Modelling and Architecture of a Generic Framework for Integrative Environmental Simulations

Inauguraldissertation

an der Fakultät für Mathematik, Informatik und Statistik
der Ludwig-Maximilians-Universität München

zur Erlangung der Würde eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

vorgelegt von

Matthias Ludwig

aus Augsburg

am 7. Juni 2011

Erster Berichterstatter:	Prof. Dr. Rolf Hennicker Ludwig-Maximilians-Universität, München
Zweiter Berichterstatter:	Prof. Dr. Bernhard Rumpe Rheinisch-Westfälische Technische Hochschule, Aachen
Tag des Rigorosums:	4. Juli 2011

Abstract

In the face of Global Climate Change the scientific evaluation of possible impacts on the natural and social environment became more and more important in recent years. In this context computer-based simulation has emerged as a means of choice in many scientific disciplines as it facilitates predictions about future impacts of climate change based on well-known physical and societal relations. While in the past simulation models were developed within single disciplines one is nowadays particularly interested in interdisciplinary modelling which can be achieved by coupling of formerly independent simulation models from different disciplines. Thus the complex, mutually dependent processes occurring in nature and in socio-economic systems can be studied and analysed in the model. Coupling of simulation models from various disciplines is a non-trivial task; the problems one has to cope with range from the usage of different (physical) units, over diverse granularity of the models concerning space and time, up to completely different modelling and simulation approaches. It emerged – even in the context of this work – that methods of computer science render considerable support in this context.

In this work the development of Generic Framework for Integrative Environmental Simulations is presented, which enables the coupling of spatially distributed, time-discrete simulation models exchanging data among each other via interfaces at run time. The framework was developed by applying a particular software engineering methodology – which is also presented within this thesis – based on modelling on different abstraction levels and several system views. For each abstraction level (requirements, design, components) precise rules for modelling as well as conditions for the consistency of structural and behavioural models are given. Moreover, an intuitive refinement relation between the single levels is introduced.

For the Generic Simulation Framework under consideration the three system views "Data Exchange between Simulation Models", "Modelling of the Simulation Space", and "Time Coordination of Integrative Simulations" were identified and – emanating from a common base architecture – independently elaborated in the course of this thesis. Finally, the architectures of the different views are integrated to obtain the complete framework architecture. Furthermore, we provide guidelines for the application of the framework for creating an integrative environmental simulation system and describe an implementation of the framework as distributed system in Java. Finally we report on the successful application of the framework within the interdisciplinary research project GLOWA-Danube in the context of which the integrative simulation and decision support system DANUBIA was developed as an instance of the framework under consideration. With DANUBIA simulation runs are executed to investigate strategies for a sustainable water management in the Upper Danube watershed.

Zusammenfassung

Im Angesicht des globalen Klimawandels bekam die Erforschung von dessen Auswirkungen auf die Umwelt im Laufe der Zeit immer größere Bedeutung. In diesem Zusammenhang hat sich die rechnergestützte Simulation in vielen wissenschaftlichen Disziplinen als geeignetes Mittel erwiesen, da mit ihrer Hilfe Vorhersagen über zukünftige Auswirkungen des Klimawandels auf der Basis bekannter physikalischer und soziologischer Zusammenhänge getätigt werden können. Während in der Vergangenheit fachspezifische Modelle zur Simulation in den einzelnen Wissenschaftsbereichen entwickelt und verwendet wurden, ist man heutzutage vor allem an einer fächerübergreifenden Modellierung interessiert, die zum Beispiel durch Kopplung vormals eigenständiger Modelle erfolgen kann. Dadurch lassen sich auch die gegenseitigen Abhängigkeiten, Wirkzusammenhänge und Rückkopplungen der in Natur und Gesellschaft ablaufenden Prozesse im Modell abbilden. Die Kopplung von Modellen aus verschiedenen Fachbereichen ist jedoch keineswegs trivial; die hierbei auftretenden Probleme reichen von der Verwendung unterschiedlicher (physikalischer) Einheiten, unterschiedlicher Granularität in Raum und Zeit, bis hin zu vollständig verschiedenen Modellierungs- und Implementierungsansätzen. Es hat sich – auch im Rahmen dieser Arbeit – gezeigt, dass die Informatik mit ihren Methoden hier wertvolle Hilfe leisten kann.

