[SE] to enable the description of the execution of an adaptation. This description is used when data corresponding to a *DataMarker* instance of an *AdaptationDescription* is found. The elements for this purpose are contained in the *AdaptationConversion* package. Its content is shown in Fig. 6.19. Instances of this class are contained in *AdaptationDefinitionRepository* instances. This enables reuse in multiple *AdaptationDescription* instances of the same repository. This approach reduces the number of *AdaptationConversion* instances because there can be multiple *AdaptationDescription* models using the same adaptation process. Also, these

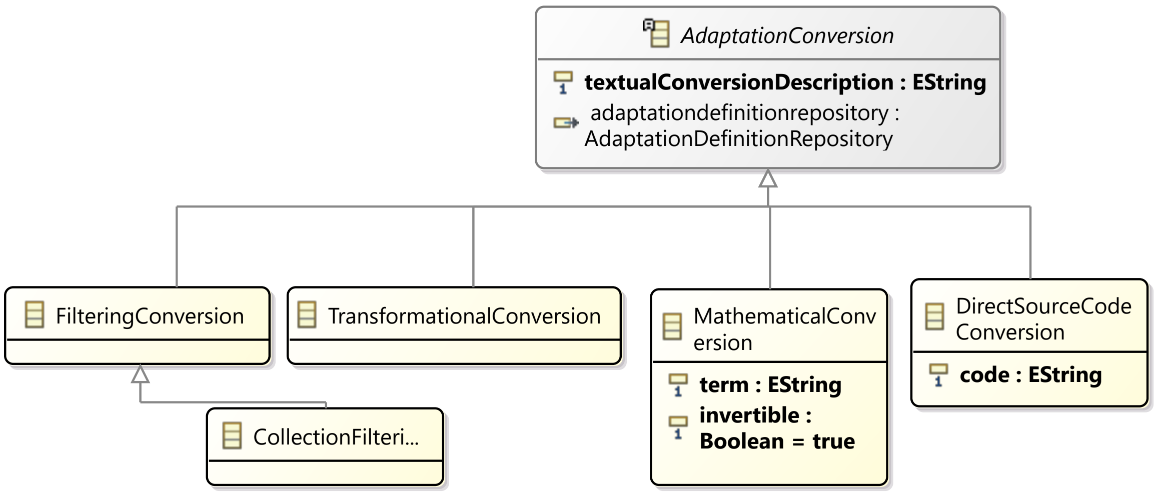

AdaptationConversion
textualConversionDescription : EString
adaptationdefinitionrepository : AdaptationDefinitionRepository
FilteringConversion
TransformationalConversion
MathematicalConversion
term : EString
invertible : Boolean = true
DirectSourceCode Conversion
code : EString
CollectionFilteri...

Figure 6.19: Classes and relations contained in the *AdaptationConversion* package

conversions can be reused due to their containment in the *AdaptationDefinitionRepository* in other models. For example, the *AdaptationConversion* describing the relabelling of a name can be reused in descriptions to translate languages and unit names. This approach also enables the reuse of implementation code in the *AdaptationConversion*. The one-time definition also enhances maintainability like with *DataMarker*. The *textualConversionDescription:EString* describes textually what the conversion is supposed to do. In general, no better model of the adaptation process can be given in the DSL. This problem exists because the execution of an adaptation is heavily implementation-dependent.

The sub-classes of *AdaptationConversion* differentiate between several kinds of adaptation processes. This design provides a guideline for different kinds of conversions. Each sub-class provides additional information required for the kind of adaptation process. The *MatematicalConversion* specifies how to convert values by the definition of a term. The term is defined as the attribute *term:EString*. The modelling of mathematical terms is not deemed of priority in the current DSL. Because of this, the term is only a textual description of the formulas (e.g."+5"). The *invertible:Boolean* attribute signals if the term can be inverted to convert the target value to the destined by the same *MathematicalConversion* (e.g. +5 is invertible with -5). This design facilitates broader reuse because one description can be used for two elements. Currently no mathematical system is supplied. However, the *Adaptation* metamodel could be enhanced for this support in the future. The *TransformationalConversion* is a general purpose conversion and can only be described. Its design purpose is to contain specific functions like the exchange of strings. An example description of such a conversion is "Transform "name"-value of source to "name"-value of the target. The *ReferenceUsingConversion* is a special kind of conversion and references other information. This conversion is used to request additional information to transform the information. This conversion models additional data requirements for the conversion described in the*textualConversionDescription:EString* attribute.

All adaptation relations between data can be modelled as one-to-one relations. Thus, every two data instances are adapted according to a certain adaptation conversion. However, the number of one-to-one description increases further when more than two context related instances of data exist. For example, when one name (e.g. unit second) is used in multiple simulations with different languages. If two languages are available, one or two descriptions are required. The number of descriptions between two information depends on whether the conversion is "invertible" (i.e. usable in both directions) or not. When three information has to be adapted, 3 or 6 adaptations have to be specified. Let n be the number of adapted related information, then $\frac{(n*(n-1))}{2}$ (invertible) or $n*(n-1)$ (not invertible) descriptions have to be modelled. This approach creates an increasing modelling complexity. An increased complexity inhibits failures in the description of the adaptation (e.g. the modeller forgets one relation). The DSL provides special structures to express certain relations with a reduced number of relations to counter this complexity. These structures are provided by the sub-classes of the abstract *AdaptationDescription*. The use of the structures reduce the number of *AdaptationDescription* instances (e.g. by using multiple *DataMarker* in one model). The *AdaptationDescription* class along with its subclasses is contained in the *AdaptationDescriptions* package. Its content is displayed in Fig. 6.20.

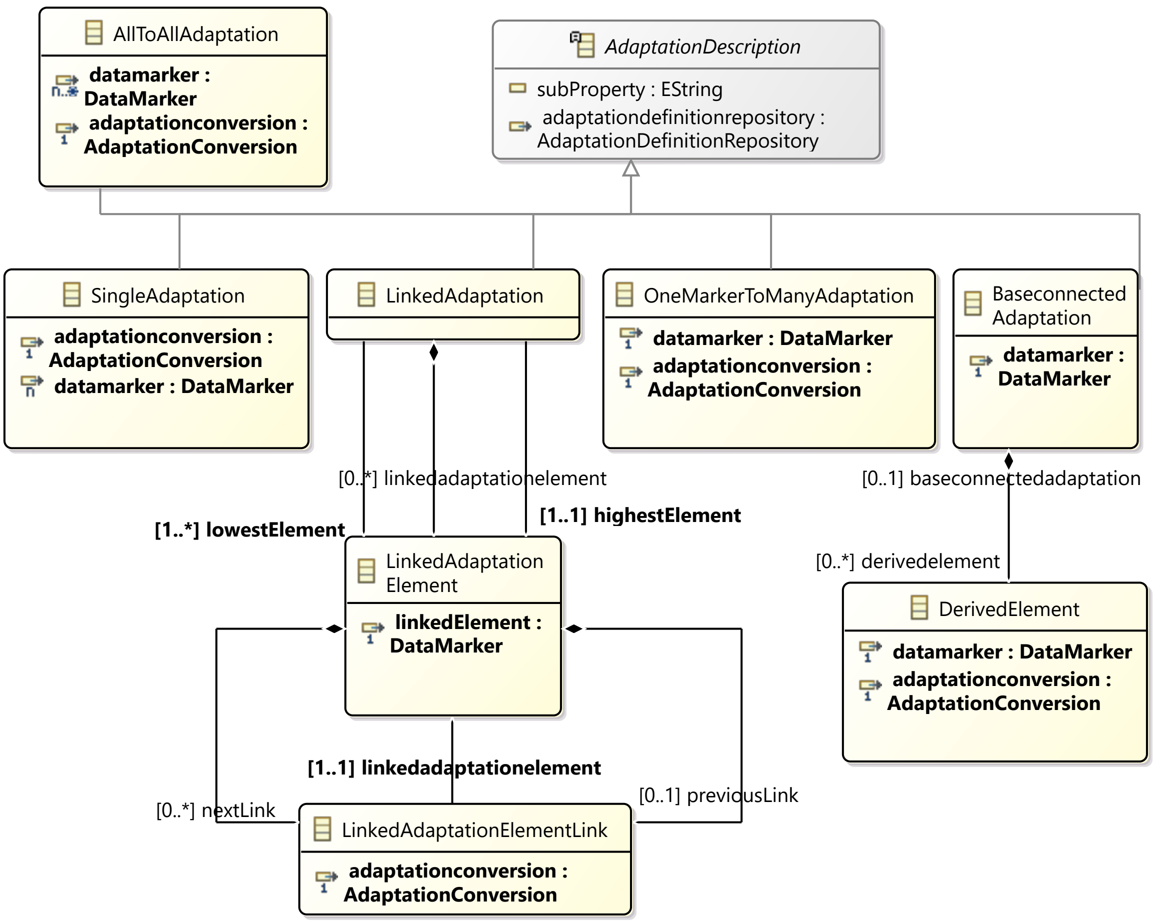

AllToAllAdaptation
datamarker : DataMarker
adaptationconversion : AdaptationConversion
AdaptationDescription
subProperty : EString
adaptationdefinitionrepository : AdaptationDefinitionRepository
SingleAdaptation
adaptationconversion : AdaptationConversion
datamarker : DataMarker
LinkedAdaptation
OneMarkerToManyAdaptation
datamarker : DataMarker
adaptationconversion : AdaptationConversion
Baseconnected Adaptation
datamarker : DataMarker
[0..*] linkedadaptationelement
[0..1] baseconnectedadaptation
[1..*] lowestElement
[1..1] highestElement
[0..*] derivedelement
LinkedAdaptation Element
linkedElement : DataMarker
DerivedElement
datamarker : DataMarker
adaptationconversion : AdaptationConversion
[1..1] linkedadaptationelement
[0..*] nextLink
[0..1] previousLink
LinkedAdaptationElementLink
adaptationconversion : AdaptationConversion

Figure 6.20: Classes and relations directly contained in the *ModularEnvironment* package

The most basic *AdaptationDescription* is the *OneToOneDescription*. This description should only be used when the other structures cannot be applied in a meaningful way. The

OneToOneDescription represents only two contextually related information to be adapted in one another. For this purpose, the *OneToOneDescription* instance references two instances of *DataMarker*. Also this description specifies how to execute the adaptation by referencing one *AdaptationConversion*. Another structure targets the adaptation of multiple information by the same process. An example is the already stated example of different languages. The words in the languages are known, and they have only to be translated. If all information is prior known, only the translation and the names have to be specified. This can be modelled by the *AllToAllAdaptation* metamodel element. In this element, two or more *DataMarker* instances are referenced. Also one *AdaptationConversion* has to be referenced to describe how to adapt the information. When data corresponding to two of the *DataMarker* instances in this description are encountered, the data is adapted as described in the *AdaptationConversion* instance. For example, the *AllToAllAdaptation* can be used for the "Hello" example of Neema [64]. Here, the names "Hello", "Bonjour" and "Namaste" have to be adapted into one another. In the *AllToAllAdaptation* the *DataMarker* instances with the corresponding values of the *name:EString* attribute have to be references. Then a "name-exchange" *AdaptationConversion* is supplied. One problem of the thought behind the *AllToAllAdaptation* is the possible quantity of referenced *DataMarker* instances. Also, the information described by instances must be known in advance to create all used *DataMarker* instances. The *OneMarkerToManyAdaptation* element is provided by the DSL to relax these conditions. This element provides the capability to reference one *DataMarker* and one *AdaptationConversion*. This description is provided to describe information like the *AllToAllAdaptation* when the *DataMarker* instances cannot be foreseen. As for example, multiple simulations describe an integer with their own data types. Thus, there are "XInt", "YInt" and "ZInt". The "text" for "X", "Y" and "Z" cannot be foreseen. The *OneMarkerToManyAdaptation* specifies only one *DataMarker* "Int" because of this unforeseeable elements. As mentioned in Sec. 6.2, the *DataMarker* are connected in the assembly of modular simulations to the data. Therefore all data can then be connected to the single *DataMarker*. This *OneMarkerToManyAdaptation* seems to define a more convenient application than *AllToAllAdaptation*. However, the *AllToAllAdaptation* provides a more exact description of the adaptation data. This can be easier understood in the case of a reuse scenario. Another identified structure originates from the definition of SI-Units as described in Sec. 6.2. SI-Units are defined by a base quantity (e.g. for time: second). Every other related quantity is derived of this base quantity. This structure is represented by the *BaseconnectedAdaptation* element. A instance of this element references one *DataMarker* as base marker. Also the *BaseconnectedAdaptation* contains zero or more instances of *DerivedElement*. Each *DerivedElement* references a *AdaptationConversion* and one *DataMarker*. To create a SI-Unit description for the time related quantities, a *BaseconnectedAdaptation* is required. Exemplary, the adaptation of the units second, minute and hour shall be described. Therefore, three *DataMarker* instances with the corresponding names are created. A created *BaseconnectedAdaptation* instance references the "second" *DataMarker*. Two *DerivedElement* instances are created for minute and hour. Each referencing one of both *DataMarker* instances. Each *DerivedElement* instance additionally references a *MathematicalConversion* which signals the calculation (*60 in minutes and *3600 in hours). The last structure provided by the DSL is the *LinkedAdaptation*. This description contains *LinkedAdaptationElement* instances and combines them by *LinkedAdaptationElementLink* to a "tree"-like structure. Each *LinkedAdaptationElement* defines a zero or one *previousLink* containment to a *LinkedAdaptationElementLink*. Also a zero to many *nextLink* references to *LinkedAdaptationElementLink* is defined. The *LinkedAdaptationElementLink* provides an *AdaptationConversion* and references a *LinkedAdaptationElement*. The *DataMarker* in a *LinkedAdaptationElement* marks the information of an element. The *LinkedAdaptation* enables another kind to describe relations. To define the first and last *LinkedAdaptationElement* instances in the tree (i.e. the leafs and the root), the *LinkedAdaptation* references *LinkedAdaptationElement* instances. In the proposed adaptation process in Sec. 6.2.2, this structure can be used to scan the elements on a more defined and efficient way.

6.8 Assembly of Modular Simulations

Modular simulations are used to create new simulations and behaviours out of existing simulations. Therefore, the used modules have to be connected. A MSE describes the capabilities to enable an interaction between the simulations. In the current DSLs definition of modular simulation assembly, the designed simulations and MSEs are combined to describe a modular simulation. Also models of other modular simulations can be used to facilitate a hierarchical composition. Adaptation is used to resolve incompatibilities between information of the independent designed elements of modular simulation.

This section describes the capabilities of the DSL to describe the assembly of modular simulations. First, the necessary elements to describe a single assembly are presented. Then, the capabilities to define the information used in the modular simulation is described. This definition includes the enhancement of information with MSE specific annotations. The enhanced information is mapped to interface to be described as provided or required by a simulation. This description is realised by the use of interfaces abstractly describing information. Also, the mapping of the abstract *AdaptationDescription* instances to the information are explained. To describe how the simulations and MSEs are connected, the model elements to connect required and provided information is defined. Also, the elements to describe the communication between the simulations and coordinators are provided.

6.8.1 Structure of Assembled Modular Simulation

Modular simulations are composed of independently designed simulation features and other modular simulations. The MSEs provide the capabilities to enable communication and information exchange between simulations. The coordinator is currently the only possible realisation for a MSE.

In the DSL, the abstract *Assembly*[SE] class is used as entry point to define assemblies. It is contained in the *ModularSimulationAssembly* package. The content of this package is shown in Fig. 6.21. The *Assembly* class provides the possibility to provide other assembly approaches in the DSL. For example, the DSL could be extended in the future by the ability to assembly different MSE models to one large MSE. The only existing sub-class of *Assembly* is currently the *SimulationAssembly* class. Every modular simulation is described by this class class. *AssembableComponent*[SE] encapsulates a simulation or MSE in the

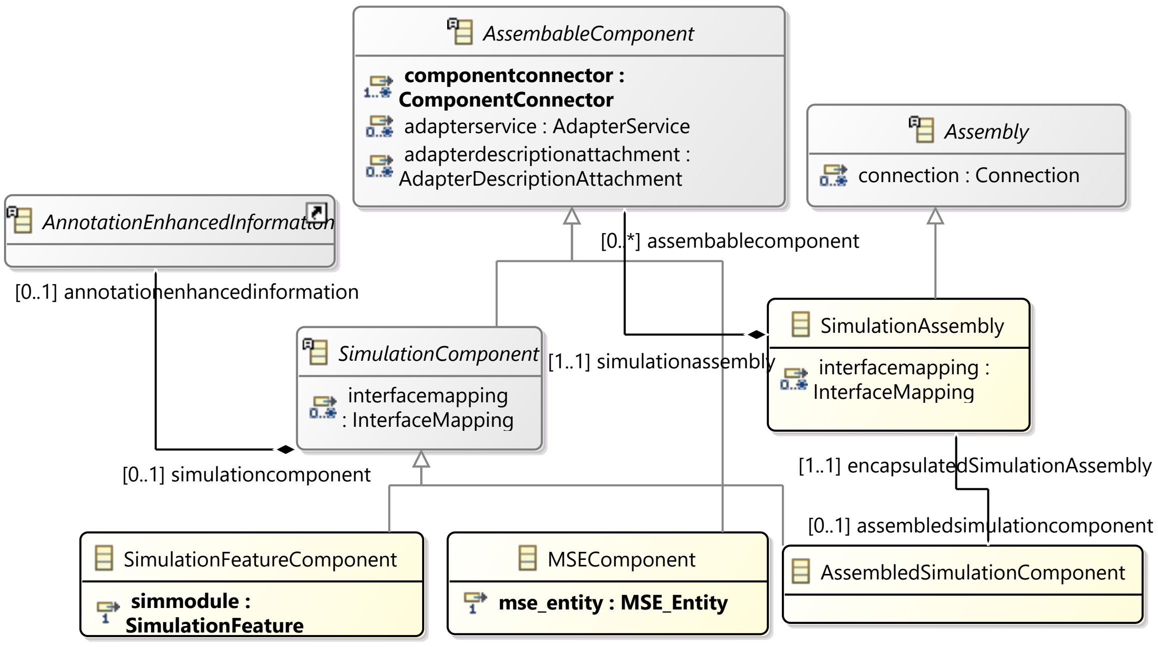

AssembableComponent
componentconnector : ComponentConnector
adapterservice : AdapterService
adapterdescriptionattachment : AdapterDescriptionAttachment
Assembly
connection : Connection
AnnotationEnhancedInformation
[0..*] assembablecomponent
[0..1] annotationenhancedinformation
SimulationComponent
interfacemapping : InterfaceMapping
[1..1] simulationassembly
SimulationAssembly
interfacemapping : InterfaceMapping
[0..1] simulationcomponent
[1..1] encapsulatedSimulationAssembly
[0..1] assembledsimulationcomponent
SimulationFeatureComponent
simmodule : SimulationFeature
MSEComponent
mse_entity : MSE_Entity
AssembledSimulationComponent

Figure 6.21: Classes and relations directly contained in the *ModularSimulationAssembly* package

modular simulation in a component (i.e. independent designed simulations, coordinator or modular simulations). With this encapsulation, the independently designed models do not have to be changed. The components enable the modular simulation assembly specific elements to be separated from the independent elements.

The simulation features, coordinators and modular simulations have to provide different definitions to be coupled in a modular simulation. Therefore, three different subtypes of *AssembableComponent* are provided. Each component references one of the corresponding models used in a modular simulation. These components are

- *SimulationFeatureComponent*: Provided for the use of independent modelled basic simulations. This component references an instance of *SimulationFeature*
- *MSEComponent*: References a *MSE_Entity*. This enables the use of an MSE to provide functionality for simulation information exchange, coordination and communication
- *AssembledSimulationComponent*: encapsulates a *SimulationAssembly* instance through a reference. This elements enables the use of modular simulations in the assembly

To define provided and required interfaces, the *AssembledSimulationComponent* and *SimulationFeatureComponent* sub-type the abstract *SimulationComponent*. The abstract class itself subtypes *AssembableComponent*. Thus, both *SimulationComponent* subtypes only indirectly inherit from *AssembableComponent*. Because *AssembledSimulationComponent* and *SimulationFeatureComponent* both subtype *SimulationComponent*, they cannot be differentiated in the use of a modular simulation. This design enables the modeller to either use simulation features or modular simulations. Because of this design the models of the simulations are then exchangeable.

The encapsulation of components enforces the reuse of the independent designed parts with different assembly specific properties. The benefit of this design is stressed by the ability to use the same simulation with different MSEs. As described in Sec. 6.6.2.1, every MSE is able to define own annotations. Information types (e.g. object class or properties) have to specify values for annotations corresponding to a coordinator models *AnnotationInterface* instances. Information not providing values for each annotation cannot be handled by the MSE. If the annotations would be described in a *SimulationFeature* model instance, other MSEs could not be used with it. The exchange of assembly components, however, is a substantial property of the DSL. Thus, only the encapsulating *AssembableComponent* model has to be changed. The encapsulated *SimulationFeature* can be designed independently of any coordinator through this approach. Another benefit of the encapsulation in components is the possible further enhancements of the DSL.

The *AdaptationDescription* have to be linked to the information used in the *SimulationAssembly* instance and adapter services. This is realised by "attachments" to adapter services contained in the components. Another important factor is to describe the required and provided information in a modular simulation. This description enables the exchange of simulation components. The definition of required and provided information also enables an exact specification of the information exchange between simulations. The connections used in a modular simulation have to be defined. These connection define the information flow between the components. These mentioned topics are described in the remainder of this section.

6.8.2 Annotation of Information with Context Information

The available and required information of a simulation have to be explicitly specified as used in a modular simulation. As stated in Sec. 6.6.2.1, the MSE needs certain contextual information to process information. This is defined by *Annotation* instances of the MSEs. All information exchanged in the assembled simulation must be enhanced with these attachments and their values to be processable. Thus, the values for annotations have to be modelled for information used in the modular simulation. In the DSL this is done for each *SimulationComponent*. The DSL provides an encapsulating model element for each information type existing in a simulation and defined in the MSE. This encapsulation enables the separation of modular simulation assembly specific information and independent designed information. An object-oriented structure as described in Sec. 6.5.3 is created. The replication of the object-oriented structure is caused by the design decision of the DSL stated in Sec. 6.5.4.

Every information has to be annotated with the values of *Annotation* instances of the MSE. In the DSL, the assertion of the values of *Annotation* instances to information is realised by the *AnnotationSetter* metamodel elements described in Sec. 6.6.2.1. The abstract class *AnnotationEnhanced*[E] provides a reusable element for the selection of *AnnotationValue* instances for information used in the modular simulation. It is located in the *AnnotationEnhancement* package. Its contained metamodel classes are depicted in Fig. 6.22. This class provides the capabilities to create instances of *AnnotationSetter*. Every setter references one *Annotation* instance. *AnnotationSetter* enables the modeller to specify the values according to the sub-type of the *Annotation*. The *AnnotationInterface* of the

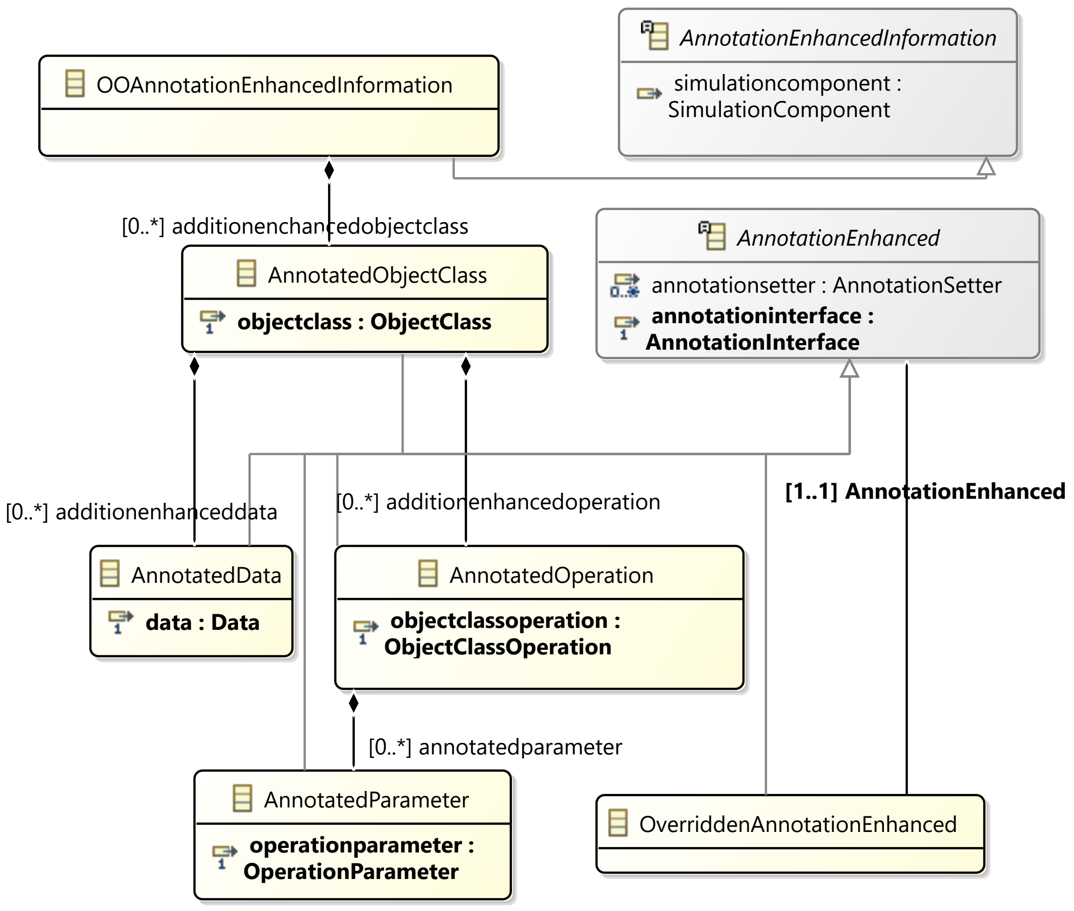

OOAnnotationEnhancedInformation
AnnotationEnhancedInformation
simulationcomponent : SimulationComponent
[0..*] additionenchancedobjectclass
AnnotatedObjectClass
objectclass : ObjectClass
AnnotationEnhanced
annotationsetter : AnnotationSetter
annotationinterface : AnnotationInterface
[0..*] additionenhanceddata
[0..*] additionenhancedoperation
[1..1] AnnotationEnhanced
AnnotatedData
data : Data
AnnotatedOperation
objectclassoperation : ObjectClassOperation
[0..*] annotatedparameter
AnnotatedParameter
operationparameter : OperationParameter
OverriddenAnnotationEnhanced

Figure 6.22: Classes and relations contained in the *AnnotationEnhancement* package

applied MSE has to be referenced by each *AnnotationEnhanced*. This reference enables an editor to support the selection of the correct interface for an information type (e.g. ObjectClass or Data). Also an editor could support the modeller that *AnnotationSetter* instances for all *Annotation* instances in the interface are defined.

Four metamodel classes exist to reproduce the object oriented structure to encapsulate the independent designed information. These classes correspond to the object oriented structure described in Sec. 6.5.3. These four classes are *AnnotatedObjectClass*, *AnnotatedData*, *AnnotatedOperation* and *AnnotatedParameter*. All four classes sub-type the *AnnotationEnhanced* class to gain the capabilities to select *AnnotationValue* instances. The *AnnotatedObjectClass* references one *ObjectClass* instance. The instance of *AnnotatedObjectClass* can contain instances of the *AnnotatedOperation* class and *AnnotatedData* class. Also *AnnotatedOperation* can contain instances of the *AnnotatedParameter* element. This structure shows the drawback of the decision stated in Sec. 6.5.4. The structure needs four metaclasses. Each of these metamodel elements encapsulates one instance of the corresponding information type by referencing it. *AnnotatedObjectClass* references one *ObjectClass*, *AnnotatedOperation* one *Operation* and *AnnotatedData* one *Data* instance. Also *AnnotatedParameter* references one *OperationParameter*. For each information type, the values of the required *Annotation* instances have to be set. The *Annotation* instances to be used are specified by the *AnnotationInterface* of the used MSE.

The prior mentioned *AnnotationEnhanced* classes encapsulates information types provided by simulation features. Modular simulations only use information required or provided of its contained simulations. If the modular simulation is used in another assembly, it is possible that another MSE is used. In this case, the information has to be fitted to the *Annotation* of this other MSE. The DSL provides the *OverriddenAnnotationEnhanced* element for the representation of this approach. It is contained in *AssembledSimulationComponent*. The *OverriddenAnnotationEnhanced* also subtypes *AnnotationEnhanced* and references one *AnnotationEnhanced* element. This enables the "override" of an annotation of the *AnnotationEnhanced* information. The encapsulation *AnnotationEnhanced* also allows to keep the representation of the annotated information when used in the *AssembledSimulationComponent*.

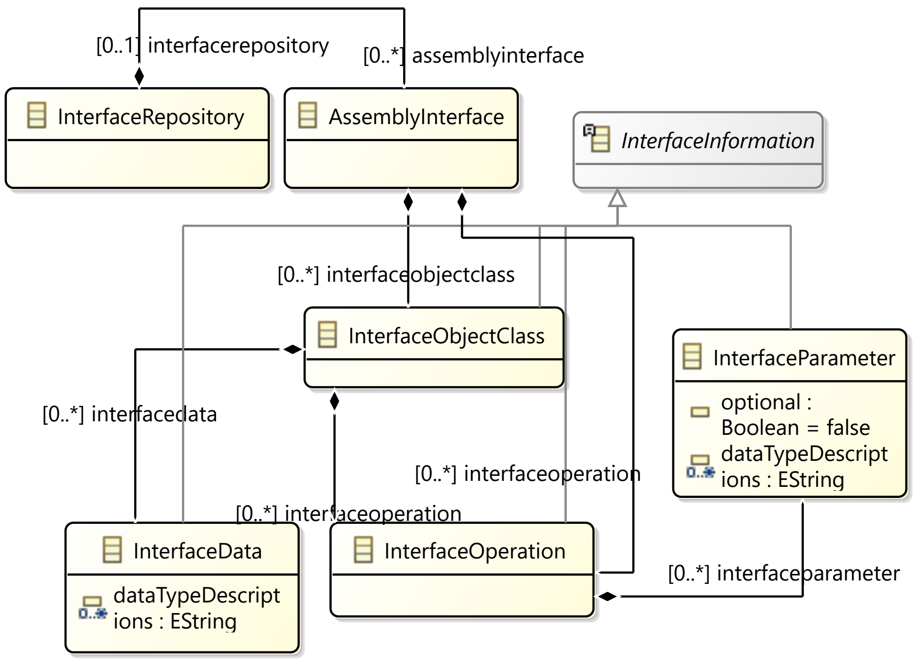
6.8.3 Definition of Required and Provided Information of Simulations

The annotation of information described in the prior section Sec. 6.8.2 defines information available for use in the modular simulation assembly. Each of the annotated information can be marked as provided or required by a simulation component. The DSL uses interfaces to define the information to be exchanged in a modular simulation. Each interface represents specific information to be exchanged. These interfaces are used in modular simulation to describe the information simulations provide and require. These interfaces can be reused by different modular simulations when the same information is exchanged. The definition of interfaces also allows an exchange of the component and the underlying simulation. Only the required and provided interfaces have to correspond with the other component to be exchangeable. The description of information interfaces in the DSL is specified in the following subsection. This specification is followed by the explanation how the DSL realises the mapping of interfaces to define information as required and provided.

6.8.3.1 Description of Information Interfaces

One reason for creating modular simulations is to use other simulations to produce the desired behaviour. These simulations can depend on information of other simulation. It is possible that one model requires data from another model to be executed correctly. In IntBIIS, for example, the business process requires the response time of Palladio components to be able to calculate its model correctly. IntBIIS is confined with Palladio in one large simulation in a monolithic simulation. If the simulations are separated, their required and provided information have to be specified to make this requirement explicit. To define data to be exchanged by simulations in a reusable and independent way, the notion of interfaces as explained in Sec. 3.2.1 is used. The content of an interface defines the information to be exchanged. The interface expresses an abstract point of interaction between multiple simulations. One simulation does not know how the other simulation creates the information. Therefore, the definition of the information in a simulation has to suffice.

The design of the interface does not include the simulation using this interface. Contrary to the interface notion of Palladio, not only operations but also object classes and data are defined in an interface. The DSL provides the *InterfaceDefinition* package to enable the modelling of interfaces. Its content is depicted in Fig. 6.23 Its entry-point for interface


[0..1] interfacerepository
[0..*] assemblyinterface
InterfaceRepository
AssemblyInterface
InterfaceInformation
[0..*] interfaceobjectclass
InterfaceObjectClass
InterfaceParameter
optional : Boolean = false
dataTypeDescriptions : EString
[0..*] interfacedata
[0..*] interfaceoperation
[0..*] interfaceoperation
InterfaceData
dataTypeDescriptions : EString
InterfaceOperation
[0..*] interfaceparameter

s

Figure 6.23: Classes and relations contained in the *InterfaceDefinition* package

definition is the *InterfaceRepository*[SE] element. The purpose of this repository is to represent multiple interfaces for one defined context. For example, information to be exchanged between Palladio and and the business process simulation. In IntBIIS the information can be the response time or the operation to execute the simulation of a specific software system. The collection of interfaces in the repository enables modellers to reuse the definitions for guidelines on what data typically flows between simulation in a certain context. One example is the context of response time calculation of software systems with components. The response time of components has to be transferred. Therefore, the response time has to be contained in an interface.

The *InterfaceRepository* contains zero or more instances of *AssemblyInterface* to model interfaces in the DSL. The DSL has to replicate the designed object-oriented structure of the metamodel to represent informations to be transferred. The DSL provides the classes *InterfaceObjectClass*, *InterfaceData*, *InterfaceOperation* and *InterfaceParameter* for this purpose. All four classes sub-type the abstract *InterfaceInformation*[SE] class. The representation of the object-oriented structure enables direct mapping between the sub-types of *AnnotationEnhanced* and the interfaces. A *AssemblyInterface* can contain zero or more *InterfaceObjectClass* instances. Each *InterfaceObjectClass* can define zero or more *InterfaceData* and *InterfaceOperation* instances. Each *InterfaceOperation* contains zero or more *InterfaceParameter* instances.

The descriptions of the content of interfaces are kept on an abstract level and do not define a concrete type. This design is necessary because of the independent development of simulations and MSE. The approach of high abstraction is similar to the one in the adaptation descriptions of Sec. 6.7.2. A description of a certain data type would hinder the interaction between the two simulations. If one simulation uses a Double to represent "percent" and another uses an Integer, a type definition would not allow one of these simulations to use the interface. Therefore, *InterfaceData* and *InterfaceParameter* contains only a textual description of the semantics of the informations data type. This description is realised by the *dataTypeDescriptions:EString* attribute. These descriptions enable a developer to check whether the semantics of its provided data fits to the interface. The interplay between adapter and abstract interface description enables the interoperability simulations. The adapter approach transforms the values in the other format. The *InterfaceParameter* also contains the *optional:Boolean* attribute to signal if a parameter is optional. This attribute describes the same optionality as of parameters in operation as presented in Sec. 6.5.2. The created *AnnotationEnhanced* information can be described as either required or provided by a simulation component with the interface descriptions.

6.8.3.2 Mapping of Enhanced Information to Interfaces

Each interface can be used to describe the information a simulation requires and provides. To define the provided or required information in a modular simulation, the *AnnotationEnhanced* information of a component has to be mapped to an interface. If one simulation marks an interface as provided and another simulation the same interface as required, the information described in this interface can be exchanged. Contrary to interfaces in Palladio, a mapping of the underlying types of the information has to be provided to the interface. This approach allows the identification of incompatibilities between the required and provided information. This design is motivated by the adaptation approach of the DSL. The underlying specifics of the information must be known (e.g. the data types) to adapt information exchanged by simulations. This knowledge enables to create exact *AdaptationDescription* instances for the simulation assembly if it is not already available.

To describe the mapping of information to an interface in the DSL, each *SimulationComponent* provides containments to the *InterfaceRequired* and *InterfaceProvided* classes. These metamodel classes are located in the *InterfaceMapping* package. The packages content is depicted in Fig. 6.24. Both classes are sub-types of the abstract *InterfaceMapping* class. *InterfaceMapping* references one *AssemblyInterface* to define the targeted interface. This allows to verify if the interface is correctly mapped. The sub-classes signal whether a component requires or provides the information of the referenced *AssemblyInterface*. The *RequiringObjectClassMapping* references the *RequiredDataEntry* of the simulation to specify the interface required by a simulation component. This design allows to check if the required information and the *AnnotatedInformation* corresponds. Additionally, the *AnnotatedObjectClass* is specified to provide the specifics of the information. The reference to *InterfaceObjectClass* provides a mapping to the entry in the interface. This enables an editor to control the correct mapping of *InterfaceObjectClass* and the *AnnotatedObjectClass*. From this class, another object-oriented structure is created. This includes zero to many containments to the classes *DataToInterfaceMapping* and *OperationToInterfaceMapping*.

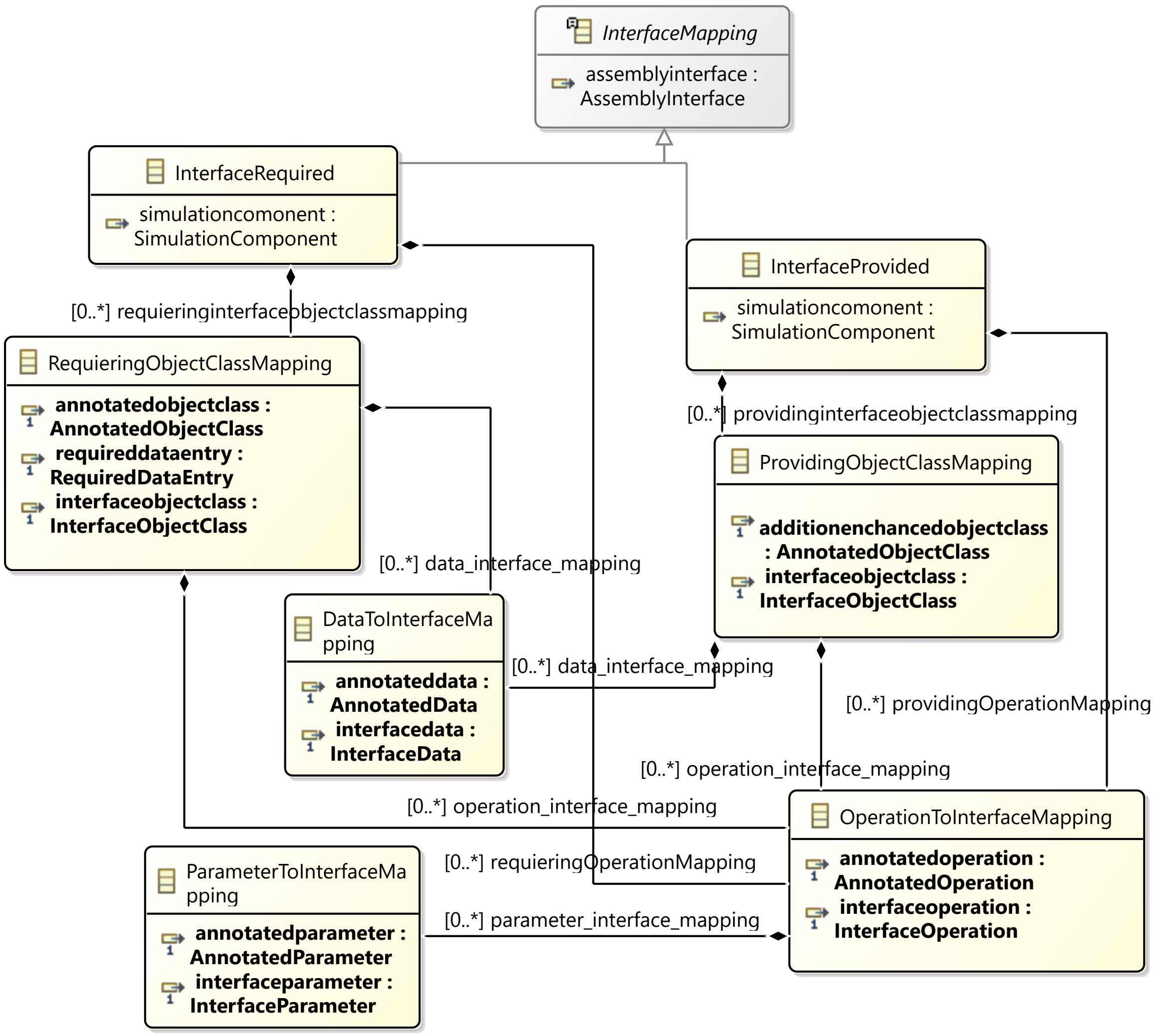

InterfaceMapping
assemblyinterface : AssemblyInterface
InterfaceRequired
simulationcomonent : SimulationComponent
InterfaceProvided
simulationcomonent : SimulationComponent
[0..*] requieringinterfaceobjectclassmapping
[0..*] providinginterfaceobjectclassmapping
RequieringObjectClassMapping
annotatedobjectclass : AnnotatedObjectClass
requireddataentry : RequiredDataEntry
interfaceobjectclass : InterfaceObjectClass
ProvidingObjectClassMapping
additionenchancedobjectclass : AnnotatedObjectClass
interfaceobjectclass : InterfaceObjectClass
[0..*] data_interface_mapping
DataToInterfaceMapping
annotateddata : AnnotatedData
interfacedata : InterfaceData
[0..*] data_interface_mapping
[0..*] providingOperationMapping
[0..*] operation_interface_mapping
[0..*] operation_interface_mapping
OperationToInterfaceMapping
annotatedoperation : AnnotatedOperation
interfaceoperation : InterfaceOperation
ParameterToInterfaceMapping
annotatedparameter : AnnotatedParameter
interfaceparameter : InterfaceParameter
[0..*] requieringOperationMapping
[0..*] parameter_interface_mapping

Figure 6.24: Classes and relations contained in the *InterfaceMapping* package

The *OperationToInterfaceMapping* contains zero or more references to the *Parameter-ToInterfaceMapping* class. Each of these classes references the corresponding type of the *AnnotationEnhanced* structure and the *Interface* structure. This allows to prohibit all mappings except of the ones between the definitions of the information types.

A modular simulation has to satisfy the requirements of the components. The requirement has to be deferred if not enough modules are contained in the modular simulation to satisfy all requirements. The modular simulation itself has to describe the information as required. This description is modelled by references to *InterfaceRequired* instances in the *SimulationAssembly* class of the *ModularSimulationAssembly* package. The modular simulation can also provide information itself. However, only information defined as provided of the contained simulation components can be provided by a modular simulation. This provision is modelled by references of the *ModularSimulationAssembly* class to the *InterfaceProvided* class. The modular simulation can be used with other simulations in another modular simulation description through deferring of provided and required information.

6.8.4 Attachment of Adaptation Descriptions to Adapters

Adaptation in the DSL is described by two parts as presented in Sec. 6.7. The *AdapterService* is created in components and the independent designed *AdaptationDescription. AdaptationDescription* instances abstractly describes the information to adapt to facilitate reuse in other modular simulations. Also the application of the adaptation is described in *AdaptationConversion* models. The *AdaptationDescription* instances are not directly modelled in *AdapterService* elements. This design is prohibited because the *AdaptationDescriptions* are designed independently of concrete modular simulation. A direct binding to an *AdapterService* would only make them reusable in the same modular simulation. The *AdaptationDescription* instances to apply can only be determined in the modular simulation assembly. Here, all exchanged information and simulations are known. The *AdaptationDescription* instances have to be attached to the *AdapterService* instances in the assembly because of this design.

This ability is provided by the *AdapterDescriptionAttachment*[I] in the *Adaptation* package as depicted in Fig. 6.18 of Sec. 6.7. *AdapterServices* instances and *AdapterDescriptionAttachment* instances are created in a *AssembableComponent.* This allows for every component to access adaptation capabilities. *AdapterDescriptionAttachment* instances reference one *AdaptationDescription* and one *AdapterService* model. These references define what adaptations a *AdapterService* instance has to realise. Multiple *AdaptationDescriptions* with the same name can exist simultaneously because *AdaptationDefinitionRepository* are modelled independently. Therefore, the *AdapterDescriptionAttachment* also references the *AdaptationDefinitionRepository* the *AdaptationDescription* is supposed to be defined in. This design supports the modeller in the modelling of the adaptations in modular simulations and provides tools to reduce the probability of confusion. Also, additional editor support is possible in the future to avoid mix-ups of different *AdaptationDescription* instances.

AdaptationDescription instances are still not applicable by the *AdapterService* with the previously mentioned elements. This problem exists because the information, which is targeted by the *AdaptationDescription*, is only abstractly described by *DataMarker* instances. The information in the modular simulation has to be connected to the *DataMarker* in the *AdaptationDescription* to be adaptable. Otherwise the *AdapterService* could apply the wrong *AdaptationDescription* to the wrong information. Therefore, the *DSL* provides the capability to connect *DataMarker* instances in the *AdaptationDescription* to elements of the super-type *Adaptable.* This connection to *Adaptable* instances is modelled by the *AdaptationMarkerMapping*[I] element. *AdaptationMarkerMapping* therefore references on *DataMarker* instance and one or more *Adaptable* instances. The *Adaptable* super-type marks the information that can be adapted by the *AdapterService* instances. The abstract *Adaptable* metaclass is located in the *AdapterServices* package. Currently, metaclasses sub-typing *Adaptable* are

- ObjectClass

- Data
- Operation
- ObjectInstance
- AnnotationEnhanced
- DataType

With the attachment of *AdaptationDescription* to the *AdapterService* instances and the *AdaptationMarkerMapping* instances, the proposed process of Sec. 6.2.2 can be realised.

6.8.5 Connections in the Modular Simulation Assembly

The connection between components is necessary to describe the architectures of a modular simulation. Two types of connections are identified in the DSL. One type of connection is realised between simulations in respect to the the interface mappings in the modular simulation. The other type of connection is related to the interaction between components. Components are able to communicate with another component if a connection exists between them. For example, each simulation component is directly connected to a coordinator component in a coordinator-based MSE. Furthermore, no connection between two simulation components exist in the coordinator-example. The *Assembly* class in the *ModularSimulationAssembly* package contains zero or more instances of the abstract *Connection*[E] class. This class is contained in the *AssemblyConnections* package and provides the entry-point for the definition of new connection types. *Connection* contains a reference to the *Assembly* instance it is created in.

6.8.5.1 Connection between Required and Provided Interfaces

One type of connection between components is the relation of interfaces in the modular simulation assembly. The definition of this connection enables the identification of the flow of information between simulations. This type of connection is realised by the sub-class *ComponentInterfaceConnection* of *Connection* in the DSL. All elements of this sub-type are provided in the *ComponentInterfaceConnection* package depicted in Fig. 6.25 Two conceptual sub-types are differentiated. One sub-type is the *RequiredProvidedInterfaceConnection* class, which references one *InterfaceRequired* and a *InterfaceProvided* instance. These interfaces have to reference the same *AssemblyInterface*. This can be checked by an editor due to the reference of a *AssemblyInterface* in the *InterfaceMapping* class. The referencing of a *InterfaceRequired* and a *InterfaceProvided* instance defines the connection between those two interface realisations. The *RequiredProvidedInterfaceConnection* class additionally references the corresponding providing and requiring components to describe the complete connection. This connection enables to verify if the referenced interfaces correspond to the intended components. This is done by the *providingComponent* and *requieringComponent* reference. The connection between interfaces enables a modeller to inspect if all required or provided interfaces are satisfied.

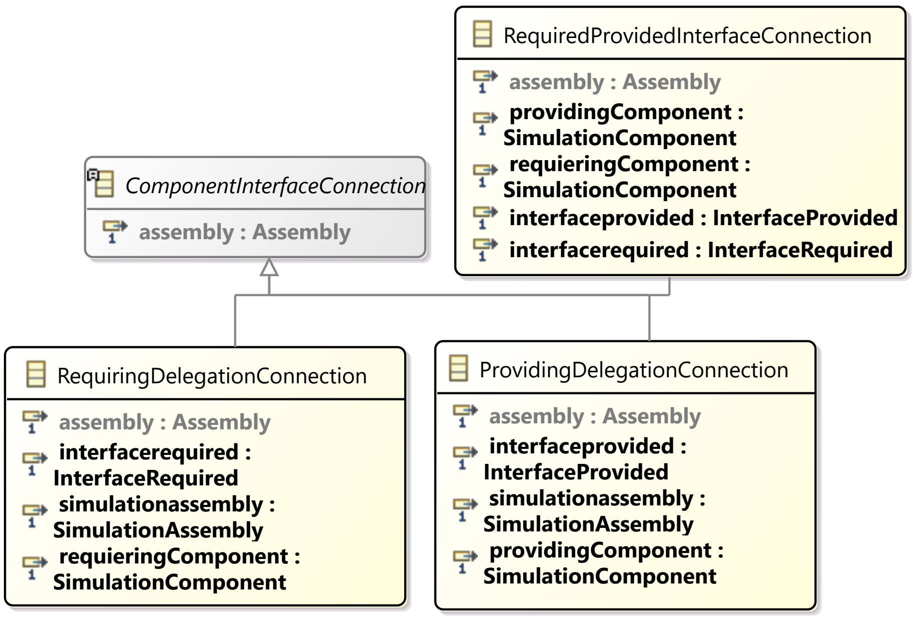

RequiredProvidedInterfaceConnection
assembly : Assembly
providingComponent : SimulationComponent
requieringComponent : SimulationComponent
interfaceprovided : InterfaceProvided
interfacerequired : InterfaceRequired
ComponentInterfaceConnection
assembly : Assembly
RequiringDelegationConnection
assembly : Assembly
interfacerequired : InterfaceRequired
simulationassembly : SimulationAssembly
requieringComponent : SimulationComponent
ProvidingDelegationConnection
assembly : Assembly
interfaceprovided : InterfaceProvided
simulationassembly : SimulationAssembly
providingComponent : SimulationComponent

Figure 6.25: Classes and relations contained in the *ComponentInterfaceConnection* package

The *SimulationAssembly* class references *InterfaceMapping* instances to signal that the information contained in the mapping is deferred to another simulation assembly. The DSL includes two subclasses of *ComponentInterfaceConnection* to connect the component whose interface is deferred. The sub-classes are named *RequiringDelegationConnection* and *ProvidingDelegationConnection*. Both classes reference the *SimulationAssembly* class to mark the simulation assembly to be connected to an *InterfaceMapping*. The connection has to mark the interface providing or requiring the information as well as the corresponding component. This design allows to inspect if all requirements are satisfied in the modular simulation with these delegations.The component containing the interface mapping of the delegation is described by a reference to a instance of *SimulationComponent*. The *RequiringDelegationConnection* and *ProvidingDelegationConnection* are separated to map to the correct sub-type of *InterfaceMapping*. This design allows a more direct mapping and enables an easier. Additionally the important required interfaces can be explicitly verified with this approach. For this purpose, *RequiringDelegationConnection* references one *InterfaceRequired* instance and the *ProvidingDelegationConnection* references one *InterfaceProvided* instance. This allows to model and also validate the connections to the corresponding instances of *SimulationAssembly*.

6.8.5.2 Connections between Components

The second type of connection describes the communication between the components. This connection type provides a more implementation oriented definition. For this purpose, the points of connection have to be defined. These points are called connectors. The connectors define the entities enabling the sending and receiving of communication. In a network-based approach, these entities are clients or servers where the data streams are created. Ambassadors are a type of connector in the HLA implementations. The connector in the DSL is represented by the abstract class *ComponentConnector* which is defined in the *AssemblyConnections* package shown in Fig. 6.26

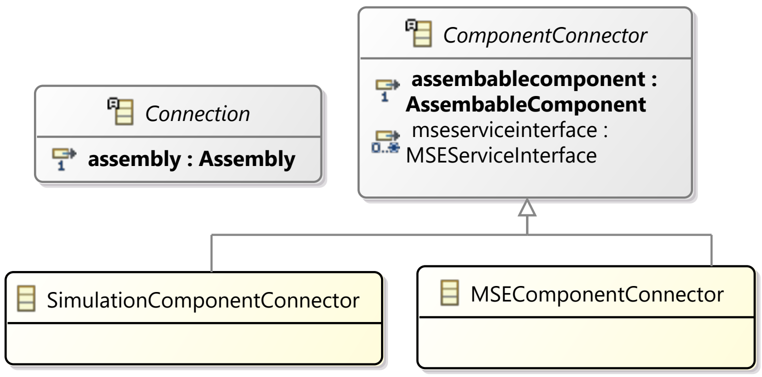

ComponentConnector
assembablecomponent : AssembableComponent
mseserviceinterface : MSEServiceInterface
Connection
assembly : Assembly
SimulationComponentConnector
MSEComponentConnector

Figure 6.26: Classes and relations contained in the *AssemblyConnections* package

Each *AssembableComponent* has to define at least one *ComponentConnector* to be connectible with other components. The *ComponentConnector* also references the corresponding *AssembableComponent*. The two classes *SimulationComponentConnector* and *MSEComponentConnector* subtype *ComponentConnector*. These sub-types are used to control the connections between components. The *ComponentConnector* element references several instances of the *MSEServiceInterface* class defined by the *MSE*. These references provide operations to access capabilities of the component. Also, these interfaces can contain the operations the simulation has to provide to be notifiable by the MSE. The sub-typing of *SimulationComponentConnector* and *MSEComponentConnector* from *ComponentConnector* introduces a design-weakness in the DSL. Because *ComponentConnector* includes a reference to *AssembableComponent*, its sub-classes cannot only specify their intended component. This design allows simulations to have a *MSEComponentConnector*. However, this design decision provides capabilities to extend the DSL. As stated in Sec. 6.6.2 currently only the coordinator scheme is represented as MSE. However, another approach would be to place the *MSE* capabilities in simulations themselves. With this approach, simulations would require *SimulationComponentConnector* as well as *MSEComponentConnector* instances.

The connection between two components of a modular simulation assembly is called a "wiring" in the *DSL*. This wiring can exist between several component types. For example, a wiring can exist between a simulation and MSE component. However, other connections are possible like one between two MSE components. The DSL provides the flexibility for the definition of such approaches. However, they are currently not included in the DSL. For example, two MSEs can be used in one modular simulation. Each MSE provides its capabilities. The complete capabilities of the modular simulation are created of both *MSE*s. The reasoning for this approach is to deploy both components on different machines to reduce possible bottlenecks.

A wiring is realised by the abstract *WiringConnection* class in the DSL. This class and all of its sub-classes are contained in the *ComponentWiring* package depicted in Fig. 6.27. This class itself does not provide references to *ComponentConnector* instances. These references

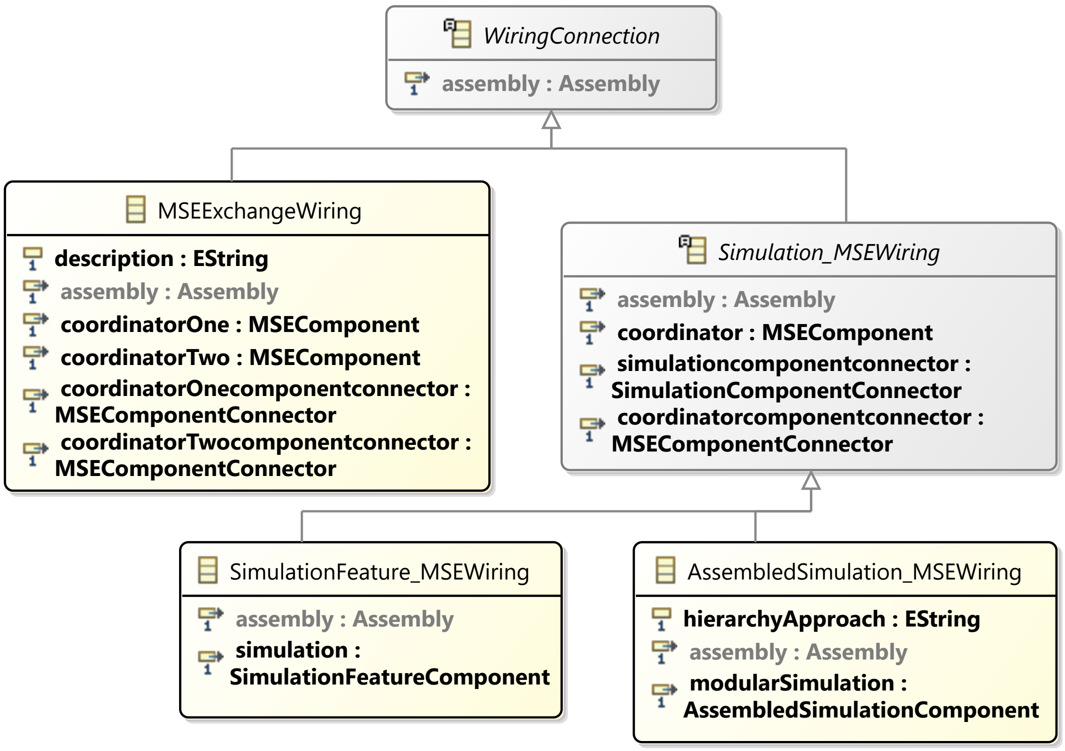

WiringConnection
assembly : Assembly
MSEExchangeWiring
description : EString
assembly : Assembly
coordinatorOne : MSEComponent
coordinatorTwo : MSEComponent
coordinatorOnecomponentconnector : MSEComponentConnector
coordinatorTwocomponentconnector : MSEComponentConnector
Simulation_MSEWiring
assembly : Assembly
coordinator : MSEComponent
simulationcomponentconnector : SimulationComponentConnector
coordinatorcomponentconnector : MSEComponentConnector
SimulationFeature_MSEWiring
assembly : Assembly
simulation : SimulationFeatureComponent
AssembledSimulation_MSEWiring
hierarchyApproach : EString
assembly : Assembly
modularSimulation : AssembledSimulationComponent

Figure 6.27: Classes and relations contained in the *AssemblyComponentWiring* package

are left to the sub-classes of *WiringConnection*. One type is the wiring between two MSE components. This approach aims to describe the representation of modular simulations with more than one MSE. The *CoordinatorExchangeWiring* subtypes *WiringConnection*. This subtype allows the provision of capabilities for different design approaches of the modular simulation. *CoordinatorExchangeWiring* contains the attribute *description:EString* to define what information is exchanged between the coordinator. Also, two instances of *MSEComponent* are referenced. One reference is named *coordinatorOne* and the other *coordinatorTwo*. These two references specify *MSEComponent* instances to be connected by this wiring. Also two references to the *MSEComponentConnector* exists. These references specify the connector instances used in the wiring. The referenced connector instances have to correspond with the referenced components. This correspondence can be verified by an editor. The other type of wiring defines a connection between a *SimulationComponent* instance and a *MSEComponent* instance. For this purpose, the abstract class *Simulation_MSEWiring* is provided. It sub-types *WiringConnection*. The *Simulation_MSEWiring* class references one *MSEComponent* and a *MSEComponentConnector*. Also one *SimulationComponentConnector* is referenced. The *Simulation_MSEWiring* differentiates between the modelling of connections to *SimulationFeatureComponent* instances and *AssembledSimulationComponent* instances. Therefore, the two classes *SimulationFeature_MSEWiring* and *AssembledSimulation_MSEWiring* sub-class *Simulation_MSEWiring*. The *SimulationFeature_MSEWiring* references a *SimulationFeatureComponent* instance. The *AssembledSimulation_MSEWiring* references a *AssembledSimulationComponent* instance. The differentiation between those two components originates from the hierarchical structure of simulations. An approach must be provided to enable components of two hierarchies to interact together. In the DSL, this approach is described by the *hierarchyApproach:EString* attribute of *AssembledSimulation_MSEWiring*. No further modelling can be provided due to the complexity of this topic. The problems concerning hierarchies of simulations is discussed in the following Sec. 6.8.6.

6.8.6 Hierarchical Assembly of Modular Simulations

A modular simulation component must be indistinguishable from a simulation feature component to facilitate a modular description of an assembly hierarchy. This property is stated by Kim and Kim [84]. Thus, there cannot be specific connections between coordinators of different levels of hierarchy. The approach to connect MSEs of different hierarchy level must be implementation dependent. However, to provide the capability to model these approaches, the *AssembledSimulation_MSEWiring* element is used in the *DSL*. It specifies that a *AssembledSimulationComponent* is connected with a *MSEComponent*. This wiring contains the attribute *hierarchyApproach:EString*. This attribute is supposed to supply a textual information on how the hierarchical approach is to be realised like, for example, through approaches of Cai et al. [85]. Each *AssembledSimulation_MSEWiring* instance uses the connector references of *Simulation_MSEWiring*. Thus, an internal coupling can be achieved between the connector in the encapsulated *AssembledSimulationComponent* model and a *MSEComponentConnector* of the current hierarchy

6.9 Role Based Modular Simulation Development with the DSL

Different developer roles are proposed to use the DSL for the creation of different parts of a modular simulation. This separation of roles can result in increased efficiency of development due to the fields of expertise the developers can provide. Also, a parallel or interleaved development flow is also possible. For this purpose, the roles of **simulation developer**, **adaptation developer**, **MSE-developer** and **simulation architect** are defined. The roles use different views on the underlying metamodel with two viewpoints. This approach is similar to the roles provided in the component-based software specification of Palladio as described in Sec. 3.2.2. The viewpoints on the systems are structured in the simulation assembly independent (development) viewpoint and the simulation assembly dependent (development) viewpoint. These viewpoints are only called **independent viewpoint** and **dependent viewpoint** in the this chapter. In the independent viewpoint, the roles use the DSL to model parts of the modular simulation independent of each other. This includes, for instance, the MSE and simulation feature representations. In the dependent viewpoint, the simulation architect interacts with the other roles to create a simulation assembly. In some cases, multiple developers have to interact to connect each part of a simulation assembly.

6.9.1 Simulation Developer

The simulation developer is responsible for the descriptions, extraction and development of simulation features. In the **independent viewpoint**, the simulation developer is responsible for modelling the simulation by use of the DSL. Another responsibility lies in the creation of the implementation of the simulation to be usable. The simulation developer utilizes the *SimulationFeature* package for the modelling aspect. The simulation developer provides a description of simulation as well as the available and required information contained in the simulation. When the model is finished, the simulation developer implements the modelled simulation feature. Because there is currently no model-to-text transformation for the DSL, the simulation developer has to implement the simulation feature according the model.

Another task of the simulation developer is the extraction of (sub-)simulations out of monolithic simulations. This task is assigned to the **dependent viewpoint**. A simulation developer has to extract simulations when the simulation architect does not have the simulations to describe the desired behaviour of the modular. The extraction is only possible if the monolithic simulation and the contained behaviour is already identified. This identification has to be done by the simulation architect due to the knowledge about the behaviour needed in the modular simulation. The DSL can be used in two different approaches in the extraction process. In the first approach, the simulation developer uses the DSL to model the (sub-)simulations before or during the process of extraction. The application of this approach enables to grasp the completeness of the extractions. For example, if all information is defined in each model (for example possible interactions). Also, it can be explored if probable starts of events between the simulation components are realised. Additionally the extracted simulation can be immediately used in the descriptions of modular simulations with the DSL. Another approach is to model the simulations with the DSL after their manual extraction. This approach only enables the usage in the description of modular simulations in the DSL. However, no further benefits are provided in this approach. Therefore, the first approach is to prefer.

6.9.2 MSE-Developer Perspective

The MSE-developer provides the description of MSE capabilities. Also, context information is modelled to state their requirement of data and notifications of simulations In the **independent viewpoint**, the *ModularEnvironment* package is utilized. The MSE-developer can enhance the *DSL* to provide elements for MSE approaches to be realised. If the *MSE* developer wants to create a coordinator like MSE, the *Coordinator* class is used as entrypint. The MSE-developer develops the functionality of the MSE and the required context information. Also the interfaces to be called by simulations or the MSE are defined. Also the MSE-developer. The MSE-developer implements the realisation of the MSE model after the definition is finished. The MSE-developer supports the simulation architect in the assignment of the correct context information to supported information types in the **dependent viewpoint**. Also the definition how simulations have to be connected to the MSE realisation is to be defined.

6.9.3 Adaptation Developer Perspective

In the **independent viewpoint** the adaptation developer designs generic adaptation descriptions and collects them in an *AdaptationDefinitionRepository*. For example, adaptation descriptions for SI-units, naming conventions or translations. Also approaches to define the adaptation conversions are specified. The adaptation developer can also develop source code for the conversions to be imported into concrete implementations.

In the **assembly viewpoint** the a adaptation developer interacts with the simulation architect to find or define adaptation descriptions used in the assembly. The simulation architect, therefore, provides information about possible mismatches in the data of the assembled simulation. The adaptation developer then inspects available modelled repositories. The *DataMarker* models of a repository provide an entry-point for the inspection. If two *DataMarker* fit the inconsistent information, the adaptation developer searches for the *AdaptationDescription* instances they are referenced in. To fit the needed purpose of the adaptation developer, the *AdaptationDescription* is not only dependent on the *DataMarker* and structure. It also depends on the modelled *AdaptationConversion*. For example, the adaptation developer needs to convert the values of two data types and *AdaptationDescription* containing both markers exist. The *AdaptationConversion* states to transform the names name of a data instead of the representing value. Here, the structure and *DataMarker* are correct, but the *AdaptationConversion* and target do not match. Because of this problem, the adaptation developer must know the purpose of the mismatching information through the interaction with the simulation architect. If no fitting *AdaptationDescription* is available for each problem, new instances are defined in an existing or new repositories. For example, the simulation architect provides the information that one simulation uses seconds and the other minutes. The a adaptation developer inspects available *AdaptationDefinitionRepository* model instances for corresponding "second" and "minute" *DataMarker* instances. If correspondences are found, the referencing *AdaptationDescription* instance is searched. If a description expresses the relationship required in the assembly, the *AdaptationDefinitionRepository* is used by the simulation architect in the modular simulation assembly. This case shows the goal of the reusable design of adaptations in the DSL. The adaptation developer creates a new adaptation description if no existing can be found. The created description is placed in a contextually fitting *AdaptationDefinitionRepository* or a new one is created.

6.9.4 Simulation Architect Perspective

In the **independent viewpoint**, the simulation architect designs interfaces to describe assumed information transfer between potential simulations. Therefore assumptions for the simulation separation have to be made. For example, the simulation architect designs an interface to specify the information exchange between simulations to calculate response times. The assumption is made that a calculation of an overall response time includes the transfer of further response times between simulations. As first step, the simulation architect searches already existing interface repositories for the required interface. The simulation architect models an interfaces describing the response time by the use of the *AssemblyInterface* class if no corresponding interface can be found,. The simulation architect defines repositories to provide contextual relations between designed interfaces. For this purpose, instances of *InterfaceRepository* are created.

In the **dependent viewpoint**, the simulation architect is the central role for connecting all designed models of the DSL. For this purpose, the simulation architect uses the *SimulationAssembly* class of the DSL as entry-point. The simulation architect assembles the simulation. Existing simulation features, modular simulations and MSEs are used to achieve a particular purpose. The simulation architect describes the created modular simulation in the *SimulationAssembly* element. This description can be used by other simulation architects to determine if the assembled simulation is of use in other assemblies. To create an assembled simulation for an individual goal, the simulation architect selects simulation features and modular simulations according to their described information and purpose. Also, simulations are selected with the wish to provide specific data capabilities to other simulations. The components of the simulations are included in the modular simulation assembly. One or more MSEs have to be inspected for the required functionality in the to be used in the assembled simulation. The best way to determine these capabilities is to interact with the MSE-developer or inspect the semantics of each *ManagementService* instance. The component models of one or more selected MSEs are then used in the modular simulation.

After all components are specified, the simulation architect connects the information to them. The simulation architect inspects each required and available data of the selected simulations. In this process, the simulation architect determines if all information requirements can be satisfied by the selected simulations. The simulation architect has to three options if not all requirements can be satisfied. The first one is to find more or exchange already selected simulations. Another possibility is to use the mechanisms of the *DSL* to define the delegation of the required and provided information to other components. Then a new modular simulation has to be built. The simulation architect has to interact with a simulation developer if the prior described options are not applied. For example because no other simulations are available or the modular simulation is determined to be used as working simulation (i.e. no delegation of required information is possible). The simulation developer has to create the simulation with the required information or extract simulation features out of existing monolithic simulations.

When all requirements are satisfied, the simulations are encapsulated to be used in the context of the assembly. Following, the simulation architect selects the interfaces for the required and provided data. No interface model should be modified if not developed for exactly the described purpose. Thus, if no interface for certain provided and required information is available, interfaces have to be created. The simulation architect sets the *Annotation* values for each information used in the modular simulation for each simulation. The simulation architect models information to be required or provided for each simulation by the mapping to an interface with the prior annotated information. Additionally, the simulation architect interacts with the adaptation developer as described in Sec. 6.9.3. With all needed *AdaptationDescription* instances available, the simulation architect connects them to the *AdapterServices* of the destined component. Also the *DataMarker* instances used in the *AdaptationDescription* instances are connected by the simulation architect to the adaptable information.

To define the connection between required and provided interfaces of simulations, the simulation architect defines *ComponentInterfaceConnection* instances. Each provided interface is therefore connected with a required interface of another component. Also with these mappings, the simulation architect can specify which requirement or provision is delegated by the modular simulation to other simulations. Through the modelling of wirings, the simulation architect specifies connections between simulation components and coordinator components by *Simulation_MSEWiring* instances. This approach enables a more flexible connection between those components.

7 Evaluation

The approach to modularise monolithic simulations by the application of the DSL is evaluated in this chapter. This evaluation includes the applicability (i.e. completeness) of the DSL itself and also characteristics like accuracy and scalability of the created modular simulation. We use the monolithic simulation *WorkwaySim* as evaluation system. This system is described in Sec. 7.1. The evaluation design is presented in Sec. 7.2. The creation of the evaluation simulations model by application of the DSL is provided in Sec. 7.3. The created model elements are evaluated in comparison to the implementation elements of the evaluation simulation's modular version in Sec. 7.4.1. Also, the behaviour preserving capabilities of the DSL are inspected by comparing simulation results of the monolithic simulation with the modular simulation. The results are presented in Sec. 7.4.2. Another aspect to evaluate is the influence of the modularisation in regard to the execution time. The results for this aspect are presented in Sec. 7.4.3. All presented results are discussed in Sec. 7.6. Assumptions, limitations and threats to validity are presented in Sec. 7.7.

7.1 Description of the Monolithic Simulation WorkwaySim

The simulation *WorkwaySim* is used as the evaluation system and is explained in more detail in this section. This explanation provides the knowledge needed for the evaluation. The explanation includes the underlying models and the declaration of variables influencing the behaviour of the *WorkwaySim* for this purpose. Also, it is discussed why the *WorkwaySim* is usable as an evaluation system.

7.1.1 WorkwaySim Simulation Model

The simulation *WorkwaySim* simulates the life-cycle of humans and a public transport system with buses and bus stops. Two underlying models are used to achieve these simulations through interaction. The model for the public transport contains buses, bus stops and routes. A bus is represented by a name, a fixed number of seats and the current number of occupied seats. The number of occupied seats cannot exceed than the maximal number of seats. A bus also contains a route. Each route consists of route segments connecting two bus stops. Every route segment describes the average speed and the distance between two bus stops. A bus visits consecutively each bus stop on its route. The bus unloads collected passengers at each bus stop if this is their destination. Also, passengers waiting at bus stops are loaded. The waiting passengers are represented by a waiting queue in the bus stops. All waiting passengers are loaded if the sum of their number and the already transported passengers is smaller than the maximum number of seats. Not all waiting passengers are loaded if this process would exceed the maximum number of seats. The waiting passengers have to wait for another bus to be loaded in this case. We call the simulation model representing the public transport system "Bus model".

The simulation of a humans life-cycle is strongly abstracted. A human in the *WorkwaySim* has a fixed home and workplace. The humans life-cycle in *WorkwaySim* simulation consists of three superordinate actions. The first action is travelling to the person' workplace and back home. The Human is the central part of this simulation model. Therefore, we call the underlying simulation model "Human". The other two actions of the human are working and spending free time. Working and spending of free-time actions are not simulated in detail by the *WorkwaySim*. Both are representing by advancing simulation time by specific duration. In the case of work, this duration is 8 hours. Two approaches are applied to simulate the humans' way to workplace and back. One approach is for the human to walk directly to their workplace and back. The other approach is taking a bus to drive between a bus stop at home and a bus stop at their workplace. One of these approaches is set for each human at its initialisation randomly. In the approach to simulate the human to walk directly, a value is assigned randomly at the initialisation process. This value determines the duration it takes for a human to walk between the home and the workplace. The walking of this way is simulated by requesting an advance of simulation time by the predetermined duration. This simulation approach is a simple scenario with no dependence to any other simulation entity. The time a human spends away from home is calculated for the simulation approach of a direct walk between the humans workplace at home as:

$$t_{away,walking} = 2 * t_{walking} + t_{work} \tag{7.1}$$

Where $t_{away,walking}$ is the total time a human spends away from home, $t_{walking}$ is the time it takes a human to walk the way between the workplace and home. t_{work} is the time a human has to spend working. In the approach to simulate a human taking a bus, two bus stop of an available route are randomly assigned to it. One bus stop is used as the bus stop near the humans home. The other bus stop is assigned as the one near the humans' workplace. The human only drives between these two stops in the simulation. The duration a human takes to walk from its home to the bus stop at home is randomly determined. The same is done for the way between the work and the humans' bus stop at its workplace. The simulation of walking these ways is again simulated by only requesting an advance of simulation time. The requested advance in time is set by the predetermined walking durations prior mentioned. The human arrives at the defined bus stop on its way to work after the simulation time is advanced. Here, the human is enqueued in the waiting queue of the bus stop. The human describes its destination as the bus stop. This destination signals where the human is unloaded by the bus. The human enters a waiting state to wait for the transportation to the destination. Depending on the workload of a bus, it is possible that the human cannot enter the first arriving bus. The humans' free-time is reduced when it is not loaded. Thus, the free time of a human depends on the waiting time at the bus stop, the driving time on the bus and the ways from and to the bus stops. A bus picks the human up and signals when the destination is reached. The bus in the *WorkwaySim* is simulated as described before. Therefore, the approach of using a bus in the life-cycle is a more complex one. This approach includes the interaction between two separate models. The free-time of a simulated human depends on the execution of the bus model and other simulated human. The time a human spends away from home is calculated for the simulation approach of utilising a bus to cross the distance between the humans workplace at home as:

$$t_{away,driving} = t_{work} + 2 * t_{HS} + 2 * t_{WS} + t_{waiting} + t_{driving} \tag{7.2}$$

Where $t_{away,driving}$ is the time the human spends away from home when utilising a bus. t_{HS} is the time a human takes to walk between the bus stop at home and its home. Analogue, t_{WS} is the time a human spends to walk between the bus stop at the workplace and its workplace. $t_{waiting}$ is the time a human waits for the bus on his way to workplace and back. $t_{driving}$ represents the time a human drives in the bus on its way to workplace and back. Like in the scenario of directly walking to the workplace, t_{work} is the time a human spends working. Therefore, when both approaches are enabled, the away time for a human is:

$$t_{away} = t_{away,driving} \text{ or } t_{away,walking} \tag{7.3}$$

Thus, the free-time of a human is calculated by:

$$t_{free} = 24 - t_{away} \tag{7.4}$$

7.1.2 Influences on the Behaviour of WorkwaySim

The resulting behaviour of the *WorkwaySim* is influenced by attribute values of human entities and the bus entities. The *WorkwaySim* uses the *Duration* class to allow descriptions of seconds, minutes and hours to measure time. The following attributes in Human entities have stochastically assigned values and influence the behaviour of the simulation:

- *homeBusStop:BusStop* and *workBusStop:BusStop* specify the bus stop at the workplace and home for a human. These bus stops are chosen randomly from all available bus stops in the *WorkwaySim*
- *behaviour:HumanBehaviour* defines whether a human walks directly to the workplace or uses the bus. The values come from the *HumanBehaviour* enum with two labels. The enum label is assigned randomly at initialisation of a human.
- *HOME_TO_STATION:Duration* contains the duration for a human to walk between the home and the bus stop at home in minutes. (Random value between 1 and 61)
- *WORK_TO_STATION:Duration* expresses the duration for a human to walk from the workplace to the bus stop at the workplace in minutes (Random value between 1 and 61)
- *WALK_DIRECTLY:Duration* contains the duration for a human to walk directly to the workplace and back in minutes (Random value between 1 and 151)

The other entity in the *WorkwaySim* determining its behaviour is the bus. The attributes of this entity are set deterministic in the source code. The possible execution of the simulation is mainly controlled by the route a bus takes to collect humans and the available maximum number of seats. The latter attribute can influence the execution of the model. If all seats are occupied, humans are not transported and thus, proceed to wait. Because of the stochastic assignment of the values for the attributes *WORK_TO_STATION:Duration* and *HOME_TO_STATION:Duration*, the bus seats can create another execution path when many humans are simulated.

To provide a more controlled execution, we implemented the possibility to alter the non-deterministic influences on the simulation behaviour to deterministic ones. This approach allows eliminating the non-deterministic behaviour as far as possible. For this purpose, each human is based on an id. This id starts at zero and is increased for every simulated human. With this id, the humans' bus stop at home and the bus stop near the workplace is determined by sorting them into a list. Then the home bus stop is selected by the human id modulo the number of available bus stops. The bus stop near the workplace is the next bus stop on the list. All durations for a human to walk are set to 30 minutes.

7.1.3 Waiting and Driving Scheme of the Human Entity

In the *WorkwaySim*, the human model expresses the waiting for the bus. As described in Sec. 5.3.4, waiting can be expressed by two approaches. One approach is the implicit waiting where the execution flow is ended and resumed by not scheduling and rescheduling events. In this approach, the waiting is realised by not scheduling the next event in an conceptual execution flow of a model. Here, an event has to be scheduled by another event to resume the conceptual execution flow. The other approach is to employ a busy-waiting scheme. The *WorkwaySim* includes both approaches. For the first approach, the execution flow of the human ends with the arriving at a bus stop. An event is scheduled in the unloading event of the bus model when the human arrives at the destination bus stop. This event resumes the logical execution flow of the human. The other approach includes the busy waiting scheme. Here, the human uses the *collected:boolean* attribute as variable to test. Listing 7.1 shows the pseudo-code of the busy-waiting scheme applied in the *WorkwaySim*.

```
//test if a human entity is not collected
if human.isCollected() == false then
        // Create new waiting event
        waitEvent = new WaitEvent()

        //Re-schedule a new waiting event with
        //a predetermined time step "BusyWaitingStep"
        waitEvent.schedule(human, BusyWaitingStep)

        //Stop preceeding of the event
        return;
end if

//create driving event
drivingEvent = new DrivingEvent()
//schedule driving event
drivingEvent.schedule(human, 0)
```

Listing 7.1: Busywaiting Approach to wait for a bus in the WorkwaySim

The value of *BusyWaitingStep* is defined as a duration in seconds. This duration describes the time point in simulation time when this event is to be scheduled. The time point is determined by the current time plus the duration provided by *BusyWaitingStep*. When the busy-waiting approach is applied, the driving of a bus is simulated like the waiting for it. The corresponding code is shown in Listing 7.2.

```
//test if a human entity is collected
if human.isCollected() == true then
        // Create new driving event
        DrivingEvent = new DrivingEvent()

        //Re-schedule a new driving event with
        //a predetermined time step "BusyWaitingStep"
        waitEvent.schedule(human, BusyWaitingStep)

        //Stop preceeding of the event
        return;
        end if

//create event to signal arrival
arriveEvent = new arriveEvent()
//schedule driving event
arriveEvent.schedule(human, 0)
```

Listing 7.2: Driving Approach when using the waiting approach of busy-waiting in the WorkwaySim

7.1.4 Discussion of Validity of WorkwaySim for Evaluation

This thesis is used to provide an approach to create a modular simulation. Therefore, the simulation has to consist of at least two separable (sub-)models. WorkwaySim is valid for this point because it provides the two underlying models with distinct concerns. The first model is the life-cycle model and the second is the public-transport model. As described in Sec. 6, extracted simulation can have requirements to execute their model and can provide information. Through the interaction between the life-cycle model and the public transport model, this is the case in the extraction of both models to separate simulations. Therefore also the required and provided information aspects of the DSL can be evaluated. Furthermore, this dependence allows identification of the information flow between the models.

7.2 Evaluation Design

We apply the Goal Question Metric (GQM) [86] approach to evaluate the modularisation with the proposed DSL. Therefore we state three goals: in the first goal **G1**, the DSL is to be analysed regarding its completeness as a language to describe the coupling of modular simulations. In the second goal **G2** the behaviour preserving aspects of the DSL are to be analysed. The third goal **G3** entails the analysis of execution times to evaluate the scalability of the modular simulation in contrast to the monolithic version. We assume that the behaviour preservation of the modular simulation compared to the monolithic version is comparable with the accuracy of the DSL.

We formulate the first research question **RQ1** to evaluate the completeness of the DSL: *Does the DSL provide the capabilities to describe the coupling between simulation features contained in a monolithic simulation*? The capabilities and completeness of DSLs are normally evaluated by approaches, discussed by Horkoff et al. [87]. These approaches rely on the application of the DSL in empirical studies, laboratory experiments (e.g. with students) or case-studies. Also, comparative approaches with other similar DSLs are discussed. The only applicable approach for evaluation in the time-frame is to conduct a case-study. This approach allows to inspect the completeness of capabilities of the DSL. Additionally the behaviour preservation and scalability can be evaluated on a concrete example. For the purpose of evaluating the DSL in the regard of RQ1, we describe the procedure of modelling the coupling of the simulation features contained in *WorkwaySim*. This approach provides insight if the DSL is complete, so that *WorkwaySim* can be modelled as the coupling of its simulation features. Additionally, we also use **Metric 1** (**M1**) on the created model:

$$M1 = \frac{E_{LANG}}{E_{IMPL}} \tag{7.5}$$

Here, E_{LANG} describes the number of elements in the modular simulation. E_{IMPL} denotes the number of elements in the implementation of the modular simulation. The results can be used to inspect if all elements of the implementation can be modelled. Also, the overhead of elements created through the application of the DSL can be evaluated. We use the *poRTIco* RTI implementation [31] in the modular simulation as coupling approach. This implementation is an open source implementation of the HLA standard [27]. The application of *poRTIco* also allows the demonstration of the capability to model the HLA with the DSL. This ability is a goal of the DSL as stated in Sec. 6. No model-to-text transformation capability is currently provided for the DSL. Because of this missing capability, the implementation of the modular simulation described by the DSL has to be created manually.

The second research question is stated regarding the behaviour preservation of the modularisation approach (**G2**). The second research question **RQ2** is defined as follows: *Are the produced simulation results in the monolithic simulation similar to those of the modular simulation*? With this research question, the similarity in behaviours of the modularised simulation to monolithic version is evaluated. *WorkwaySim* uses its entities and their initial variables to control its simulation behaviour. Some of these attributes are assigned with stochastic techniques (e.g. using a random number generator). Therefore, the Earth Mover's Distance (EMD) [88] is used as **Metric 2** (**M2**). The EMD is confirmed by Rubner et al. [89] as the most useful metric for measuring mutual difference in an empirical comparison of distribution-based similarity metrics. The EMD compares two probability distributions and takes their shapes and locations into account. The EMD provides a distance metric specifying the effort necessary to transform one distribution into another [88]. This distance signals the difference between the two distributions. With the EMD, we compare the resulting values of the monolithic simulation and its modularised version. Also, two statistical significance tests are applied to provide further information. The first test is the two-sample Wilcoxon rank sum test (hereafter called Wilcoxon test) [90]. In the case of monolithic and modular simulations, the sample consists of multiple values for the same data (e.g. created by multiple runs or multiple entities). With this comparison, the Wilcoxon test states the probability p that the two samples come from the same distribution. The p value constitutes the **Metric 3** (**M3**). The null-hypothesis is stated that both samples come from the same distribution. The alternative hypothesis states that they do not come from the same distribution. Therefore, if the p-value is smaller than an assumed threshold (i.e. the level of significance), the alternative-hypothesis has to be accepted. The second significance test is the two-sample Kolmogorov-Smirnov test [90]. With this test, two samples can be tested if they follow the same probability distributions. The null hypothesis of the Kolmogorov-Smirnov test states that two samples follow the same distribution. Similar to the Wilcoxon test, the Kolmogorov-Smirnov test provides a probability that both samples conform to the same probability distribution. Therefore, if the p-value is smaller than the selected level of significance, the alternative hypothesis has to be accepted. This alternative hypothesis states that both samples do not originate from the same probability distribution. A level of significance of $\alpha = 0.05$ is chosen for the Wilcoxon test as well as the Kolmogorov-Smirnov test. The p-value of the Kolmogorov-Smirnov test is used as **Metric 4** (**M4**). The quality of these measures depends on the calculated samples. Even if both simulations would be equal, a small number of values can result in a significant difference because of their stochastic nature. Therefore, the metrics above are also used on a deterministic version of *WorkwaySim*. In this version, all controllable stochastic attributes are set to deterministic values and references. In an entirely deterministic simulation, both simulations produce equivalent values if they provide the same modelled behaviour.

The third research question is related to the influence of the modularisation in performance. Also it is to evaluate how well the modular simulation scales. Therefore, the third research question **RQ3** is the following: *Can the modularised simulation cope with an increasing amount of simulations to a MSE.* For this purpose, multiple simulations are coupled. The resulting execution times are measured. Therefore, we use the execution times of the simulation as **Metric 5** (**M5**)

7.3 Presentation of the DSL Model of WorkwaySim

One goal of the DSL is the property to model the capabilities of the HLA, or more precisely, a concrete RTI. As a concrete RTI, the *poRTIco* implementation is chosen. Therefore, *poRTIco* is modelled as MSE and coordinator. The *WorkwaySim* consists of the underlying Bus model and the Human model. Therefore, we call the simulation features *BusSim* and *HumanSim*. These features are modelled with the DSL in order to describe the information confined in the WorkwaySim. Also, the description of modelling the assembly of the modular simulation *WorkwaySim* is provided. The purpose of the DSL is to be used in the development of simulations. We only provide model elements used in *WorkwaySim* for evaluation. For example, in *poRTIco*, only data types used in *WorkwaySim* are described.

7.3.1 Modelling of poRTIco

The source code of *poRTIco* is analysed to provide its model. The *modularenvironment* model of the DSL is created to describe *poRTIco*. *PoRTIco* provides data types used in the communication with the coupled simulations. Also functions to be called by the simulations are realised to access the capabilities of *poRTIco*. These callable functions are defined in an class called *RTIambassador*. Also functions are provided to be called from *poRTIco* in the simulations. These functions are defined in a class called *FederateAmbassador*. For every function, *poRTIco* defines several exceptions that mark occurrences of errors in the execution of a functionality. The functions in the ambassadors are ordered by the surrounding of commentary blocks according to capabilities of *poRTIco*. *PoRTIco* conforms to the HLA and therefore realises the contextual information required of object classes with their attributes and interactions with parameters according to the OMT. To model these capabilities, the OMT specification has to be analysed as well. The implementation details of each capability is of minor importance in the DSL. However, the DSL provides capabilities to describe entities used in management services. For some capabilities, *poRTIco* defines classes for such supporting entities.

PoRTIco uses standard data types with different representations. However, they are based on Java base data types like Integer or String. Therefore the *DataTypeContainer* with the *Name* attribute set to "poRTIco_DTC" is created. The model instance is only called by its name in this text when the *Name* metamodel attribute is set. For example, when the *Name* attribute is set to "poRTIco_DTC", it is expressed as: the *DataTypeContainer* "poRTIco_DTC". *DataType* instances are created for the types supported by the RTI. *PoRTIco* represents an Integer with big endian encoding by the class *HLAInteger32BE*. This data type is modelled by a *BasicDataType* instance "HLAInteger32BE". The results of the modelling process of the data types are shown in Table 7.1. Generic data types, abstract classes and super classes are used in *poRTIco* to provide data types used in the communication with simulations. One example is the class *HLA1516eVariableArray.java*. This data type represents variable arrays through lists with generic type parameters. A representation of generics and super-type references is currently not supported by the DSL. Therefore, only the description of each concrete array types can be modelled by the specification of a *DataTypeCollection* element. Therefore, a *HLA1516eVariableArray.java* of type byte can be modelled by a *DataTypeCollection* "Byte[]". A specification of the representation of the underlying primitive types is also not provided in the DSL (e.g. little or big endian). Nevertheless, such specifics can be abstractly described by the name attribute of *BaseDataType* instances like *HLAInteger32BE*. This representation of information does however not affect the created models.

Metamodel Element	Model Element
BaseDataType	HLAboolean
BaseDataType	HLAbyte
BaseDataType	HLAfloat
BaseDataType	HLAACIIstring
BaseDataType	HLAASCIIchar
BaseDataType	HLAInteger32BE
DataTypeCollection	HLAInteger32BE[]
BaseDataType	HLAfloat64Time
BaseDataType	HLAfloat64Interval
DataTypeCollection	HandleSet
BaseDataType	HLAfloat64BE
DataTypeCollection	Byte[]
BaseDataType	Handle

Table 7.1: Comparison between the elements of the poRTIco implementation and its DSLs model relevant for WorkwaySim

The exceptions used in functions of the *RTIambassador* or *FederateAmbassador* of *poRTIco* are modelled in an *ExceptionContainer* "poRTIco_EC" instance. Every exception in *poRTIco* is contained in the package *hla.rti.1516e.exceptions*. Therefore, for each class in this package, a corresponding instance of *Exception* is modelled.

The HLA specifies contextual information provided by *ObjecClasses*, *Attributes* (DSL: *Data*), *Interactions* (DSL: *Operation*) and *Parameters*. Due to the conformance of *poRTIco* to HLA, this information has to be provided. To model the requirement of these contextual information and their values, an *AnnotationContainer* instance "poRTIco_AC" is created, in which the instances of *Annotation* are specified. All created *Annotation* instances are shown in Table 7.2. When data of simulations are declared to the RTI, it can be specified whether an attribute can be acquired, divested or both in a federation. Also federates can specify if they want to divest and acquire the attribute. Divestiture means that a federate owning an instance of a object class allows that another federate can acquire the right to modify the attribute of this instance. The specification of the acquisition defines that a federate want to modify the attributes of an instance of an object class. For this purpose, the HLA specification provides the contextual information "ownership". It can be specified to the values "acquire" or "divest". To model this definition, the *MultipleSettableAnnotation* instance "Ownership" is specified. The attribute *OnlyUsedInDefinition* of "Ownership" is set to false, because simulation features not defining the attribute can specify acquisition or divestiture for it. This allows to not only specify that federates can acquire or divest the attribute but also their explicit intention to do so. To define the values specifiable for this *Annotation* instance, two *AnnotationValue* instances "Divest" and "Acquire" are modelled. The policy when an instance of an information value has to be updated can also be chosen in HLA. Possible policies are "Static" (i.e., owner updates it when requested) or "Periodic" (i.e., update at regular time intervals). Furthermore, the policies "Conditional" (i.e., update when a condition occurs) or "NA" (i.e., information is never updated) are selectable. This contextual information is modelled by a *ExclusiveSettableAnnotation* "UpdateType" instance. Its value for *OnlyUsedInDefinition* is set to true because this policy is only specified by the simulation defining this information. The *AnnotationValue* instances "Static", "Periodic", "Conditional" and "NA" are defined to model the corresponding values. The conditions for the "Conditional" value has to be specifiable, which can be expressed with a *WritableAnnotation* instance. Therefore, a model instance of *WritableAnnotation* "UpdateCondition" is specified. The HLA defines contextual information to be semantically connected. Exemplary, when "Conditional" is specified as update type, it has to be possible to specify update conditions. This connection can be modelled by *CombinedAnnotation* instances. Here, semantically connected *Annotation* instances are modelled to belong together. For example the *CombinedAnnotation* "Update" is modelled. This *Annotation* references the *ExclusiveSettableAnnotation* "UpdateType" and the *WritableAnnotation* "UpdateCondition".

AnnotationType	Annotation Name	Annotation Values
MultipleSettableAnnotation	Sharing	Publish, Subscribe
MultipleSettableAnnotation	Ownership	Acquire, Divest
CombinedAnnotation	Update	
ExclusiveSettableAnnotation	UpdateType	Static, Periodic, Conditional, NA
WritableAnnotation	UpdateCondition	
CombinedAnnotation	Distribution	
ExclusiveSettableAnnotation	Order	Receive, Timestamp
ExclusiveSettableAnnotation	Transportation	HLAreliable, HLAbestEffort
WritableAnnotation	Dimension	

Table 7.2: The *Annotation* model elements specified for poRTIco. Each horizontal line in the table specifies contextually connected *Annotation* instances

The capabilities of a MSE are categorized by conceptional concerns and represented by *ManagementService* instances in the DSL. These services are specified by analysing the source code of *poRTIco*. The HLA federate interface specification [29] is also inspected due to the conformance of *poRTIco* to it. All model elements can be inspected in the created *WorkwaySim* model [91].

All capabilities in *poRTIco* are confined in sub-packages of the package *org.portico.lrc.services.* For example the package *org.portico.lrc.services.ownership* contains the capabilities to manage the ownership of information. A *ManagementService* "Ownership Service" is modelled to describe these capabilities. *PoRTIco* provides the class *AttributeDivest.java* to describe a message to divest one or more attributes. Also, the *FederateAmbassador.java* and *RTI-ambassador.java* classes provide functions to be called by federates or by the RTI to handle the ownership of information. The *RTIambassador.java* class contains the method *attributeOwnershipAcquisition(. . .)* enabling federates to request the acquisition of the ownership of attributes of a object instance in the federation. The federate interface specification also pre- and post-conditions are provided as well as exceptions, supplied arguments and returned arguments for all accessible functions by the RTI or the federates. Each callable service of *poRTIco* is modelled by a instance of *ManagementServiceFunction*. Therefore, we model a *ManagementServiceFunction* "attributeOwnershipAcquisition" in the "Ownership Service" instance for the representation of the "attributeOwnershipAcquisition(. . .)" function. The function does not return any data. Thus, the corresponding *ReturnType* is not assigned. Exceptions can be referenced to describe their occurrence in the execution of the corresponding function. Each exception provided by the *attributeOwnershipAcquisition(. . .)* is modelled by a reference to a corresponding *Exception* model instance defined in the "poRTIco_EC" *ExceptionContainer* instance. *PoRTIco* specifies three parameters for the *attributeOwnershipAcquisition(. . .)* method. These parameters are a handle for an object instance, a *AttributeHandleSet* to specify the attributes to acquire and a user supplied tag. These attributes are modelled by the creation of three *OperationParameter* instances "theObjectClass", desiredAttributes" and "userSuppliedTag". The *poRTIco* parameter "theObjectClass" uses a handle to specify the object class the attributes are published for. Therefore the *DataType* "Handle" is referenced. The "desiredAttributes" attribute in *poRTIco* is of the type *AttributeHandleSet.* A reference to the *DataTypeCollection* "HandleSet" in the "desiredAttributes" model element represents this parameter. The "userSuppliedTag" parameter is represented by an array of the type Byte. Therefore, the *DataTypeCollection* "Byte[]" instance is referenced. The *Annotation* "Ownership" is required by the "Ownership Service" to correctly execute its functionality. The "Ownership" *Annotation* instance is referenced by the "Ownership Service" model to provide this information in the model. *PoRTIco* additionally defines the *OwnershipManager.java* class, which is responsible for the management of the declared divestitures and acquisitions of the federates. This entity is modelled by an instance of *ManagementServiceSupportEntity* "Ownership Manager" in the "Ownership Service". Its purpose is textually described by *Purpose* attribute. All modelled management services are shown in Table 7.3 The ambassadors provided *PoRTIco* realise functions to access capabilities of management services. However, the functions are only arranged by commentary blocks in *FederateAmbassador.java* and *RTIAmbassador.java* to describe their relation to a certain management service. An exemplary excerpt of the *poRTIco* source-code of the *RTIAmbassador.java* is provided in Listing 7.3 to show this structure. The dots show that more functions are placed in the source-code.

ManagementService Instance
Federation Management
Declaration Service
Object Management
Ownership Service
Time Management
Data Distribution Management
RTI Support Service Management

Table 7.3: All modelled *ManagementService* instances in the poRTIco DSL model

```
/////////////////////////////////////
// Ownership Management Services //
/////////////////////////////////////

...

// 7.8
void attributeOwnershipAcquisition(
                        ObjectInstanceHandle theObject,
                        AttributeHandleSet desiredAttributes,
                        byte[] userSuppliedTag)
throws AttributeNotPublished, ...

...

///////////////////////////////
// Time Management Services //
///////////////////////////////
// 8.2
void enableTimeRegulation(LogicalTimeInterval theLookahead)
throws InvalidLookahead, ...

...
```

Listing 7.3: Excerpt of the poRTIco RTIAmbassador.java

Instances of *MSEServiceInterface* are created in the *poRTIco* model to explicitly model these semantic relations. The names of the modelled instances provide the information, which interface is used in a federate ambassador and which in the RTI ambassador. An example for a model instance is the *MSEServiceInterface* "OwnershipManagement Functionality_RTI_Ambassador". This instance references all *ManagementServiceFunction* corresponding to the functions in *RTIambassador.java* from the "Ownership Management Services" commentary to the "Time Management Services" commentary. Thus, the "Ownership Management Functionality_RTI_Ambassador" instance contains all *ManagementServiceFunction* instances for the ownership management in the RTIambassador.

Examples of these functions are "attributeOwnershipAcquisition", "cancelAttributeOwnershipAcquisition" or "confirmDivestiture". All *MSEServiceInterface* instances can be viewed in the model in the corresponding GitHub repository [91] and are listed in Table 7.4.

MSEServiceInterface
Federation Management Functionality_RTIAmbassador
Federation Management Functionality_FederateAmbassador
Declaration Management Functionality_RTIAmbassador
Declaration Management Functionality_FederateAmbassador
Object Management Functionality_RTIAmbassador
Object Management Functionality_FederateAmbassador
Ownership Management Service Functionality_RTAmbassador
Ownership Management Service Functionality_FederateAmbassador
Time ManagementFunctionality_RTIAmbassador
Time ManagementFunctionality_FederateAmbassador
Data Distribution Management_RTIAmbassador
RTI Support Service Functionality_RTIAmbassador
poRTIco API-specific functionality_RTIAmbassador

Table 7.4: Modelled *MSEServiceInterface* instances in the poRTIco DSL model

7.3.2 Modelling of the HumanSim

To show the capabilities of the DSL for the independent modelling of simulation features, the *HumanSim* simulation feature is modelled. A *modularsimulation* model is created.

A *DataTypeContainer* with the name "HumanSim_DTC" is defined to model the data types used in the *HumanSim*. Time units have to be defined to describe the time aspects in the *HumanSim* like the duration a human takes to walk to a bus stop. The unit-element requires a *BaseDataType* to be described. For this purpose, the *BaseDataType* "HumanDouble" is created. This data type uses the label "DOUBLE" as primitive data type to represent the time. The "HumanDouble" is used as a general representation of the *double* data type of the Java language. Therefore, no restrictions of the possible value by application of the *Range* model instance is specified. The units second, minute and hour are used in *HumanSim* to define time aspects. Thus, a *Unit* model element is created for each unit. Their reference to *BaseDataType* is set to "HumanDouble", the name to either "second", "minute" or "hour". The unit symbol is set to "s" for second, "min" for minute and "h" for hour. A *UnitTypeContainer* "Duration" is modelled to provide a collected view of these units. This container references the "second", "minute" and "hour" *Unit* elements. Besides these units, the human entity contains the Enum HumanBehaviour. Therefore, a instance of *EnumType* is modelled. The "HumanBehaviour" enum contains two *EnumLiteral* elements with the *LiteralName* values "DRIVING_BY_BUS" and "WALKING". To be updated by a bus, the *HumanSim* has to specify the *collected* attribute for a human. This attribute is based on a boolean value. Therefore, the *BaseDataType* "HumanBoolean" is defined to represent the

Boolean data type in the model. Its *initialValue* attribute is set to 0 and its *stepSize* value to 1. The *PrimitiveDataType* attribute of "HumanBoolean" is set to "BOOL". Also, a range is created with the *lowerBounds* attribute set to 0 and the *upperBounds* attribute value set to 1. The names of entities in *HumanSim* are represented by String values. Therefore, a corresponding String data type has to be modelled for the description of the name of the available and required modelled entities. This representation is realised by a *BaseDataType* "HumanString" with the *PrimitiveDataType* label "String". Table 7.5 lists all data types defined in the *HumanSim* model.

Metamodel Element	Model Element
BaseDataType	HumanInt
BaseDataType	HumanDouble
BaseDataType	HumanString
BaseDataType	HumanByte
Unit	second
Unit	minute
Unit	hour
UnitTypeContainer	Duration
DataTypeCollection	HumanByte[]
EnumType	HumanBehaviour
ObjectClassReferenceDataType	BusStop

Table 7.5: DataType model instances of the HumanSim model

An instance of *SimulationDescription* "Human_Simulation_Description" is created to describe the relevant information contained in *HumanSim*. The *SimulationTimeInformation* is created. Its contained *SimulationTimeType* value is set to "DiscreteEvent", due to the discrete event execution flow representation of *HumanSim*. The available and provided data in the simulation is modelled by the *ObjectOrientedViewSimulationData* instance. Only the active entity "Human" exists in *HumanSim*. Therefore, an *ObjectClass* "Human_HS" is created. The suffix "_HS" in names is only required for identification purposes in the model due to the textual basis of the model editor. For this purpose, suffixes of "_HS" stand for "HumanSim" and the "_BS" suffix stands for "BusSim". These suffixes are of no other use. When a better editor is provided, these suffixes can be eliminated. Please note that not all information contained in the *HumanSim* is textually described in this section to avoid redundant descriptions. The final model of the *ObjectOrientedViewSimulationData* is presented in Table 7.6.

A *EnumTypedData* "behaviour" is created to represent the attribute corresponding to the humans' behaviour. The *Data* instance "name" uses the *BaseDataType* "HumanString" to describe the name of an human entity. Similar to this design, the created *Data* "collected" is based on the data type "HumanBoolean". A human in the *HumanSim* can register at bus stops. A instance of *ObjectClassOperation* "registerHumanAtBusStop_HS" is modelled to provide notifications of this interaction to other simulations. The notification is used by a bus stop entity to provide information to other simulations that a human has registered

Metamodel Element	Model Element
ObjectClassOperation	registerHumanAtBusStop_HS
OperationParameter	HumanName_HS
OperationParameter	BusStopName_HS
ObjectClass	BusStop_HS
ObjectClass	Human_HS
Data	behaviour_HS
Data	name_HS
Data	Destination_HS
ReferencingRequiredOOEntry	referencing BusStop_HS
Data	BusStopName_HS
ReferencingRequiredOOEntry	referencing Human_HS
Data	collected_HS

Table 7.6: Model elements contained in the ObjectOrientedViewSimulationData instance of the HumanSim model

itself. Therefore, a *ObjectClass* "BusStop_HS" is created and references the "registerHumanAtBusStop_HS" instance. The *HumanSim* only uses bus stops to represent them in the simulation and to sends notifications about the registration of a human. However, the *HumanSim* itself does not define available bus stops. Therefore, the names of the bus stops are required by the *HumanSim*. A *ReferencingRequiredOOEntry* is used to represent this need. A referencing entry is used because the *ObjectClass* "BusStop" already exists due to its provision of "registerHumanAtBusStop_HS". To describe the requirement of names of bus stops, the *ReferencingRequiredOOEntry* contains the *Data* instance "BusStopName_HS", which references the *BaseDataType* "HumanString". The *HumanSim* does not specify the values for the attribute *collected* of a human because no collection-mechanism is provided. This value has to be specified and changed by another simulation. This requirement is modelled by another *ReferencingRequiredOOEntry*, which references the *Human_HS ObjectClass*. A *Data* instance "collected_HS" is created and references the "HumanBoolean" to mark the requirement of a boolean value. With the created model, the *HumanSim* simulation feature is represented to be used in the assembly in the DSL. Before *WorkwaySim* can be assembled, the *BusSim* has to be defined.

7.3.3 Modelling of BusSim

The *BusSim* simulation feature has also to be modelled in order to describe the coupling of the modular *WorkwaySim* with the DSL. This process is similar to the modelling of the *HumanSim* as described in Section 7.3.2. Only elements not occurred in the *HumanSim* model are described in depth to avoid redundant descriptions already explained in Sec 7.3.2. The *SimulationFeature* element is named *BusSim*. Furthermore, a *DataTypeContainer* "BusSim_DTC" is used to describe the data types in *BusSim*. The content of this *DataTypeContainer* instance is provided in Table 7.7 Available and required information are also described by the *ObjectOrientedViewSimulationData*. The modelling of these object

Metamodel Element	Model Element
BaseDataType	BusInt
BaseDataType	BusDouble
BaseDataType	BusString
BaseDataType	BusByte
Unit	second
Unit	minute
Unit	hour
UnitTypeContainer	Duration
DataTypeCollection	BusByte[]
EnumType	BusState

Table 7.7: DataType model instances of the BusSim model

classes is done similar to the *HumanSim* model. The content is provided in Table 7.8 The

Metamodel Element	Model Element
ObjectClassOperation	registerHumanAtBusStop_BS
OperationParameter	HumanName_BS
OperationParameter	BusStopName_BS
ObjectClass	Bus_BS
Data	BusName_BS
ObjectClass	BusStop_BS
Data	BusStopName_BS
ObjectClass	Human_BS
Data	collected
ReferencingRequiredOOEntry	referencing BusStop_BS
ReferencingRequiredOOEntry	referencing Human_BS
Data	Destination_BS
Data	HumanName_BS

Table 7.8: Model elements contained in the ObjectOrientedViewSimulationData instance in the BusSim model

Table 7.8 is similar to Table 7.6. However, due to the extraction of both models out of the same monolithic simulation (i.e. *WorkwaySim*), their models are complementary. For example, *HumanSim* describes the requirement of the name of a bus stop. *BusSim* describes the name of a bus stop as available with the *Data* instance "BusStopName_BS". The *HumanSim* specifies the attribute "collected" as required, where the *BusSim* is modelled with the *ObjectClass* "Human_BS" and the *Data* instance "collected" as available. The bus has to know the humans it transports. This information is necessary to be able to represent the humans in the *BusSim* to unload them and to modify their "collected" attribute. The requirement of the humans' name and its destination is modelled by a *ReferencingRequiredOOEntry*. This entry references "Human_BS" and specifies the *Data* instances "Destination_BS" and

"HumanName_BS". The second *ReferencingRequiredOOEntry* references "BusStop_BS". Also the *ObjectClassOperation* registerHumanAtBusStop_BS" is referenced. This model element specifies that the *BusSim* requires notifications when a human arrives at the bus stop.

7.3.4 Modelling of the Adaptations used in WorkwaySim

The simulation features *HumanSim* and *BusSim* as well as *poRTIco* have different representations of information in the modular *WorkwaySim*. An example is the transfer of data by *poRTIco* through the encoding arrays of type Byte. The simulation features use and expect Strings and Integer values. Adaptations are defined specific for *WorkwaySim* by an adaptation model to encounter these problems.

The *DefinitionRepository* "WorkwaySimAdaptationRepository" is created for this purpose. For the description of the byte-array problem, *DataMarker* instances are created with the names "byteArray", "string" and "integer". These *DataMarker* instances are modelled because the transferred information between *BusSim* and *HumanSim* are either String or Integer values. For example, the names of humans or bus stops are transferred and are represented in both simulations as strings. *poRTIco*, however, uses the data types of Bytes structured in an array for the transfer of values. The transformation between an Array of Bytes to String or Integer values are described by *TransformationalConversion* instances named "ByteArrayToStringConversion" and "ByteArrayToIntegerConversion". Because of the current limitations of the adaptation approach of the DSL, the conversion can only be described textually. An example for this description is: "Transforms an integer to an array of type byte or array of type byte to an integer conforming to poRTIco". With these model elements, the adaptation is described by an instance of the *AdaptationDescription* subclass *BaseconnectedAdaptation* named "HLAByteArrayAdaption". The type of Byte-Array is selected as the base by referencing the *DataMarker* "byteArray". Two *DerivedElement* model instances are created for this *BaseconnectedAdaptation* instance. One instance references the *DataMarker* "string" and the "ByteArrayToStringConversion". The other *DerivedElement* instance references the *DataMarker* "integer" and the *TransformationalConversion* "ByteArrayToIntegerConversion". The resulting "WorkwaySimAdaptationRepository" content is shown in Table 7.9

Metamodel Element	Model Element
DataMarker	byteArray
DataMarker	string
DataMarker	integer
TransformationalConversion	ByteArrayToStringConversion
TransformationalConversion	ByteArrayToIntegerConversion
BaseConnectedAdaptation	HLAByteArrayAdaption
DerivedElement	with DataMarker: string
DerivedElement	with DataMarker: integer

Table 7.9: Model elements to model the adaptation approach

7.3.5 Modelling of the WorkwaySim Interfaces

To define required and provided data by simulations in an assembly, interfaces are used. To model the transferred information in *WorkwaySim* the *interfacedefinition* model "WorkwaySimInterfaces" is created. The entry point is a *InterfaceRepository* instance named "WorkwaySim_Interfaces".

The transferred information in *WorkwaySim* consist of the human attributes "collected", "name" and "destination". Also the name of bus stops has to be exchanged. Additionally, the notification of a human registering at a bus stop has to be send by the *WorkwaySim* and received by the *BusSim*. To provide a fine-granular approach, multiple interfaces are created to describe these information to transfer. The *AssemblyInterface* instance *HumanAttributes* is created for the human attributes defined by the *WorkwaySim* itself. In this instance, an *InterfaceObjectClass* named "Human_Attr" is defined. This element contains the *InterfaceData* "humanName" and "destination". Another instance of *AssemblyInterface* "HumanCollected" specifies a interface for the "collected" attribute of a human. The *AssemblyInterface* instance "BusStopRegisterInteraction" contains a *InterfaceOperation* "registerHumanAtBusStop" where two parameters are used. One parameter represents the humans name and the other the bus stop the human registers on. Therefore, two instances of *InterfaceParameter* are modelled in the *InterfaceOperation* "RegisterHumanAtBusStop" element. The *InterfaceOperation* instance "registerHumanAtBusStop" is not contained in an *InterfaceObjectClass* to enable that his notification can be sent by other entities than bus stops or by the simulation features themselves. All defined *AssemblyInterface* models contained in the "WorkwaySim_Interfaces" *InterfaceRepository* instance are presented in Table 7.10

Metamodel Element	Model Element
AssemblyInterface	HumanAttributes
InterfaceObjectClass	Human_Attr
InterfaceData	humanName
InterfaceData	destination
AssemblyInterface	BusStopAttributes
InterfaceObjectClass	BusStop_Attr
InterfaceData	busStopName
AssemblyInterface	BusStopRegisterInteraction
InterfaceOperation	registerHumanAtBusStop
InterfaceParameter	humanName
InterfaceParameter	busStopName
AssemblyInterface	HumanCollected
InterfaceObjectClass	Human_Coll
InterfaceData	collected

Table 7.10: Model elements to model the abstract interfaces to describe the information of a simulation

7.3.6 Modelling of the WorkwaySim Assembly

The models of *poRTIco*, *HumanSim* and *BusSim* as well as the created interface and adaptation models are used to describe the coupling to the modular *WorkwaySim*. The *SimulationAssembly* element is modelled with the name "WorkwaySim". The first step in the modelling approach is to create components of the simulations and the coordinator. Therefore, a *AssembableComponent* instance is created for each model of *HumanSim*, *BusSim* and *poRTIco*.

7.3.6.1 Modelling of AssembableComponents and AnnotationEnhanced Information

The *SimulationFeatureComponent* instances "HumanSimComponent" and "BusComponent" are created to use the prior created defined models in the *WorkwaySim* definition. "HumanSimComponent" references the "HumanSim" *SimulationFeature* element of the created model. Respective, "BusSimComponent" references the "BusSim" *SimulationFeature* element. The created *MSEComponent* "poRTIcoComponent" references the *Coordinator* instance in the *poRTIco* model.

In both *SimulationFeatureComponent* instance, the information made available in the assembly are defined by the creation of *OOAnnotationEnhancedInformation* instances. All annotated information used in the *WorkwaySim* are the object classes "Human", "BusStop" and "Bus". For each corresponding *ObjectClass* element in the *BusSim* and *HumanSim* a *AnnotatedObjectClass* is created. Also for their confined *Data*, *Operation* and *OperationParameter* a corresponding sub-class element of *AnnotationEnhanced* is created. All of their annotation are set. All *AnnotationEnhanced* elements in the *WorkwaySim* model are presented in tabular form in this text. This presentation includes their annotations set according to the *AnnotationInterface* of *poRTIco* for their *InformationType*. Table 7.11 contains the *AnnotatedObjectClass* instances and Table 7.12 presents the *AnnotatedData* instances. Also, Table 7.13 shows the *AnnotatedOperation* elements and Table 7.14 provides the *AnnotatedParameter* instances. If multiple *AnnotationValue* instances are referenced by a *MultipleSettableAnnotation* they are marked with a "/".

In the "HumanSimComponent", the *AnnotatedObjectClass* "Human_HS" is created. The corresponding *AnnotationInterface* instance defined in the *poRTIco* is referenced to define the correct interface. In this case, the *AnnotatedObjectClass* references the *AnnotationInterface* "ObjectClass_AI". The *AnnotationInterface* of *poRTIco* for *ObjectClass* instances contains the "Sharing" annotation. For this purpose, a *MultipleSelectionAnnotationSetter* instance is defined because of the *MultipleSettableAnnotation* type of "Sharing". The information what sub-type of *Annotation* is used has to be known by the modeller. In the *MultipleSelectionAnnotationSetter*, the values of *Publish* and *Subscribe* are referenced. Due to the conformance of poRTIco to the HLA specification, this defines that the human *ObjectClass* can be published and subscribed. The used operations and data of the "Human_HS" model is also annotated for the use in the *WorkwaySim*. The "name", "destination" and "collected" attributes of human entities are used in the assembly (i.e. they are transferred to or from *BusSim*). A corresponding *AnnotatedData* entry is defined for each *Data* instance. This results in the *AnnotatedData* elements named "HumanName_HS", "Collected_HS" and "Destination_HS". Each instance references its corresponding *Data* instances in the *HumanSim* model. Each *AnnotatedData* references the *AnnotationInterface* "Data_AI". This interface contains multiple annotations. The description of the process of setting *AnnotationValue* instances for every *Data* instance in the *WorkwaySim* model according to a corresponding *AnnotationInterface* would be redundant in this text. Therefore, this process is only described for one *AnnotatedData* instance in detail. This instance is the *AnnotatedData* "HumanName_HS". All modelled instances can be viewed in Table 7.12. For the two *MultipleSettableAnnotation* "Ownership" and "Sharing", two *MultipleSelectionAnnotationSetter* instances are modelled. One instance references the *Annotation* "Ownership" and the other the *Annotation* "Sharing". The setter for "Sharing" references the *AnnotationValue* instances "Publish" and "Subscribe" to show that changes in the name or collected data can be made but also received. The setter for "ownership" references the *AnnotationValue* instances "Acquire" and "Divest". Thus, the ownership of the data can be divested and acquired by the *HumanSim* and by every other simulation feature. The *poRTIco* model specifies two *CombinedAnnotation* instances. One of those instances is "Distribution", which references the two *ExclusiveSettableAnnotation* instances "Order" and "Transportation". For each of those instances, a *ExclusiveSelectionAnnotationSetter* is created and their corresponding *Annotation* instances referenced. For "Order" the *AnnotationValue* "Timestamp" is referenced. This reference defines that updates to the *Data* "HumanName_HS" is received in a time-ordered way. For "Transportation" the *AnnotationValue* "HLAreliable" is referenced. This value specifies that *poRTIco* has to assure that the transfer of the information to a federates is successful. Also, a *WritableAnnotationSetter* is defined for the *WritableAnnotation* "Dimension". The dimension of the self defined data type "Bool" is described by this *Annotation* instance. This description allows to only receive information conforming to this data type. The "Update" annotation contains one *ExclusiveSettableAnnotation* "UpdateType" and one *WritableAnnotation* "Condition". For "UpdateType" a *ExclusiveSelectionAnnotationSetter* is specified. This setter references the *AnnotationValue* "Static". This value specifies that a change in the value of *HumanName_HS* is only propagated when the value is needed by another simulation in the assembly. For the *WritableAnnotation* "Condition" a *WritableAnnotationSetter* is defined. However, "UpdateType" does not select the *AnnotationValue* "Conditional". Therefore, the *ValueContent* in the *WritableAnnotationSetter* is left empty. *HumanSim* also exchanges information corresponding to the "BusStop" *ObjectClass*. Therefore, an *AnnotatedObjectClass* BusStop_HS is created. For the required BusStop attribute "BusStopName_HS", a *AnnotatedData* with the name "BusStopName_HS" is defined. With the "Sharing" and "Ownership" Setter as in the *AnnotatedData* in the human definition. A *AnnotatedOperation* "HumanRegistersAtBusStop_HS" is defined and a "Sharing" Setter with "Publish" and "Subscribe" defined. Also an *AnnotatedParameter* called "HumanName_BusStopRegOp_HS" and "BusStopName_BusStopRegOp_HumanNameParam_HS" is created. References the Parameter_AI *AnnotationInterface* specifies that the *AnnotationSetter* corresponding to this interface are defined.

The information used by "BusSimComponent" are defined analogue to the process of setting annotations for information in *HumanSim*. Therefore, the *AnnotationEnhanced* instances for the description of information of *BusSim* used in the *WorkwaySim* are also shown in the Tables 7.11 to 7.14.

ObjectClass	Sharing
Human_HS	Publish
BusStop_HS	Subscribe
Human_BS	Subscribe
BusStop_BS	Publish

Table 7.11: ObjectClasses of BusSim and HumanSim with their set Annotations used in WorkwaySim

ObjectClass, Data	P/S	A/D	O	Trans.	Dim	Update	Cond.
Human_BS, HumanName_BS	S	-	TS	rel	-	Static	-
Human_BS, Collected_BS	P	A	TS	rel	Bool	Static	-
Human_BS, Destination_BS	S	-	TS	rel	-	Static	-
BusStop_BS, BusStopName_BS	P	-	TS	rel	-	Static	-
Human_HS, HumanName_HS	P	-	TS	rel	-	Static	-
Human_HS, Destination_HS	P	-	TS	rel	-	Static	-
Human_HS, Collected_HS	S	D	TS	rel	Bool	Static	-
BusStop_HS, BusStopName_HS	S	-	TS	rel	-	Static	-

Table 7.12: Data of BusSim and HumanSim with their set Annotations used in WorkwaySim. Abbreviations: O = Order, TS = Timestamp, P/S = Sharing, P = Publish, S = Subscribe, Trans. = Transportation, rel = HLA reliable, Dim = Dimension, Ownership = A/D, Acquire = A, Divest = D, Update = Update Type ,Cond. = condition

Operation	Sharing
HumanRegistersAtBusStop_BS	Subscribe
HumanRegistersAtBusStop_HS	Publish

Table 7.13: Operations of BusSim and HumanSim with their set Annotations used in WorkwaySim

Operation, Parameter	Order	Transportation	Dimension
HumanRegistersAtBusStop_BS, HumanName_BusStopRegOp_BS	Timestamp	HLAreliable	-
HumanRegistersAtBusStop_BS, BusStopName_BusStopRegOp_BS	Timestamp	HLAreliable	-
HumanRegistersAtBusStop_HS, HumanName_BusStopRegOp_HS	Timestamp	HLAreliable	-
HumanRegistersAtBusStop_HS, BusStopName_BusStopRegOp_HS	Timestamp	HLAreliable	-

Table 7.14: Parameters of BusSim and HumanSim with their set Annotations used in WorkwaySim

7.3.6.2 Modelling of Mappings from Annotation Enhanced Instances to Interfaces

The *HumanSim* requires the names of bus stops and the the collected attribute values of humans. The names of bus stops is defined in the "BusStopAttributes" *AssemblyInterface* instance. Therefore a *InterfaceRequired* model element is created which references this instance. The *AnnotatedObjectClass* "BusStop_HS" is referenced together with a *InterfaceObjectClass* "BusStop_Attr". Also the *ReferencingRequiredOOEntry* of the *HumanSim* model referencing the *ObjectClass* "BusStop_HS" is assigned. Within the *RequiringObjectClassMapping*, a *DataToInterfaceMapping* model element is created. This element references the *AnnotatedData* "BusStopName_HS" and the *InterfaceData busStopName*. With this mapping, it is signalled that the *AnnotatedData* is required by the *HumanSim*. *HumanSim* provides the name and the destination attribute values of humans. A instance of *InterfaceProvided* is created to model this provision. The mapping of the instances to specify is done analogue to the process in *InterfaceRequired*. ProvidingObjectClassMapping instances are created for each *InterfaceObjectClass* in the target interface. In the case of the human attributes, the *ProvidingObjectClassMapping* contains references to the *AnnotatedObjectClass* "Human_HS" and the *InterfaceObjectClass* "Human_Attr". For each *InterfaceData* or *InterfaceOperation*, corresponding *DataToInterfaceMapping* instances or *OperationToInterfaceMapping* instances are created. In these mappings, the *AnnotationEnhanced* model instance and the mapped *Interface* instance are referenced. In the case of *WorkwaySim*, the interface "HumanAttributes" contains two *InterfaceData* entries. Therefore, two *DataToInterfaceMapping* instances are created. One instance references the *AnnotatedData* instance "Destination_HS" and the *InterfaceData* "destination". The other *DataToInterfaceMapping* contains references to the *AnnotatedData* instance "HumanName_HS" and the *InterfaceData* "humanName". This definition of *InterfaceRequired* and *InterfaceProvided* mappings is repeated for the *AssemblyInterface* instances "BusStopRegisterInteraction" (provided) and "HumanCollected" (required) in the *WorkwaySim*. The *InterfaceProvided* and *InterfaceRequired* mappings are also modelled for *BusSim* with exchanged roles (i.e. required and provided). Table 7.15 shows the complete definition for all *InterfaceProvided* mappings instances. Table 7.16 displays all *InterfaceRequired* model instances. In these tables it can be seen, that only the suffixes of the data are exchanged (i.e. WS and BS). This shows, that all required data by *HumanSim* are provided by *BusSim*. It is also visible that all information required by *BusSim* is provided by *HumanSim*.

Mapping	AnnotationEnhanced Name	Interface Element Name
Operation	HumanRegistersAtBusStop_HS	registerHumanAtBusStop
Parameter	BusStopName_BusStopRegOp_HS	busStopName
Parameter	HumanName_BusStopRegOp_HS	humanName
ObjectClass	Human_HS	Human_Attr
Data	Destination_HS	destination
Data	HumanName_HS	humanName
ObjectClass	BusStop_BS	BusStop_Attr
Data	BusStopName_BS	busStopName
ObjectClass	Human_BS	Human_Coll
Data	Collected_BS	collected

Table 7.15: Model elements contained in *InterfaceProvided* models in the HumanSimComponent and BusSimComponent. For each row with "Operation", "Parameter" or "Data" in the "Mapping" column, the corresponding model elements are [X]-ToInterfaceMapping where X is one of the information types prior mentioned. When "ObjectClass" is contained, the corresponding model element is the "ProvidingObjectClassMapping"

Mapping	AnnotationEnhanced Name	Interface Element Name
Operation	HumanRegistersAtBusStop_BS	registerHumanAtBusStop
Parameter	BusStopName_BusStopRegOp_BS	busStopName
Parameter	HumanName_BusStopRegOp_BS	humanName
ObjectClass	Human_BS	Human_Attr
Data	Destination_BS	destination
Data	HumanName_BS	humanName
ObjectClass	BusStop_HS	BusStop_Attr
Data	BusStopName_HS	busStopName
ObjectClass	Human_HS	Human_Coll
Data	Collected_HS	collected

Table 7.16: Model elements contained in *InterfaceRequired* models in the HumanSimComponent and BusSimComponent. For each row with "Operation", "Parameter" or "Data" in the "Mapping" column, the corresponding model elements are [X]-ToInterfaceMapping where X is one of the information types prior mentioned. When "ObjectClass" is contained, the corresponding model element is the RequiringObjectClassMapping"

7.3.6.3 Modelling of Connections in WorkwaySim

The *InterfaceMapping* instances described in Section 7.17 have to be connected in the DSL in order to describe the implicit information flow between the components "BusSimComponent", "HumanSimComponent" and "poRTIcoComponent".

Four instances of *RequiredProvidedInterfaceConnection* are specified for the connection between the interfaces. These instances connect the *InterfaceProvided* mappings to the *InterfaceRequired* mappings. Therefore, the implicit data flow from provisioning simulations to requiring simulations is defined. The created *RequiredProvidedInterfaceConnection* instances are presented in Table 7.17. In this table, only the underlying *AssemblyInterface* is named, because *InterfaceRequired* and *InterfaceProvided* model instances do not have a *Name* attribute. One connection is textually described as example. The *RequiredProvidedInterfaceConnection* with the name "HumanCollectedMapping" provides a connection between the *InterfaceRequired* and *InterfaceProvided* instances for the "HumanCollected" *AssemblyInterface* element. Therefore, the corresponding *InterfaceRequired* and *InterfaceProvided* model elements are referenced. The providing *SimulationFeatureComponent* "BusSimComponent" together with the requiring "HumanSimComponent" referenced as well.

Connection Name	Assembly Interface	Providing Component, Requiring Component
HumanCollectedConnection	HumanCollected	BusSimComponent, HumanSimComponent
HumanAttributeConnection	HumanAttributes	HumanSimComponent, BusSimComponent
BusStopAttributeConnection	BusStopAttributes	BusSimComponent, HumanSimComponent
BusStopRegisterConnection	BusStopRegisterInteraction	HumanSimComponent, BusSimComponent

Table 7.17: Defined model elements to connect *InterfaceRequired* elements and *InterfaceProvided* elements in the WorkwaySim assembly model

WiringConnection elements define the structure of *WorkwaySim* and the communication paths between the components. Models of components have to define *Connector* elements to be defined in *WiringConnection* instances. These connectors are in *BusSim* and *HumanSim* equal to the federate ambassadors and in *poRTIco* to the *RTIambassador*. *SimulationComponentConnector* instances are defined in the *SimulationFeatureComponent* instances "BusSimComponent" and "HumanSimComponent". These connectors are named "HumanSimFederateAmbassador" and "BusSimFederateAmbassador". The *MSEServiceInterface* instances of the *poRTIco* model which end with "_FederateAmbassador" are referenced in these connectors. These references are equivalent to defined methods in the *FederateAmbassadors* needed in of *BusSim* and *HumanSim* (e.g. *receiveInteraction(…)* or *reflectAttributeValues(…)*). A *MSEComponentConnector* instance is modelled in the *MSEComponent* element "poRTIcoComponent" to define the "RTIambassador". This connector is named "poRTIcoRTIAmbassador" and contains references to all *MSEServiceInterface* instances of the *poRTIco* model which end with "_RTI_Ambassador". These references model the functionality of the "RTIAmbassador" of *poRTIco*. *PoRTIco* realises a centralised scheme, where each simulation only communicates with the *poRTIco* RTI. The connectors are connected with *WiringConnection* instances to describe the communication paths between *poRTIco*, *HumanSim* and *BusSim*. Therefore two instances of *SimulationFeature_MSEWiring* are specified. One of these instances is named "poRTIco_HumanSimWiring". It references the *MSEComponent* instance "poRTIcoComponent" along with its *MSEComponentConnector* "poRTIcoRTIAmbassador". Also the wiring describes the simulation features as endpoints. For this purpose, the *SimulationFeatureComponent* "HumanSimComponent" and the *SimulationComponentConnector* "HumanSimFederateAmbassador" are referenced. This wiring describes the connection between the federate ambassador and the RTIambassador. The same model is realised analogue for the "BusSimComponent" model.

7.3.6.4 Modelling of Adaptation Mappings

The simulation feature *BusSim* realises the adaptation approach proposed in Sec. 6.2. The used *AdapterService* "HLAAdapter" is modelled by a *StructuralAdapter* service in the *SimulationFeatureComponent* instance "BusSimComponent". The abstract adaptation description has to be connected to the data of a modular simulation to be adapted. Therefore, an *AdapterDescriptionAttachment* element is created in the "BusSimComponent" instance. This element references the *DefinitionRepository* model "WorkwaySimAdaptationRepository", the *StructuralAdapter* instance "HLAAdapter" and the *BaseConnectedAdaptation* "HLAByteArrayAdaptation". These references models the usage of the "HLAByteArrayAdaptation" in the "HLAAdapter". Three instances of *MarkerMapping* are created to provide the mapping between the *DataMarker* instances and the data in *WorkwaySim*. The content of the *MarkerMapping* instances is shown in Table 7.18. This table shows

DataMarker Name	Adaptable Names
byteArray	BusByte[], Byte[], HumanByte[]
string	BusString
integer	BusInt

Table 7.18: References of model elements in MarkerMapping instances

that the data flow adapted by the *AdaptationDescription* is predetermined. The *BusSim* only works with its String and Integer values. The "byteArray" *DataMarker*, on the other hand, describes multiple names and therefore a sending or receiving by possibly different participants in the modular simulation. The same elements are represented in the "HumanSimComponent" because it also implements these adaptation capabilities.

7.4 Evaluation Results

The evaluation results for RQ1 to RQ3 are presented in the subsections of this section. All evaluation results are found on GitHub [91]. The results were gathered on a computer with the specifications shown in Table 7.19. The knowledge about the specification of the computer is especially relevant for RQ3 where the execution times of the monolithic and modular simulation are evaluated in regard to the scalability.

CPU Type	Intel Core i7-6700
CPU Speed	3.4 GHz
RAM Size	16 GB
RAM Type	DDR4
RAM Speed	2133 MHz
Operating System	Windows 10 64bit

Table 7.19: Simulation PC

7.4.1 Evaluation Results for RQ1 - Completeness

The evaluation results regarding the completeness of the DSL are presented in this section. Therefore a mapping between the model elements of the WorkwaySim DSL model and its implementation is presented in Table 7.20 to Table 7.25. In this mapping, the number of model elements are asserted to the corresponding implementation elements. This enables to reason about the capabilities of the DSL in comparison to the working example implementations. If model elements cannot be mapped to a implementation element, the element is denoted with "no correspondence".

Table 7.20 provides a comparison of the number of model elements of the *BusSim* simulation feature model to the implementation elements in the the *BusSim* implementation. In the model, 35 elements are created. All 22 elements in the implementation have a correspondence in the model. Therefore, 13 of the 35 model elements have no correspondence. This results in a value of 1.59 in **metric 1** for the BusSim model. The Table 7.21 compares the model elements and the implementation elements of the WorkwaySim simulation. Here, 30 model elements are used. The implementation includes 21 elements. Thus, 9 model elements have no correspondence in the source code. Thus, **metric 1** is 1.43. Table 7.22 shows the comparison between the *poRTIco* implementation elements and the elements of the *poRTIco* model. The *poRTIco* model consists of 405 created elements. The implementation contains 365 elements relevant for the coupling of *WorkwaySim*. Therefore, 40 model elements have no correspondence. This results in **metric 1** with 1.11.

There is no correspondence with the "WorkwaySimInterfaces" model to the modular *WorkwaySim* implementation. The abstract description of interfaces in the DSL is used for the reusable abstract description of the exchanged information in a modular simulation. Therefore, no realisation in the implementation can be mapped to the model elements. Because of this reason, a mapping table in is dismissed. The model of the adaptation approach is implemented in the *HumanSim* and *BusSim* to provide an implementation

Metamodel Element	Number of Instances	Java-Sourcecode-Element	Number of Code-Elements
SimulationFeature BusSim	1	BusModel.java	1
DataTypeContainer	1	No correspondence	0
UnitTypeContainer Duration	1	Duration.java	1
Unit (second, minute, hour)	3	No correspondence	0
BaseDataType	5	Data types of Java	5
DataTypeCollection (BusByte[])	1	Java "Array"	1
EnumType BusState	1	BusState Enum (Bus.java)	1
EnumLiterals	4	No correspondence	0
Object Oriented View Simulation Data	1	No correspondence	0
Bus_BS, BusStop_BS, Human_BS_B	3	Bus.java, BusStop.java, Human.java	3
Data (Bus_BS)	5	Bus.java fields	5
Data (BusStop_BS)	1	BusStop.java fields	1
Data (Human_BS)	1	Human.java fields	1
ObjectClassOperation registerHumanAt-BusStop_HS	3	"registerHumanAtBusStop()"	1
OperationParameter (registerHumanAt-BusStop_HS)	2	"registerHumanAtBusStop()" parameter	2
ReferencingRequiredOOEntry	2	No correspondence	0
Data (ReferencingRequiredOOEntry)	2	Human.java fields	2
Total Number of Model-Instances	**35**	**Total Number of Java-Sourcecode-Elements**	**22**

Table 7.20: Comparison between the elements of the implementation of the Bus Simulation and its DSLs model

Metamodel Element	Number of Instances	Java-Sourcecode-Element	Number of Code-Elements
SimulationFeature HumanSim	1	HumanModel.java	1
DataTypeContainer HumanSim_DTC	1	No correspondence	0
UnitTypeContainer Duration	1	Duration.java	1
Unit (second, minute, hour)	3	No correspondence	0
BaseDataType	5	Data types of Java	5
DataTypeCollection (WorkwayByte[])	1	Java "Array"	1
EnumType HumanBehaviour	1	HumanBehaviour Enum	1
EnumLiterals	2	No correspondence	0
ObjectClassReferenceDataType BusStop	1	No correspondence	0
Object Oriented View Simulation Data	1	No correspondence	0
BusStop_HS, Human_HS_B	3	Bus.java, BusStop.java, Human.java	3
Data (Human_BS)	4	Human.java fields	4
ObjectClassOperation registerHumanAt-BusStop_BS	1	"registerHumanAtBusStop()"	1
OperationParameter (registerHumanAt-BusStop_BS)	2	"registerHumanAtBusStop()" parameter	2
ReferencingRequiredOOEntry	2	No correspondence	0
Data (ReferencingRequiredOOEntry)	2	Human.java field, BusStop.java field	2
Total Number of Model-Instances	**30**	**Total Number of Java-Sourcecode-Elements**	**21**

Table 7.21: Comparison between the elements of the implementation of the Human Simulation and its DSLs model

Metamodel Element	Number of Instances	Java-Sourcecode-Element	Number of Code-Elements
Coordinator poRTIco	1	LRC.java	1
DataTypeContainer poRTIco_DTC	1	No correspondence	0
BaseDataType (e.g., HLAbyte)	8	org.portico.impl.hla1516e.types.encoding	8
BaseDataType (e.g., HLAfloat64Time)	2	org.portico.impl.hla1516e.types.time	2
BaseDataType Handle	1	HLA1516eHandle.java	1
DataTypeCollection	2	Java-Language	2
ExceptionContainer poRTIco_EC	1	hla.rti1516e.exceptions package	1
Exception	110	.java files in hla.rti1516e.exceptions package	110
MSEServiceInterface	13	No correspondence	0
AnnotationContainer	1	org.portico.lrc.model package	1
AnnotationInterface	4	org.portico.lrc.model package (and OMT)	4
MultipleSettableAnnotation Sharing, Ownership	2	No correspondence implicit	0
AnnotationValue publish, subscribe, divest, acquire	4	No correspondence	0
ExclusiveSettableAnnotation UpdateType, Order, Transportation	3	org.portico.lrc.model package Order.java, Transportation.java	2
AnnotationValue	8	No correspondence	0
WritableAnnotation	2	No correspondence	0
CombinedAnnotation	2	No correspondence	0
ManagementService	7	No correspondence	0
ManagementServiceFunction	232	NullFederateAmbassador.java and RTIambassador Methods	232
ManagementServiceSupportEntity	1	TimeManager.java	1
Total Number of Model-Instances	**405**	**Total Number of Java-Sourcecode-Elements**	**365**

Table 7.22: Comparison between the elements of the poRTIco implementation and its DSLs model relevant for WorkwaySim

example of the adaptation. The comparison of the model elements and the implementation elements are shown in Table 7.23. This implementation results in nine model elements in the *WorkwaySimAdaptation* model and 14 elements (seven elements in *HumanSim* and seven in *BusSim*). Therefore, only one model element has no correspondence. **Metric 1** has a value of 0.64. The model of the assembly of the *WorkwaySim* model is asserted to

Metamodel Element	Number of Instances	Java-Sourcecode-Element	Number of Code-Elements
DefinitionRepository	1	No correspondence	0
DataMarker	3	Objects in BusFederate.java and HumanFederate.java	6
TransformationalConversion	2	ByteArrayTo[Integer/Sting]Conversion.java	4
BaseConnectedAdaptation	1	HLAByteArrayAdaption.java	2
DerivedElement	2	Objects of HLAByteArrayDerivedElement.java	4
Total Number of Model-Instances	**9**	**Total Number of Java-Sourcecode-Elements**	**14**

Table 7.23: Comparison between the elements of the WorkwaySimAdaptation model and the implementation in BusSim anf HumanSim

its corresponding implementation elements. This mapping is presented in the Table 7.24. This table is continued by Table 7.25 due to the many model elements. 159 model elements are used to represent the coupling of the modular *WorkwaySim*. These model elements can be mapped to 28 elements in the implementation. This results in 131 model elements without correspondences in the source code. Therefore **metric 1** is 5.68.

Metamodel Element	Number of Instances	Java-Sourcecode-Element	Number of Code-Elements
SimulationAssembly WorkwaySim	1	No correspondence	0
SimulationFeatureComponent BusSimComponent, HumanSimComponent	2	BusFederate, HumanFederate	2
MSEComponent poRTIcoComponent	1	poRTIco.jar	1
StructuralAdapter	2	HLAAdapter.java (HumanSim, BusSim)	2
AdapterDescriptionAttachment	2	No correspondence	0
MarkerMapping	6	DataMarkerMapping objects (BusFederate.java, HumanFederate.java)	6
SimulationComponentConnector (BusSimComponent, HumanSimComponent)	2	BusFederateAmbassador.java, HumanFederateAmbassador.java	2
MSEComponentConnector	1	Rti1616eAmbassador.java	1
OOAnnotationEnhancedInformation	2	No correspondence	0
AnnotatedObjectClass	4	No correspondence	0
AnnotatedData	8	No correspondence	0
AnnotatedOperation	2	No correspondence	0
AnnotatedParameters	4	No correspondence	0
MultipleSelectionAnnotationSetter	20	No correspondence	0
ExclusiveSelectionAnnotationSetter	32	No correspondence	0
WritableAnnotation	20	No correspondence	0
CombinedAnnotationSetter	20	No correspondence	0
InterfaceProvided	4	No correspondence	0
InterfaceRequired	4	No correspondence	0

Table 7.24: A comparison between the elements of the WorkwaySim implementation and its modular simulation assembly model

Metamodel Element	Number of Instances	Java-Sourcecode-Element	Number of Code-Elements
RequiringObjectClassMapping	2	*subscribeObjectClassAttributes* (BusFederate.java, HumanFederate.java)	2
DataToInterfaceMapping (InterfaceRequired)	3	Attribute handles in *subscribeObjectClassAttributes()* (BusFederate.java, HumanFederate.java)	3
ProvidingObjectClassMapping	2	*publishObjectClassAttributes()* (BusFederate.java, HumanFederate.java)	2
DataToInterfaceMapping (InterfaceProvided)	3	Attribute handles in *publishObjectClassAttributes()* (BusFederate.java, HumanFederate.java)	3
OperationToInterfaceMapping (InterfaceRequired)	1	*subscribeInteractionClass()* (BusFederate.java)	1
ParameterToInterfaceMapping (InterfaceRequired)	2	No correspondence	0
OperationToInterfaceMapping (InterfaceProvided)	1	*publishInteractionClass()* (HumanFeder-ate.java)	1
ParameterToInterfaceMapping (InterfaceProvided)	2	No correspondence	0
RequiredProvidedInterfaceConnection	4	No correspondence	0
SimulationFeature_MSEWiring	2	*rtiamb.connect()* BusFederate.java , HumanFederate.java	2
Total Number of Model-Instances	**159**	**Total Number of Java-Sourcecode-Elements**	**28**

Table 7.25: Continuation of Table 7.24: comparison between the elements of the WorkwaySim implementation and its modular simulation assembly model - continued

7.4.2 Evaluation Results for RQ2 - Accuracy

Values of the life-cycle of humans are created to evaluate the **RQ2**. The human entities in the simulation are directly influenced by the bus and their daily routine. The humans indirectly influence themselves if they provide a number greater than the maximum capacity by the bus. Therefore, values of the human entities can provide insight into the simulations behaviour. As described in section 7.1, in *HumanSim*, the way to the workplace and back home of a human is of the primary interest. Therefore, values of t_{away}, $t_{waiting}$ and $t_{driving}$ are collected to resemble this way to the workplace and back. These values are subsumed by the names *AwayTimes* (t_{away}), *WaitingTimes* ($t_{waiting}$) and *DrivingTimes* $t_{driving}$ for a number of simulated humans. Therefore, the contained values are provided in simulation time. The values in *WaitingTimes* is dependent on the bus and its load due to the possible overloading scenarios.

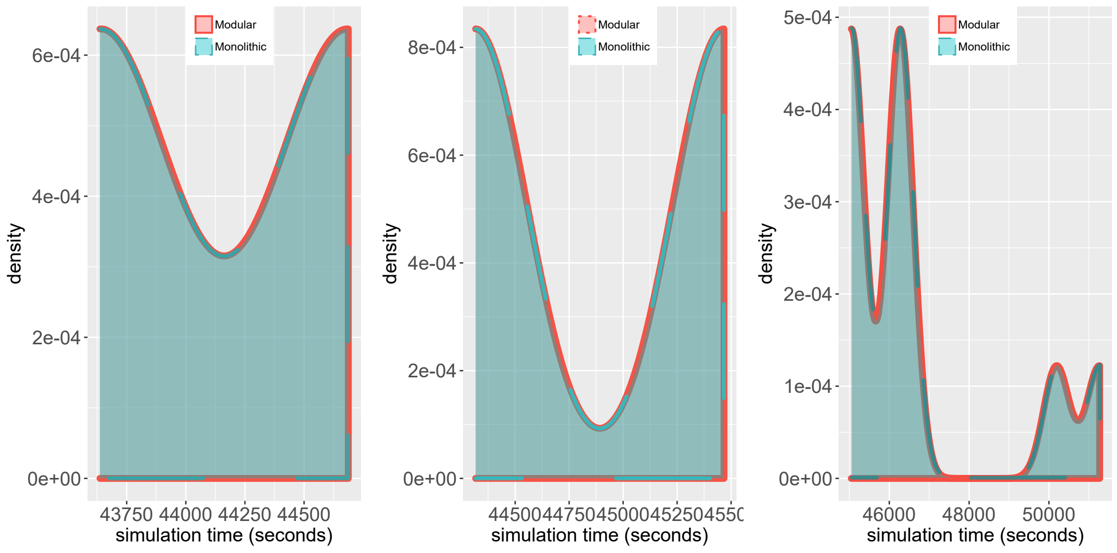
Due to the stochastic nature of the behaviour controlling variables in human entities, it can be challenging to provide a statement about the equivalence of an approach. All controllable stochastic dependent variables are altered to be deterministic to evaluate the behaviour preservation of the modular simulation. This alteration allows discovering of influences on the simulation behaviour if any exist. Additionally to use the deterministic values described in Sec. 7.1 no human will walk directly to their workplace due to the independence of this approach. Therefore, only the interaction between the both simulation feature influence the results.

M2 to **M4** are shown in Table 7.26 for the collected values of $t_{away,driving}$ for 10, 50 and 100 simulation humans. Only the metrics for $T_{away,driving}$ are presented in the table due to its dependence on $t_{waiting}$ and $t_{driving}$. In a deterministic simulation, a change in the last two value types shows immediate results in a deviation of the values in $t_{away,driving}$. This deviation can be discovered by a change in the EMD or in possible changes in p-values of the statistic tests. Fig. 7.1 shows the distributions of the *AwayTimes* values of the

Number Humans	EMD (M2)	Wilcoxon p (M3)	Kolmogorov-Smirnov p (M4)
10	0	1	1
50	0	1	1
100	0.0964	0.949	1

Table 7.26: Deterministic WorkwaySim AwayTimes Results

monolithic and the modular simulations for 10, 50 and 100 simulated humans. The x-axis represents the *AwayTimes* values of humans in seconds. The y-axis illustrates the density of the *AwayTimes* values for the corresponding quantity of simulated humans. The EMD and both p-values are at their maximum with 10 and 50 simulated humans. However, deviations of the maximum can be seen in the simulation of 100 humans. To analyse the deviation with 100 simulated humans, Table 7.27 presents the metrics for the collected values for *AwayTimes*, *DrivingTimes* and *WaitingTimes*. In this table, it can be seen that metric 1 and 2 differ where metric 3 is still at 1.00. In the *WaitingTimes*, EMD is 0.145 and Wilcoxon p-value is 0.949. The *DrivingTimes* result of EMD is 0.05 and the Wilcoxon



(a) 10 Humans (b) 50 Humans (c) 100 Humans

Figure 7.1: Figures of Deterministic Simulation Results for AwayTimes values

p-value is 0.991. In the overall *AwayTimes*, the Wilcoxon p-value is 0.949 and the EMD is 0.0964. The sums of the three value types are provided in Table 7.28 together with their differences. The differences are calculated with by subtracting the modular times by the monolithic times. It is visible, that the differences are multiples of five.

Value Type	EMD (M2)	Wilcoxon p (M3)	Kolmogorov-Smirnov p (M4)
AwayTimes	0.0964	0.949	1
WaitingTimes	0.145	0.949	1
DrivingTimes	0.05	0.991	1

Table 7.27: Simulation results AwayTimes, WaitingTimes and DrivingTimes for 100 simulated humans in the Deterministic WorkwaySim

Value Type	Modular Duration (Sum)	Monolithic Duration (Sum)	Difference
AwayTimes	4667390	4667400	-10
WaitingTimes	545185	545200	-15
DrivingTimes	522205	522200	5

Table 7.28: A comparison between the resulting durations in the monolithic WorkwaySim and Modular WorkwaySim

More data is gathered to see a trend in the evaluation when applying the non-deterministic attributes of humans in *WorkwaySim*. The number of simulated humans is increased step-wise to gather more data. However, technical limitations were experienced when simulating more than 100 humans on the evaluation machine. To be able to gather more than 100 values of humans, it was deemed useful to collect information of multiple runs and use them as one data set. The metrics for each value types are provided in separate tables. The values of *AwayTimes* are now t_{away} and are presented in Table 7.29. The metrics for *WaitingTimes* are shown in Table 7.30 and for *DrivingTimes* in Table 7.31. Fig. 7.2 shows the density distributions of the non-deterministic runs of the monolithic simulation and the modular version for the *AwayTimes*. The distribution differ from the ones shown in Fig. 7.1 because of the non-deterministic variables.

Number Humans	EMD (M2)	Wilcoxon p (M3)	Kolmogorov-Smirnov p (M4)
10	3554.36	0.082	0.164
50	182.14	0.629	0.964
100	437.67	0.543	0.699
300	201.70	0.523	0.847
500	349.24	0.329	0.413
100	103.04 0	0.992	0.954

Table 7.29: Evaluation Results for the AwayTimes in the non-deterministic WorkwaySim version

Number Humans	EMD (M2)	Wilcoxon p (M3)	Kolmogorov-Smirnov p (M4)
10	994.00	0.322	0.759
50	528.05	0.14	0.544
100	312.66	0.269	0.581
300	47.89	0.73	0.97
500	152.83	0.359	0.46
1000	46.84	0.993	0.936

Table 7.30: Evaluation Results for the WaitingTimes in the non-deterministic WorkwaySim version

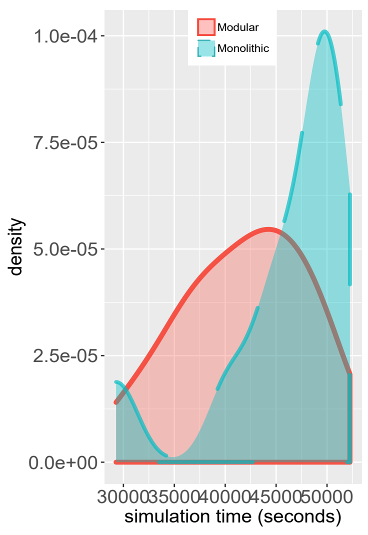
At ten humans, the Wilcoxon p-value is at 0.082 and the Kolmogorov-Smirnov p-value at 0.164. The EMD distance is at 3554.36 between both simulations. Fig. 7.2 (a) shows the dissimilarities in the results and distributions of both simulations. For 50 humans, the EMD is 182.14 and the Wilcoxon p-value 0.629. The Kolmogorov-Smirnov p-value is 0.964. The two distributions depicted in Fig. 7.2 (b) show a trend to similarities. However, differences are visible. For 100 humans, the EMD is 437.67 and the Wilcoxon p-value 0.543. The Kolmogorov-Smirnov p-value is 0.699. The two distributions provided in Fig. 7.2 (c)

Number Humans	EMD (M2)	Wilcoxon p (M3)	Kolmogorov-Smirnov p (M4)
10	425.17	0.319	0.759
50	554.74	0.437	0.544
100	475.51	0.217	0.281
300	94.89	0.796	0.249
500	94.89	0.726	0.863
100	6.37	0.945	0.759

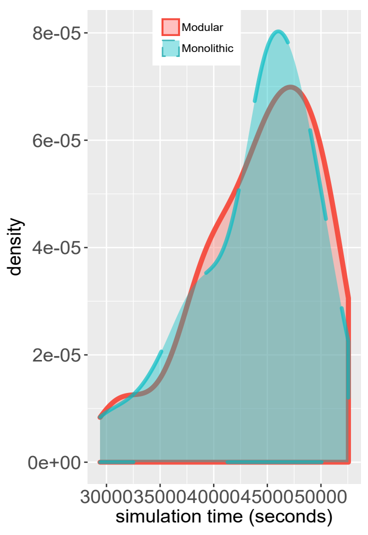
Table 7.31: Evaluation Results for the DrivingTimes in the non-deterministic WorkwaySim version

show the trend to the increase in distance and dissimilarity. With 300 humans, the EMD is 201.70 and the Wilcoxon p-value 0.523. The Kolmogorov-Smirnov p-value is 0.847. The corresponding distributions are shown in Fig. 7.2 (d) where a clear trend to an alignment of the two distributions can be seen. At 500 humans, the EMD is 349.24 and the Wilcoxon p-value 0.329. The Kolmogorov-Smirnov p-value is 0.413. The increase in non-alignment is also visible in Fig. 7.2 (e). For 1000 humans, the EMD is 103.04 and the Wilcoxon p-value 0.992. The Kolmogorov-Smirnov p-value is 0.954. In Fig. 7.2 (f) shows the trend also seen in the metrics 2 to 4. Similarities between the forms of the monolithic simulations distributions and the one of the modular version can be seen.

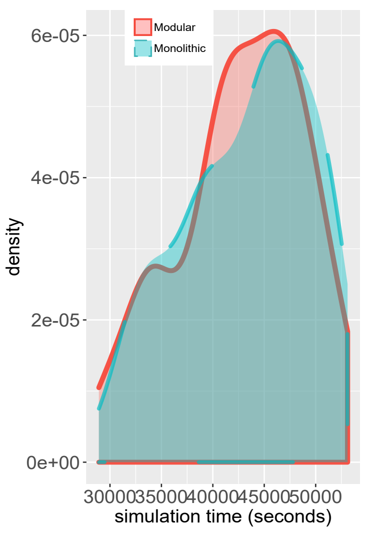
Fig. 7.3 depicts the EMD values of the *AwayTimes*, *WaitingTimes* and the *DrivingTimes* for further inspection of the dependencies of the EMD. On the x-axis, the numbers of simulated humans are represented. On the y-axis, the EMD is marked.



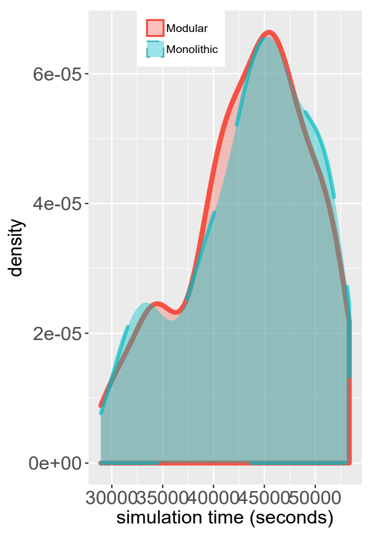
(a) 10 Humans



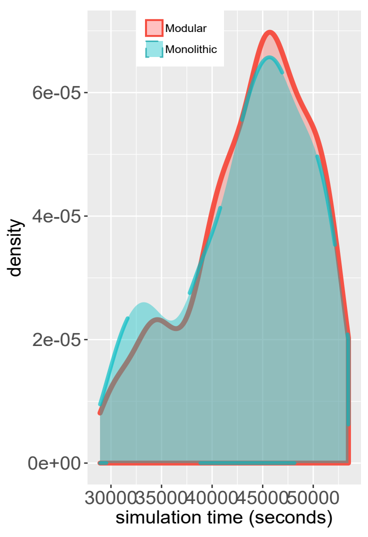
(b) 50 Humans



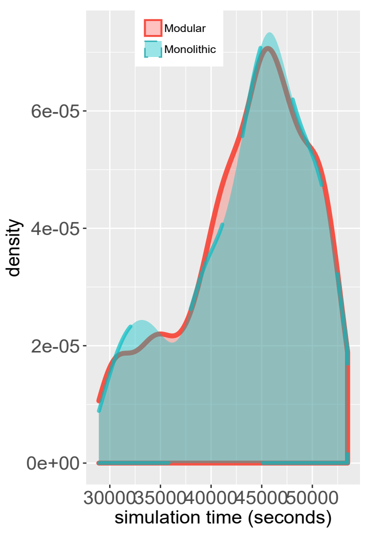
(c) 100 Humans



(d) 300 Humans



(e) 500 Humans



(f) 1000 Humans

Figure 7.2: Deterministic Simulation Results

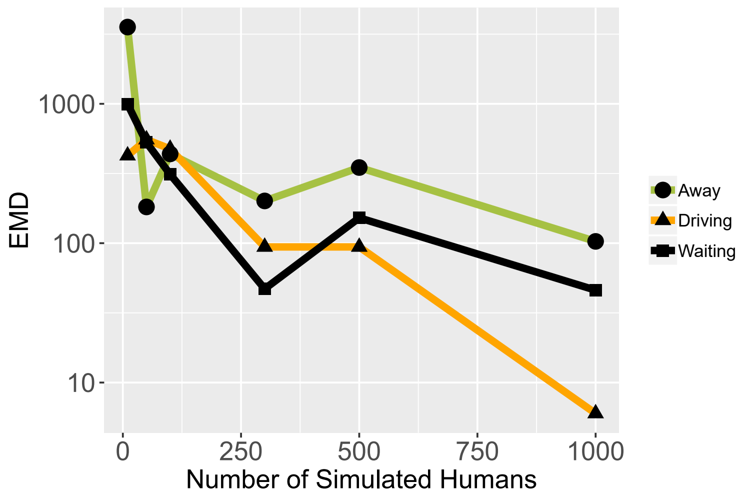


Figure 7.3: EMD Values for non-deterministic WorkwaySim Humans sorted by AwayTimes, DrivingTimes and WaitingTimes

7.4.3 Evaluation Results of RQ3 - Scalability

Another aspect to evaluate possible drawbacks or benefits of modularisation are the execution times of the monolithic simulation and modular simulation. The maximum number of simulation features to be run with *poRTIco* is limited by the performance of the used computer. On the used simulation PC, the number of maximum simulated features is between 128 and 140 at once. The execution times of the modular and monolithic *WorkwaySim* are depicted in Figure 7.4. The x-axis of the figure shows the number of simulated humans. The y-axis shows the execution times of the simulation. The figure shows the execution times for 10 to 100 simulated humans. It is of importance to notice the logarithmic scale of the y-axis. The difference between the execution times of the

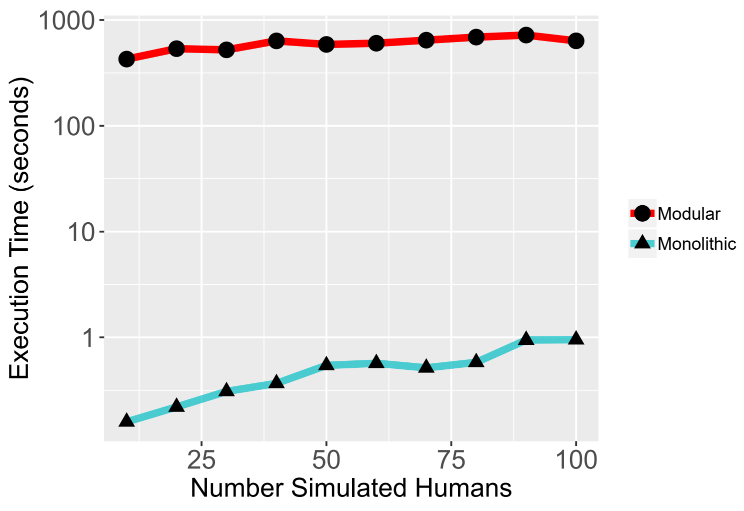


Figure 7.4: Execution Times of the Monolithic and Modular WorkwaySim

monolithic simulation and the modular simulation is visible. The logarithmic scale induces the impression that the monolithic simulations execution time increases in great amounts where the modular execution times are nearly the same. However, the execution times of the monolithic simulation range between 0.16 seconds and 0.95 seconds. The execution times of the modular simulation ranges between 427.00 seconds and 720,19 seconds.

7.5 Discussion of Evaluation-Relevant Design Decisions of WorkwaySim

Design and implementation decisions of the modular *WorkwaySim* which are relevant for the evaluation are discussed in this section. One decision is the application of the waiting approach in the monolithic *WorkwaySim* and its modular version. Also the implementation of the simulation features of *WorkwaySim* are discussed. This discussion also provides insight about the positioning of the location in the source code, where the points of times are taken to calculate the execution times.

7.5.1 Implementation of the Waiting Scheme

The waiting scheme employed in *WorkwaySim* has to be equal in the monolithic simulation and its modular version to evaluate the behaviour preservation of the approach. This equality allows a more exact evaluation of the behaviour of both simulations. The modular simulation employs the busy-waiting scheme. This application is motivated by implementation problems for resuming an control flow in interaction with *poRTIco*. Even when a simulation feature schedules events for the human, it has to test for incoming events by *poRTIco*. This testing has to be done by use a time lookahead value. However, if this lookahead is chosen to great, advancements in time can be invalid. If it is selected to small, no changes can be found. Therefore, time has to be advanced to scan the time-line. Because of this problem, the busy-waiting scheme is employed. Due to scheduling problems with *poRTIco*, the busy-waiting time advance duration of five seconds was determined. Scheduling problems were experienced with times below this duration. To provide equal results, also the monolithic WorkwaySim employs this time scheme with the same time advance.

7.5.2 Implementation of Multiple HumanSim Features

Another implementation specific decision is the simulation of each human. In modular WorkwaySim, each human and bus is represented by its own model with its own simulation engine and its own time line. This time line is then synchronised with the poRTIco time line. Multi-threading is used to simulate multiple humans in one application. For each human, a simulation is created and connected to *poRTIco*. This allows to spawn multiple features with one application. To realise the correct collection of values used in the evaluation, each created *HumanSim* simulation feature contains an *ID*. A "master" simulation feature is determined by the *ID* zero. This simulation feature waits for all *HumanSim* simulation features to be finished and then gathers and writes the collected values to files.

Technical limitations were experienced in the interaction with multi-threading approach and *poRTIco*. *PoRTIco* specifies that each joining federate has to respond within 5 seconds to a call-back from the RTI. In the multi-threading approach, this limitation creates problems when many threads are spawned to simulate multiple humans. When 100 humans have to be simulated, it is possible that some threads cannot respond within 5 seconds. *poRTIco* throws an exception when the response to the callback does not arrive within this time-window. Therefore, the *HumanSim* program spawns threads with a pre-set sleeping-period in between (e.g. four humans and then 30 seconds sleeping duration) to avoid this problem. This enables the spawned threads to answer in the time frame of 5 seconds. Contrary to this design, all entities are created in the same model and are scheduled to the same time-line in the monolithic simulation.

7.5.3 Calculation of Execution Time in WorkwaySim

In the modular and monolithic of *WorkwaySim*, execution times are calculated by collecting two points in time t_{being} and t_{end}. Here, t_{begin} is a point in time to signal the begin of the simulation execution. t_{end} is a point in time denoting the end of the simulation execution. This collection is realised by the assignment of *Double* values with the Java-Method *System.nanoTime()*. The execution time is then calculated with $t_{end} - t_{begin}$. The collection these time points are not implemented in the same locations in the source code of the monolithic *WorkwaySim* and its modular version due to the multi-threading approach. The time-point t_{begin} in the monolithic simulation is the immediate entry in the simulation execution itself. Therefore, the creation of all entities (i.e. bus, bus stops and humans) are included. t_{end} is collected before the writing of evaluation data to files after the end of simulation time. In the modular *WorkwaySim*, t_{begin} is not located after the immediate start of each simulation. The execution time is calculated by use of a master *HumanSim* simulation feature (i.e. with ID zero). The entity synchronisation between *BusSim* and *HumanSim* simulation features is left out of the execution time. All federates have to wait on a synchronisation point. If the execution time would be calculated before this point of time, the thread spawn overhead would also be included. Therefore, t_{begin} is collected before the master *HumanSim* simulation feature starts the simulation of its human. This allows obtaining the simulation time without setup and waiting mechanisms for threads not ready to execute the simulation model. t_{begin} is taken when all simulation features have ended their simulation execution. This approach allows to achieve a common ending point. This synchronisation represents the joined exit of simulation as in the monolithic version.

7.6 Discussion of Results

With the description of the model, it is shown that is possible to describe the *WorkwaySim* in a modularised form with the DSL. The created models with the DSL along with the evaluation results can be found on GitHub [91].

The model shows limitations of modelling capabilities. The DSL does not support generics type parameters and inheritance as used in the programming language Java. This drawback is already mentioned in section 7.3.1 where it is not possible to describe the data type "HLAvariableArray". This drawback is also supported by Table 7.22 where no corresponding model element to this Java class can be specified. Therefore the the DSL is described as insufficient concerning these capabilities. However, a workaround for this problem is to specify these capabilities as model elements of type (e.g. "Byte[]", "Int[]" or "Second[]").

Furthermore, concerning the completeness of the DSL (**RQ1**), Sec. 7.4.1 provides a mapping between the model elements and the Java implementation elements. The provided Tables 7.20 to 7.25 show that, for most model elements correspondences in the code exists. One reason for missing correspondences is that some model elements are for reuse and structuring purposes of the model. Examples for such elements are the *DataTypeContainer*, *ObjectOrientedViewSimulationData* and *ReferencingRequiredOOEntry* model instances in Table 7.21 and Table 7.20. Another reason for the missing correspondences are that elements in the implementation are seen as one entity. For example the *Duration.java* in *BusSim* with its representation of time units is counted as one element. In the DSL, the *UnitTypeContainer* duration is counted as well as all three *Unit* model instances. The same can be shown for the *EnumType* "HumanBehaviour" with the two *EnumLiteral* instances (counted as 3). This enum is also present in the *HumanSim* implementation. However, these literals are not counted in the implementation as elements. Additionally, model elements of the DSL can express concepts only implicitly implemented. For example the *Annotation* and *AnnotationValue* instances of the *poRTIco* model represent the possible columns of the HLA OMT for object class, attribute, parameter and interaction. In *poRTIco*, these columns are not represented as separate classes except *Order.java* and *Transportation.java*. Therefore, a greater difference in the between model elements and implementation elements exists. The adaptation approach described in Sec. 6.2 is also applied in the implementation. A *Metric 1* of 0.64 shows, that fewer model elements than implementation elements are used. This result supports the intend for the reuse of the abstract adaptation description. The one model element without correspondence can be explained by the structural characteristics of *DefinitionRepository*. The working application of the approach shows that the idea of data adaptation can be applied. However, due to the very limited application in the *WorkwaySim* example, a concrete statement whether the presented adaptation approach is applicable in more complex scenarios cannot be provided. Thus, the adaptation approach has to be further researched and applied to more and complex simulations and problems. The DSL metamodel elements of *InterfaceDefinition* lack any correspondence to the modular *WorkwaySim* implementation. This, however, is of no surprise to the author. The *InterfaceDefinition* metamodel elements are designed as reusable model descriptions for the definition of information transferred in the simulation. Therefore, no correspondences in the source code can be found. Table 7.24 also shows a considerable difference between the number of model elements and the implementation elements. One significant difference is related to the annotation of the information used in the *WorkwaySim* model. This annotation is equal to the provision of SOMs and FOMs in the HLA. The FOM *WorkwaySimFOM.xml* of the *WorkwaySim* can be found in both implementations of the simulation features *HumanSim* and *BusSim* in the GitHub repository [91]. Because the content of the *WorkwaySimFOM.xml* is not provided directly in the implementation code, no correspondence can be found. Because of the approach of not counting elements outside the source code creates a difference of 112 elements in this particular case. Another difference is provided by the application of interfaces and their connection which also can be asserted to the structural aspect of the simulation model.

With the **metric 1** of Sec. 7.4.1 the goal **G1** can be defined as reached with certain limitations. It is visible that the description of modular simulations can be achieved and implementation elements can be mapped to the model elements. The results show that more model elements have to be provided than implementation elements. This problem also results from the structural description of the DSL and the explicit description of the information provided not in the source code (e.g. the context information in FOMs and SOMs). It can also be declared that the DSLs goal to model the HLA as stated in Sec. 6 as reached. This is shown by the successful application of the DSL to *poRTIco*.

For the discussion of goal **G2**, the results provided in Sec. 7.4.2 are inspected. The accuracy in the context of this work describes if the modularised simulations exhibit the same behaviour as the monolithic version. When viewing the deterministic results, with 10 and 50 simulated humans, the p-value of Wilcoxon as well as of Kolmogorov–Smirnov are found to be 1.00. Therefore, the Wilcoxon-test expresses a 100% probability that the values of the analysed $AwayTimes$ come from the same distribution. The Kolmogorov-Smirnov-test also expresses a 100% probability that both calculated distributions are based on the same underlying distribution. The *EMD* shows a value of 0. This value implies that no transformation has to be done to transform one distribution into the other. Therefore, for ten and 50 simulated humans, it can be said that the distributions are identical. This is also made visible by the overlay of the distributions shown in Fig. 7.1. Here, both curves are directly overlapping. At 100 simulated humans, deviation of the optimal values are detected in the p-value of the Wilcox-test and the EMD. Because the walking durations for all humans are the same, problems in the interaction between *BusSim* and *HumanSim* have to exist. When inspecting Fig. 7.1 for 100 humans, a different form of the distribution is visible compared to ten and 50 humans. This different form can be explained by the quantity of the total seats of the bus, which is set to 40. In the deterministic case the bus cannot be overfilled when ten or 50 humans are simulation. The impossibility to overfill a bus is because all humans arrive at one bus stop and exit at the next. Furthermore all humans are separated into two groups and arrive at exactly the same time. Therefore 5 or 25 humans arrive at bus stop one and exit at bus stop two. Then the next 5 or 25 humans enter the bus and exit at bus stop three. Therefore, every human can be picked up after arrived at the bus stop. Also, every human arrives exactly at the same point in time. However, the bus can be overloaded when simulating 100 humans. In the deterministic scenario, 50 humans arrive at the same point of simulation time at the bus stop. Only 40 humans can be loaded into the bus, and therefore the humans have to wait for one full circle to be transported. Thus, the second slope exists. The deviation from the optimum in **metric 2** and **metric 3** shown in Table 7.26 when simulating 100 humans is surprising. 10 and 50 humans have shown, that the simulation coupling in itself works. Therefore, it is to assume, that the deviation is a result of scheduling differences by the loading and unloading interactions together with the busy-waiting scheme. This assumption is reinforced by the differences in the *AwayTimes*, *DrivingTimes* and *WaitingTimes* of the monolithic *WorkwaySim* and modular *WorkwaySim* as shown in Table 7.28. In this table it can be seen that the deviations are a multiple of five. As described in Sec. 7.5.1 the busy-waiting step time is 5 seconds duration.

However, this assumption cannot be confirmed or rejected. Further research has to be done to find the cause of this deviation. Also, a more elaborate implementation could be designed to avoid the busy-waiting scheme altogether. Another important aspect is the inspection of the metrics and diagrams provided in Sec. 7.4.2 for the non-deterministic version of *WorkwaySim*. In the visible aspects of *AwayTimes* as shown in Fig. 7.2, a trend for both distributions to better fit together is to be seen. When inspecting the values given in Table 7.29 this trend is not constant. There is an increase in the EMD between 50 and 100 simulated humans and between 300 and 500 simulated humans. This indication is resembled by the p-values of **metric 2** and **metric 3**. A reason for this inconsistency can be found in the impacts of the random variables in the simulations. Both distributions can be influenced by these variables in a different direction and thus provide even more divergence. To explore the dependency of T_{away}, $T_{waiting}$ and $T_{driving}$, the EMD as expressed in Fig. 7.3 can be consulted. No correlation of the three EMD-curves can be seen in the first three data points of each value type. The increase of the EMD in from 50 to 100 humans is against the trend of driving and waiting. Here, the walking times and the behaviour of the human can constitute the influencing factors. However, this factor seems to reduce when inspecting the EMD values with 100, 300, 500 and 1000 humans. Here, the interaction between T_{away}, $T_{waiting}$ and $T_{driving}$ can be seen. From 100 to 300 humans, all three curves tend to decrease. As inspected in the tables, the EMD values T_{away} and $T_{driving}$ both increase while $T_{waiting}$ stays nearly constant.

With the above information, it is concluded that the second goal **G2** is fulfilled. The results in Sec. 7.4.2 show that the accuracy of the modularised simulation by the DSL approach is achieved. The deterministic results show that only a slight difference between the results can exist when simulating more humans than a bus can pick up. In this scenario problems can arise. These results can be accounted to the technical specifics of the threading approach and the communication aspects of the *poRTIco* RTI. The results show that the accuracy is obtained when enough simulation runs are executed. If only a few samples are collected, the probabilistic variables express a great impact on the results. A trend to better metric values is visible with the increasing sample size. However, this result is influenced by the assumption, that a the collection of ten times a sample size of 100 values is the same as one time a sample size of 1000 values. However, this assumption is not correct due to the overloading of the bus. When 1000 simulated humans would arrive at the same bus stop to the same time, the bus has to drive its route at least 25 times to collect all humans. This value can increase when humans already finished work in this time frame. In the applied case, a bus loads all humans within at three repetitions of the route. However, this assumption had to be made to increase the sample size due to the technical limitations of *poRTIco* and the multi-threading approach. Therefore, it has to be further researched if the same trend can be experienced when more than 100 humans are simulated at once.

For the evaluation of **G3**, the execution time of the simulation is used as provided in sec. 7.4.3. In Fig. 7.4 the large difference between the modular and monolithic simulation can be seen. For 10 to 100 simulated humans, the execution times of the monolithic simulation range between 0.15 and 0.95 seconds. On the other hand, the modular simulation ranges from 427 to 720.19 seconds. This difference is alarmingly high, but even with 100 simulation humans, no exponential increase is noticed. Because in both simulations the same busy-waiting mechanism is applied, it cannot be the influencing factor. Therefore, the difference in time has to be located with the synchronisation of federates. However, it can be assumed that not so many simulation features are applied in one federation. The simulation itself could be created to simulate 100 humans within one simulation and then synchronise the interactions of one simulation feature only with the bus stop feature. The results also indicate that the execution times do not increase exponentially with the number of simulated humans. Therefore the goal **G3** is seen as reached.

7.7 Assumptions and Limitations

Several assumptions have been made and encountered limiting factors in the frame of the evaluation.

The DSL is evaluated on the working example of *WorkwaySim*. This simulation is considered as static so that no dynamic behaviour exists in the logic of the humans or the bus. Exemplary, every human uses predetermined bus stops and every bus drives predetermined routes. Thus, no dynamic processes are evaluated. The small complexity of *WorkwaySim* is another limiting factor in this evaluation. More complex simulations could result in behaviours not encountered in the evaluation. However, the limited complexity of *WorkwaySim* allows us to control and eliminate most of the stochastic influence factors in its execution. Therefore, the behaviour preservation of the modularisation approach could be explored. Another assumption is that the modularisation approach by the DSL can be evaluated on the coupling of two simulation features (i.e. *BusSim* and *WorkwaySim*). The evaluation and the DSL is dependent on the HLA approach and its realisation *poRTIco*. Therefore, the limitations of *poRTIco* and HLA also apply to the evaluation and the DSL. The limitation of 5 seconds response time from *poRTIco* in the interaction with the multi-threading approach resulted in the limitation to a maximum of 120 humans to be simulated. This results in the problem, that repeated runs had to be started to achieve more than 120 values of human. This limitation leads to the stated assumption that ten times a sample size is of 100 simulated human is equivalent to one time a sample size of 1000 humans. The challenges with this assumption and limitation are discussed in Sec. 7.6.

7.8 Threats to Validity

The threats to validity are presented in this section. These threats are *internal validity*, *external validity*, *construct validity* and *conclusion validity*.

7.8.1 Internal Validity

Internal validity defines if our conclusion is in a causal relationship to our inspections. Therefore, an unknown third factor could influence our conclusion without our knowledge. This unknown factor is a threat to the validity of our conclusion [1].

In this thesis, the quality of the *DSL* itself is a threat to validity. Furthermore the quality of the simulation descriptions is of importance and could be a threat to validity. Other developers could produce other structures or implementations which could influence the results of the evaluation. An example is another implementation of the waiting scheme without spin-wait by another developer. Another example is a hypothetical implementation of the *Workway* model as one simulation feature with multiple internal human entities. However, this implementation would result in similar implementation and model elements. Therefore, this can be considered a minor threat. Another threat can be that the types of times (i.e. *AwayTimes*, *DrivingTimes* and *WaitingTimes*) are dependent on other factors than the discussed capacity of the bus, the number of humans and their values and the synchronisation in the modular simulation. Therefore the EMD and both applied statistical tests could provide different metrics. This is considered a major threat. However, the small scope of the *WorkwaySim* allows to control other influence factors and therefore reduces this threat.

7.8.2 External Validity

External validity describes the extend to which the findings can be generalised to other entities [92]. In the case of the evaluation, this means if the findings can be generalised to other applied simulations. Case-studies provide weaker representativeness due to their limited focus. Other coupling approaches could use other representations not describable with the provided DSL. Also, other simulations could contain simulation features and structures not representable with the provided DSL capabilities. However, case-studies provide insight in the abilities of modularisation approach and the applicability to similar cases.

7.8.3 Construct Validity

Construct validity is concerned about how far the measures that are studied represents what the researcher thinks to study [1]. In this thesis, several aspects of the presented modularisation approach with the DSL are studied. With respect to the completeness, the relation of created model elements to implementation elements with measure **M1** is meant to check if the DSL represents all implementation parts of the system. However, because not all elements are measured, only a limited insight can be obtained with this approach. Other measures can provide different insight in the completeness of the DSL. Furthermore, a threat to validity exists in the evaluation of the completeness by the description of the modelling of the limited scope of *WorkwaySim*. It is intended to inspect if the the completeness of the simulation coupling on the *WorkwaySim* can be evaluated due to the case-study characteristics. Therefore, this can be seen as a minor threat. In respect to the accuracy (i.e. behaviour preservation) aspect, it is expected to gather insight if the behaviour of the monolithic simulation and its modular version are similar. It is also assumed that similarity of behaviour is equivalent to similarity of generated data. Furthermore, with M2 to M4 it is expected to measure the similarity of the value distributions of the gathered data. This assumption is considered major threat to validity in the case it does not hold. A further threat is that the results are influenced by the implementation aspects of the HLA implementation and the realised modular *WorkwaySim*. A bad design or problems in the implementation of *poRTIco* or *WorkwaySim* can overlay the behavioural aspects of measured. This is considered a minor threat due to the similarity of deterministic results as shown in Sec. 7.4.2. Another threat to construct validity is the exploration of scalability by multiple runs of the simulation with different humans. The performance of a personal computer is not constant due to other programs using the same resources. However, both curves in Fig. 7.4.3 show similar behaviours and a constant trend. Therefore, this is consider as a minor threat.

7.8.4 Conclusion Validity

Conclusion validity describes the validity of the inference of conclusion of the metrics. Statistical metrics are used in the investigation of the behaviour preservation, which reduces subjective interpretation. However, these statistical metrics are not used on samples of unknown underlying distributions and therefore the expressiveness of these tests are limited.

8 Conclusion

In this thesis, the modularisation in the context of monolithic simulations was inspected. Modularisation allows the reuse of already existing or newly developed simulations by the ability to be composed with other simulations. This approach enables the development of new simulations by composing them out of several simulation features. Also, existing simulations can be enhanced by new capabilities through modularisation of the simulation and the coupling of new simulation features. Furthermore, monolithic simulations can be decoupled into features to reuse them in potentially different contexts. The modularisation in the context of monolithic simulations was approached in this thesis in two ways. First, the existing monolithic simulation IntBIIS was analysed to provided points of interests that can be inspected when extracting simulation features out of a monolithic simulation. This information can be used to identify important aspects in the interaction of simulation features like required or provided data. Also, challenges in the extraction of simulation features out of monolithic simulations are identified in different aspects of development.

In the second part of the approach, a DSL was provided to model the coupling of simulations to describe a modular simulation. For this purpose, the DSL provides the capabilities to model simulation features and coupling approaches independent of their application in a concrete modular simulation and thus, independent of each other. This design allows reusing the created models of simulation features with models of different coupling approaches. The DSL uses these models to describe the coupling between simulation features with a concrete coupling approach to a modular simulation. The model of a modular simulation includes structural properties such as the used simulation features and their connection through the coupling approach. The data and interactions that are required or provided by features can be described by interfaces. These interfaces can be reused in other models of modular simulations. Furthermore developer roles for the application of the DSL in the development process of a modular simulation are provided as an idea on how the development of a modular simulation with the DSL can be structured. The developer roles are designed with the goal to provide concurrent work in a development process.

A problem in the reuse of simulations developed by third-parties are incompatible couplings due to differences in data and its representations (e.g. one simulation uses a String and the other simulation an Integer to represent IDs). These incompatibilities can reduce the number of usable simulation features in a coupling and therefore hinder broad reuse. The approach of *adaptation* was proposed to mitigate such incompatibilities and is supported by the DSL. An entity called *adapter service* transforms incoming and outgoing data in this approach. The data to be transformed and the process of transformation is described by *adaptation descriptions*.

In a case-study, the proposed approach was evaluated by modularisation of the monolithic simulation *Workway*. For this purpose, a model was created with the DSL to describe a modular version of *Workway* by modelling the coupling of its simulation features. As coupling approach, the HLA implementation *poRTIco* was used. The modular version of the simulation was implemented manually according to the created model, because no model-to-text transformation was provided for the DSL. The evaluation was carried out to inspect the modularisation approach in regards to three research aspects. The first research aspect was the completeness of the DSL to describe a modular version of a monolithic simulation. The second research aspect was the accuracy to preserve the behaviour when modularising a monolithic simulation with the DSL. The third research aspect in the evaluation was the scalability of the modular approach when multiple simulation features are coupled.

The evaluation results indicated the completeness of the DSL in the application to *WorkwaySim* and the HLA implementation *poRTIco*. Furthermore, the findings suggested the accuracy of the DSL and therefore the overall preservation of the behaviour in the modular simulation. Additionally, the evaluation results have shown that the execution time of the modular *Workway* was higher than in the monolithic simulation in the current implementation. Nevertheless, it was found that the execution time had shown no exponential increase when coupling multiple simulations, and thus that the scalability is achieved. Therefore, it was concluded that the goal to develop an approach to describe the coupling of simulations to a modular simulation succeeded with the creation of the DSL. Also the application of the *adaptation* approach showed promising capabilities to mitigate incompatibilities in the implementation of the modular simulation.

One main threat to the validity of the evaluation is the limited scope of the evaluation simulation *WorkwaySim*. Therefore, further work will have to include the application of the modularisation approach to more complex and different simulations. This application also enables the further evaluation of the *adaptation* approach. Also, a laboratory experiments with multiple students to evaluate the completeness and applicability of the DSL in a bigger scope could are possible which would also reduce threats to external validity. A claim of the DSL to model hierarchical approaches could not be evaluated in the frame of this thesis, because the evaluated *WorkwaySim* did not contain hierarchical structured simulation features. Therefore, this capability has to be evaluated in future work by the application of hierarchical defined simulations. Additionally, the implementation of a model-to-text transformation for the DSL is planned. This transformation allows to generate code for automatic coupling of simulation features. With this code-generation, the coupling approach is to be used on other coupling approaches asides from HLA. For this purpose, approaches such as DIS will be analysed. Also, the extension of the DSLs' metamodel is planned to support behavioural aspects of simulation features and modular simulations. Additionally, improvements to the metamodel of the DSL regarding the representation data (e.g. data types) are possible. Furthermore, an enhancement the DSLs' capabilities to describe adaptations should be performed. An idea for the latter is to increase the capabilities to describe adaptation conversions (e.g. with a complete mathematical system).

Bibliography

[1] Per Runeson et al. *Case Study Research in Software Engineering: Guidelines and Examples*. 1st. Wiley Publishing, 2012. ISBN: 1118104358, 9781118104354.

[2] Hans van Vliet. *Software Engineering: Principles and Practice*. New York, NY, USA: John Wiley & Sons, Inc., 1993. ISBN: 0-471-93611-1.

[3] Ralf [HerausgeberIn] Reussner et al., eds. *Modeling and simulating software architectures : the Palladio approach*. Includes bibliographical references and index. Cambridge, Massachusetts: MIT Press, [2016]. ISBN: 978-0-262-03476-0.

[4] Robert Heinrich. *Aligning Business Processes and Information Systems: New Approaches to Continuous Quality Engineering*. Springer Vieweg, 2014. ISBN: 3658065176, 9783658065171.

[5] Herbert Stachowiak. *Allgemeine Modelltheorie*. Wien, New York: Springer, 1973. ISBN: 0387811060.

[6] Thomas Stahl and Markus Völter. *Model driven software development : technology, engineering, management*. Einheitssacht. im Buch fälschl. als Refactorings in großen Softwareprojekten angegeben. Chichester [u.a.]: Wiley, 2006. ISBN: 0-470-02570-0; 978-0-470-02570-3.

[7] *About the Meta Object Facility Specification Version 2.5.1*. 2016. URL: `https://www.omg.org/spec/MOF/2.5.1/` (visited on 07/21/2018).

[8] Thomas Goldschmidt, Steffen Becker, and Erik Burger. "Towards a Tool-Oriented Taxonomy of View-Based Modelling". In: *Proceedings of the Modellierung 2012*. Ed. by Elmar J. Sinz and Andy Schürr. Vol. P-201. GI-Edition – Lecture Notes in Informatics (LNI). Bamberg: Gesellschaft für Informatik e.V. (GI), Mar. 2012, pp. 59–74. ISBN: 978-3-88579-295-6.

[9] Erik Burger. "Flexible Views for View-Based Model-Driven Development". In: *Proceedings of the 18th international doctoral symposium on Components and architecture*. WCOP '13. Vancouver, British Columbia, Canada: ACM, 2013, pp. 25–30. ISBN: 978-1-4503-2125-9. DOI: `10.1145/2465498.2465501`. URL: `http://doi.acm.org/10.1145/2465498.2465501`.

[10] Averill M. Law. *Simulation modeling and analysis*. 4. ed., international ed., [Nachdr.] McGraw-Hill series in industrial engineering and management science. Boston [u.a.]: McGraw-Hill, [20]11. ISBN: 978-0-07-125519-6; 0-07-125519-2.

[11] Jerry Banks. "Introduction to Simulation". In: *Proceedings of the 31st Conference on Winter Simulation: Simulation—a Bridge to the Future - Volume 1*. WSC '99. Phoenix, Arizona, USA: ACM, 1999, pp. 7–13. ISBN: 0-7803-5780-9. DOI: `10.1145/324138.324142`. URL: `http://doi.acm.org/10.1145/324138.324142`.

[12] Andreas Tolk et al., eds. *Advances in Modeling and Simulation: Seminal Research from 50 Years of Winter Simulation Conferences.* en. Simulation Foundations, Methods and Applications. Springer International Publishing, 2017. ISBN: 978-3-319-64181-2. URL: `//www.springer.com/de/book/9783319641812` (visited on 05/18/2018).

[13] Jerry [Hrsg.] Banks, ed. *Handbook of simulation : principles, methodology, advances, applications, and practice.* A Wiley-Interscience publication. Includes index. New York: Wiley, c1998. ISBN: 0-471-13403-1; 978-0-471-13403-9.

[14] Robert Heinrich et al. "Integrating business process simulation and information system simulation for performance prediction". In: *Software & Systems Modeling* 16.1 (2017), pp. 257–277. URL: `http://link.springer.com/article/10.1007/s10270-015-0457-1`.

[15] Mikel D. Petty et al. "Software Frameworks for Model Composition". In: *Model. Simul. Eng.* 2014 (Jan. 2014), 4:4–4:4. ISSN: 1687-5591. DOI: `10.1155/2014/492737`. URL: `http://dx.doi.org/10.1155/2014/492737`.

[16] Rod [VerfasserIn] Stephens. *Beginning software engineering.* Indianapolis, IN, [2015]. URL: `http://swbplus.bsz-bw.de/bsz453328865cov.htmhttp://lib.myilibrary.com/detail.asp?id=770088%20;%20http://lib.myilibrary.com?id=770088`.

[17] Roberto Setola et al. *Managing the Complexity of Critical Infrastructures: A Modelling and Simulation Approach.* 1st. Springer Publishing Company, Incorporated, 2017. ISBN: 3319510428, 9783319510422.

[18] Robert G. Bartholet et al. *In search of the philosopher's stone: Simulation composability versus component-based software design.* Tech. rep. VIRGINIA UNIV CHARLOTTESVILLE DEPT OF COMPUTER SCIENCE, 2004. URL: `http://www.dtic.mil/docs/citations/ADA446993`.

[19] P.K. Davis et al. *Improving the Composability of Department of Defense Models and Simulations.* National Defense Research Institute. Rand, 2003. ISBN: 9780833035257.

[20] M. D. Petty and E. W. Weisel. "A composability lexicon". In: *Proceedings of the Spring 2003 Simulation Interoperability Workshop* (2003), pp. 181–187. URL: `http://www.cs.virginia.edu/%5C~%7B%7Drgb2u/03S-SIW-023.doc`.

[21] Okan Topçu et al. *Distributed Simulation - A Model Driven Engineering Approach.* Simulation Foundations, Methods and Applications. Springer, 2016. ISBN: 978-3-319-03049-4. DOI: `10.1007/978-3-319-03050-0`. URL: `https://doi.org/10.1007/978-3-319-03050-0`.

[22] Richard Briggs and John A. Tufarolo. "Toward a Family of Maturity Models for the Simulation Interconnection Problem." Paper 04S-SIW-145". In: *In Proceedings of the Spring Interoperability Workshop.* 2004.

[23] Andreas Tolk and James Muguira. *The Levels of Conceptual Interoperability Model.* Sept. 2003.

[24] C4ISR Architecture Working Group et al. "Levels of information systems interoperability (LISI)". In: *US DoD* (1998).

[25] Wenguang Wang, Andreas Tolk, and Weiping Wang. "The Levels of Conceptual Interoperability Model: Applying Systems Engineering Principles to M&S". In: *Proceedings of the 2009 Spring Simulation Multiconference*. SpringSim '09. San Diego, California: Society for Computer Simulation International, 2009, 168:1–168:9. URL: `http://dl.acm.org/citation.cfm?id=1639809.1655398`.

[26] Cláudio Gomes et al. "Co-simulation: State of the art". In: *CoRR* abs/1702.00686 (2017). arXiv: `1702.00686`. URL: `http://arxiv.org/abs/1702.00686`.

[27] Judith S. Dahmann, Richard M. Fujimoto, and Richard M. Weatherly. "The department of defense high level architecture". In: *Proceedings of the 29th conference on Winter simulation*. IEEE Computer Society, 1997, pp. 142–149. URL: `http://dl.acm.org/citation.cfm?id=268465`.

[28] DIS Steering Committee et al. "The DIS vision: A map to the future of distributed simulation". In: *Institute for Simulation and Training* (1994).

[29] "IEEE Standard for Modeling and Simulation (M amp;S) High Level Architecture (HLA)– Federate Interface Specification - Redline". In: *IEEE Std 1516.1-2010 (Revision of IEEE Std 1516.1-2000) - Redline* (Aug. 2010), pp. 1–378. DOI: `10.1109/IEEESTD.2010.5954120`.

[30] "IEEE Standard for Modeling and Simulation (M amp;S) High Level Architecture (HLA)– Object Model Template (OMT) Specification". In: *IEEE Std 1516.2-2010 (Revision of IEEE Std 1516.2-2000)* (Aug. 2010), pp. 1–110. DOI: `10.1109/IEEESTD.2010.5557731`.

[31] Tim Pokorny and Michael Fraser. *The Portico Project*. URL: `http://www.porticoproject.org/comingsoon/` (visited on 07/28/2018).

[32] "IEEE Standard for Modeling and Simulation (M amp;S) High Level Architecture (HLA)– Framework and Rules". In: *IEEE Std 1516-2010 (Revision of IEEE Std 1516-2000)* (Aug. 2010), pp. 1–38. DOI: `10.1109/IEEESTD.2010.5553440`.

[33] Paul Gustavson et al. "The Base Object Model (BOM) Primer: A Distilled Look at a Component Reuse Methodology for Simulation Interoperability". In: (Jan. 2018).

[34] Jianxing Gong et al. "Applying BOM-based simulation model components to rapidly compose simulations and simulation environments". In: *2010 International Conference on Computer Application and System Modeling (ICCASM 2010)* 14 (2010), pp. V14-249-V14-252.

[35] Y. M. Teo and C. Szabo. "CODES: An Integrated Approach to Composable Modeling and Simulation". In: *41st Annual Simulation Symposium (anss-41 2008)*. Apr. 2008, pp. 103–110. DOI: `10.1109/ANSS-41.2008.24`.

[36] Y. M. Teo and C. Szabo. "CODES: An Integrated Approach to Composable Modeling and Simulation". In: *41st Annual Simulation Symposium (anss-41 2008)*. Apr. 2008, pp. 103–110. DOI: `10.1109/ANSS-41.2008.24`.

[37] Bernard P. Zeigler. *Multifacetted modelling and discrete event simulation*. London [u.a.]: Acad. Press, 1984. ISBN: 0-12-778450-0.

[38] J. S. Lee Bernard P. Zeigler. *Theory of quantized systems: formal basis for DEVS/HLA distributed simulation environment.* 1998. DOI: 10.1117/12.319354. URL: https://doi.org/10.1117/12.319354.

[39] Alex Chung Hen Chow and Bernard P. Zeigler. "Parallel DEVS: A Parallel, Hierarchical, Modular, Modeling Formalism". In: *Proceedings of the 26th Conference on Winter Simulation.* WSC '94. Orlando, Florida, USA: Society for Computer Simulation International, 1994, pp. 716–722. ISBN: 0-7803-2109-X. URL: http://dl.acm.org/citation.cfm?id=193201.194336.

[40] M. Rohl and A. M. Uhrmacher. "Composing Simulations from XML-Specified Model Components". In: *Proceedings of the 2006 Winter Simulation Conference.* Dec. 2006, pp. 1083–1090. DOI: 10.1109/WSC.2006.323198.

[41] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming.* 2nd. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 0201745720.

[42] *James II.* URL: http://wwwmosi.informatik.uni-rostock.de/mosi/projects/cosa/james-ii (visited on 08/29/2018).

[43] J. Eker et al. "Taming heterogeneity - the Ptolemy approach". In: *Proceedings of the IEEE* 91.1 (Jan. 2003), pp. 127–144. ISSN: 0018-9219. DOI: 10.1109/JPROC.2002.805829.

[44] Claudius Ptolemaeus, ed. *System Design, Modeling, and Simulation using Ptolemy II.* Ptolemy.org, 2014. URL: http://ptolemy.org/books/Systems.

[45] András Varga and Rudolf Hornig. "An Overview of the OMNeT++ Simulation Environment". In: *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops.* Simutools '08. Marseille, France: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008, 60:1–60:10. ISBN: 978-963-9799-20-2. URL: http://dl.acm.org/citation.cfm?id=1416222.1416290.

[46] András Varga. "The OMNeT++ Discrete Event Simulation System". In: *Proceedings of the European Simulation Multiconference (ESM'2001)* (June 2001).

[47] Steve McConnell. *Code Complete, Second Edition.* Redmond, WA, USA: Microsoft Press, 2004. ISBN: 0735619670, 9780735619678.

[48] Martin Fowler. *Domain Specific Languages.* 1st. Addison-Wesley Professional, 2010. ISBN: 0321712943, 9780321712943.

[49] Arne N. Johanson and Wilhelm Hasselbring. "Hierarchical Combination of Internal and External Domain-Specific Languages for Scientific Computing". In: *Proceedings of the 2014 European Conference on Software Architecture Workshops.* ECSAW '14. Vienna, Austria: ACM, 2014, 17:1–17:8. ISBN: 978-1-4503-2778-7. DOI: 10.1145/2642803.2642820. URL: http://doi.acm.org/10.1145/2642803.2642820.

[50] P. C. Clements. "A survey of architecture description languages". In: *Proceedings of the 8th International Workshop on Software Specification and Design.* Mar. 1996, pp. 16–25. DOI: 10.1109/IWSSD.1996.501143.

[51] D. C. Luckham and J. Vera. "An event-based architecture definition language". In: *IEEE Transactions on Software Engineering* 21.9 (Sept. 1995), pp. 717–734. ISSN: 0098-5589. DOI: 10.1109/32.464548.

[52] Frederic D. McKenzie, Mikel D. Petty, and Qingwen Xu. "Usefulness of Software Architecture Description Languages for Modeling and Analysis of Federates and Federation Architectures." In: *Simulation* 80.11 (Sept. 29, 2009), pp. 559–576. URL: http://dblp.uni-trier.de/db/journals/simulation/simulation80.html#McKenziePX04.

[53] Albert Endres and Dieter Rombach. *A handbook of software and systems engineering : empirical observations, laws and theories*. 1. publ. The Fraunhofer IESE series on software engineering. Includes index. Harlow [u.a.]: Pearson, Addison Wesley, 2003. ISBN: 0-321-15420-7. URL: http://bvbr.bib-bvb.de:8991/F?func=service&doc_library=BVB01&doc_number=012855132&line_number=0001&func_code=DB_RECORDS&service_type=MEDIA.

[54] Phillip Merkle and Henss Joerg. "EVENTSIM – An Event-driven Palladio Software Architecture Simulator". In: *Palladio Days 2011. Proceedings*, pp. 15–22.

[55] Johannes Göbel et al. "The Discrete Event Simulation Framework DESMO-J: Review, Comparison To Other Frameworks And Latest Development." In: *ECMS*. 2013, pp. 100–109.

[56] Pierre L'Ecuyer, Lakhdar Meliani, and Jean Vaucher. "SSJ: SSJ: A Framework for Stochastic Simulation in Java". In: *Proceedings of the 34th Conference on Winter Simulation: Exploring New Frontiers*. WSC '02. San Diego, California: Winter Simulation Conference, 2002, pp. 234–242. ISBN: 0-7803-7615-3. URL: http://dl.acm.org/citation.cfm?id=1030453.1030488.

[57] Len Granowetter. "RTI Interoperability Issues - API Standards, Wire Standards, and RTI Bridges". en. In: (), p. 23. URL: https://www.sisostds.org/DesktopModules/Bring2mind/DMX/Download.aspx?Command=Core_Download&EntryId=24573&PortalId=0&TabId=105 (visited on 06/30/2018).

[58] Dr. Andreas Tolk. "Metamodels and mappings - ending the interoperability war". In: *04F-SIW-105, Fall Simulation Interoperability Workshop*. 2004.

[59] Shawn Parr Russell Keith-Magee. "The Next Step-Applying the Model Driven Architecture to HLA". In: 2003.

[60] David Scerri et al. "An Architecture for Modular Distributed Simulation with Agent-based Models". In: *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: Volume 1 - Volume 1*. AAMAS '10. Toronto, Canada: International Foundation for Autonomous Agents and Multiagent Systems, 2010, pp. 541–548. ISBN: 978-0-9826571-1-9. URL: http://dl.acm.org/citation.cfm?id=1838206.1838283.

[61] Paolo Bocciarelli et al. “A Model-driven Framework for Distributed Simulation of Autonomous Systems”. In: *Proceedings of the Symposium on Theory of Modeling &#38; Simulation: DEVS Integrative M&#38;S Symposium.* DEVS ’15. Alexandria, Virginia: Society for Computer Simulation International, 2015, pp. 213–220. ISBN: 978-1-5108-0105-9. URL: http://dl.acm.org/citation.cfm?id=2872965.2872994.

[62] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A practical guide to SysML : the systems modeling language.* Includes bibliographical references and index. - Description based on print version record. Waltham, MA, c2012. URL: http://lib.myilibrary.com/detail.asp?id=329367.

[63] *Pitch Technologies – Pitch pRTI – a Certified HLA RTI.* URL: http://www.pitchtechnologies.com/products/prti/ (visited on 07/21/2018).

[64] Himanshu Neema. “Large-Scale Integration of Heterogeneous Simulations”. PhD thesis. Vanderbilt University, 2018.

[65] P. Benjamin, M. Patki, and R. Mayer. “Using Ontologies for Simulation Modeling”. In: *Proceedings of the 2006 Winter Simulation Conference.* Dec. 2006, pp. 1151–1159. DOI: 10.1109/WSC.2006.323206.

[66] P. Benjamin and K. Akella. “Towards ontology-driven interoperability for simulation-based applications”. In: *Proceedings of the 2009 Winter Simulation Conference (WSC).* Dec. 2009, pp. 1375–1386. DOI: 10.1109/WSC.2009.5429286.

[67] María Gutiérrez and Horacio Leone. “Composability Model in a Distributed Simulation Environment for Supply Chain”. In: 5 (Dec. 2013), pp. 55–69.

[68] Özer Özdikiş, Umut Durak, and Halit Oğuztüzün. “Tool Support for Transformation from an OWL Ontology to an HLA Object Model”. In: *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques.* SIMUTools ’10. Torremolinos, Malaga, Spain: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2010, 55:1–55:6. ISBN: 978-963-9799-87-5. DOI: 10.4108/ICST.SIMUTOOLS2010.8678. URL: https://doi.org/10.4108/ICST.SIMUTOOLS2010.8678.

[69] Alessandro Vittorio Papadopoulos and Alberto Leva. “Automating Dynamic Decoupling in Object-Oriented Modelling and Simulation Tools”. In: *Proceedings of the 5th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools, EOOLT 2013, April 19, University of Nottingham, Nottingham, UK.* 2013, pp. 37–44. URL: http://www.ep.liu.se/ecp_article/index.en.aspx?issue=084;article=005.

[70] Jacob Fish and Wen Chen. “Modeling and simulation of piezocomposites”. In: *Computer Methods in Applied Mechanics and Engineering* 192.28 (2003). Multiscale Computational Mechanics for Materials and Structures, pp. 3211–3232. ISSN: 0045-7825. DOI: https://doi.org/10.1016/S0045-7825(03)00343-8. URL: http://www.sciencedirect.com/science/article/pii/S0045782503003438.

[71] Soo Dong Kim and Soo Ho Chang. “A systematic method to identify software components”. In: *11th Asia-Pacific Software Engineering Conference.* Nov. 2004, pp. 538–545. DOI: 10.1109/APSEC.2004.11.

[72] Misook Choi and Eunsook Cho. “Component Identification Methods Applying Method Call Types between Classes”. In: *J. Inf. Sci. Eng.* 22.2 (2006), pp. 247–267.

[73] Zhamak Dehghani. *How to break a Monolith into Microservices.* URL: https://martinfowler.com/articles/break-monolith-into-microservices.html (visited on 07/04/2018).

[74] S. Sarkar et al. “Modularization of a Large-Scale Business Application: A Case Study”. In: *IEEE Software* 26.2 (Mar. 2009), pp. 28–35. ISSN: 0740-7459. DOI: 10.1109/MS.2009.42.

[75] D. Taibi, V. Lenarduzzi, and C. Pahl. “Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation”. In: *IEEE Cloud Computing* 4.5 (Sept. 2017), pp. 22–32. DOI: 10.1109/MCC.2017.4250931.

[76] Luciano Baresi and A Coen-Porisini. “An approach for designing and enacting distributed simulation environments”. In: *International Conference on Software: Theory and Practice, Beijing, China.* 2000, pp. 25–28.

[77] D. C. Luckham et al. “Specification and analysis of system architecture using Rapide”. In: *IEEE Transactions on Software Engineering* 21.4 (Apr. 1995), pp. 336–354. ISSN: 0098-5589. DOI: 10.1109/32.385971.

[78] David Garlan, Robert Monroe, and David Wile. “Acme: An Architecture Description Interchange Language”. In: *Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative Research.* CASCON ’97. Toronto, Ontario, Canada: IBM Press, 1997, pp. 7–. URL: http://dl.acm.org/citation.cfm?id=782010.782017.

[79] *Structure101 Home » Structure101.* en-US. URL: https://structure101.com/ (visited on 09/05/2018).

[80] *hello2morrow - Sonargraph.* URL: https://www.hello2morrow.com/products/sonargraph (visited on 09/05/2018).

[81] Richard Gronback. *Eclipse Modeling Project | The Eclipse Foundation.* en. URL: https://www.eclipse.org/modeling/emf/ (visited on 08/06/2018).

[82] James J Nutaro. *Discrete-Event Simulation of Continuous Systems.* 2007.

[83] *KAMP GitHub repository.* original-date: 2017-06-02T08:20:59Z. June 2017. URL: https://github.com/KAMP-Research/KAMP (visited on 08/01/2018).

[84] Jae-Hyun Kim and Tag Gon Kim. “Hierarchical HLA: Mapping hierarchical model structure into hierarchical federation”. In: *Proc. of M&S-MTSA’06* (2006), pp. 75–80.

[85] Wentong Cai, S. J. Turner, and Boon Ping Gan. “Hierarchical federations: an architecture for information hiding”. In: *Proceedings 15th Workshop on Parallel and Distributed Simulation.* 2001, pp. 67–74. DOI: 10.1109/PADS.2001.924622.

[86] Victor R. Basili et al. “The Goal Question Metric Approach”. In: *Encyclopedia of Software Engineering.* Wiley, 1994, 2:528–532.

[87] Jennifer Horkoff et al. “Evaluating Modeling Languages: An Example from the Requirements Domain”. In: *Conceptual Modeling*. Ed. by Eric Yu et al. Cham: Springer International Publishing, 2014, pp. 260–274. ISBN: 978-3-319-12206-9.

[88] Y. Rubner, C. Tomasi, and L. J. Guibas. “A metric for distributions with applications to image databases”. In: *Sixth International Conference on Computer Vision (IEEE Cat. No.98CH36271)*. Jan. 1998, pp. 59–66. DOI: 10.1109/ICCV.1998.710701.

[89] J. Puzicha et al. “Empirical evaluation of dissimilarity measures for color and texture”. In: *Proceedings of the Seventh IEEE International Conference on Computer Vision*. Vol. 2. Sept. 1999, 1165–1172 vol.2. DOI: 10.1109/ICCV.1999.790412.

[90] Herbert Büning and Götz Trenkler. *Nichtparametrische statistische Methoden : [mit 69 Tabellen]*. 2., erw. u. völlig überarb. Aufl. de Gruyter Lehrbuch. Berlin [u.a.]: de Gruyter, 1994. ISBN: 3-11-013860-3; 3-11-014105-1; 3-11-016351-9. URL: http://digitale-objekte.hbz-nrw.de/webclient/DeliveryManager?pid=1214313&custom_att_2=simple_viewer.

[91] *MoSimEngine - MoSimLanguage, WorkwaySimModel, WorkwaySim*. en. URL: https://github.com/MoSimEngine (visited on 09/07/2018).

[92] Larry B. Christensen. *Experimental methodology*. 10. ed., Pearson internat. ed. Boston: Pearson/Allyn and Bacon, 2007. ISBN: 0-205-48473-5.