In der vorliegenden Arbeit wird die Entwicklung eines generischen Frameworks zur integrativen Umweltsimulation vorgestellt, mit dem räumlich aufgelöste, zeit-diskrete Simulationsmodelle gekoppelt werden können und zur Laufzeit über Schnittstellen untereinander Daten austauschen. Die Entwicklung und Modellierung des Frameworks erfolgte nach einer ebenfalls in dieser Arbeit vorgestellten Methodik, die auf verschiedenen Abstraktionsebenen und einer sichtweisen Entwicklung des Systems beruht. Für jede in der Methodik auftretende Abstraktionsebene (Anforderungsanalyse, Entwurf und Komponentenarchitektur) werden präzise Darstellungsregeln für die zu verwendenden Modellierungselemente sowie Bedingungen für die Konsistenz der statischen und dynamischen Modelle angegeben. Des Weiteren wird eine intuitive Verfeinerungsrelation zwischen den einzelnen Ebenen definiert.

Für das betrachtete generische Simulations-Framework wurden die Systemsichten „Datenaustausch zwischen den Simulationsmodellen“, „Modellierung des Simulationsraums“ und „Zeitliche Koordination unterschiedlicher Modellzeitschritte“ identifiziert. Diese Sichten werden im Verlauf der Arbeit – aufbauend auf einer gemeinsamen Basisarchitektur – getrennt voneinander entwickelt und schließlich zu einer Architektur des gesamten Frameworks integriert. Weiterhin werden Richtlinien zur Verwendung des Frameworks zur Erstellung eines integrativen Simulationssystems gegeben und eine Implementierung des Frameworks als verteiltes System in Java vorgestellt. Schließlich wird über die erfolgreiche Anwendung des Frameworks zur Entwicklung des integrativen Simulations- und Entscheidungsunterstützungssystems DANUBIA berichtet, das im Rahmen des interdisziplinären Umweltforschungsprojekts GLOWA-Danube entwickelt wurde. Mit DANUBIA werden Simulationsläufe zur Erforschung von Strategien für ein nachhaltiges Wasser-Management im Einzugsgebiet der Oberen Donau durchgeführt.

Acknowledgements

The present thesis was prepared during the project GLOWA-Danube, funded by the German Federal Ministry of Education and Research. I gratefully acknowledge this financial support.

The thesis was supervised by Prof. Dr. Rolf Hennicker, to whom I wish to express my gratitude not only for introducing me to a fascinating field of research, but also for his guidance throughout and for invaluable directives and suggestions for improving this work. I also sincerely thank Prof. Dr. Bernhard Rumpe for consenting to act as second reviewer.

I am grateful to all (former) colleagues within the GLOWA-Danube project for their friendly collaboration, in particular to the members of the computer science group in the different stages of the project: Michael Barth, Sebastian Bauer, Stephan Janisch, and Andreas Kraus. Likewise I thank the whole team of the Chair of Programming and Software Engineering at the Ludwig-Maximilians-Universität München for providing a perfect working atmosphere.

Finally, I thank my wife Stephanie for her love and patience, and my whole family for their encouragement and support during all the years preparing this work.

Contents

List of Figures

1 Introduction

In the face of Global Climate Change the scientific evaluation of possible impacts on the natural and social environment became more and more important in recent years. For this purpose computer-based simulation emerged as an appropriate means for making predictions about the future without changing the system under consideration.

While in the past simulation models often were developed as monolithic applications by a particular discipline to provide answers for specific questions of the respective discipline, in recent years the need of interdisciplinary research grew in order to better understand the complex, mutually dependent processes occurring in nature and in socio-economic systems. Of particular importance are water-related processes which have an eminent impact on the global change of the hydrological cycle with various consequences concerning water availability, water quality and water risks like water pollution, water deficiency, and floods. In the last decade several projects have been initiated dealing with methods, techniques and tools to support a sustainable water resource management, in particular in the context of the European Water Framework Directive [Eur00].

With the demand for interdisciplinary research new challenges came up to the scientists dealing with environmental modelling and simulation. Although computer science typically does not deal with climatic or environmental issues the methods of computer science – in particular well-founded software engineering techniques – emerged as important means for supporting the affected disciplines. In this context even separate branches of informatics arose, like environmental informatics [PH94] or geographical informatics [Bar95].

1.1 Background and Motivation

The background of this thesis is the integrative environmental research project GLOWA-Danube [LMN+03], which is part of the German initiative GLOWA [Rie04] funded by the German Ministry of Education and Research (GLOWA is an abbreviation for “Globaler Wandel des Wasserkreislaufs" which means "Global Change of the Hydrological Cycle" in English) and ran from 2001 to 2010.

It was the goal of GLOWA-Danube to develop and validate integration techniques, simulation models and new monitoring methods to assess the impact of Global Change on the water resources of the Upper Danube basin. For this purpose the integrative simulation and decision support system DANUBIA was developed to identify strategies for the management of water by analysing different global change scenarios for the period 2011–2060. A prototype of DANUBIA was developed in 2004 (cf. [BHKL04]). The final version of DANUBIA was released in 2010.

It is not the goal of this thesis to describe DANUBIA, though the development of DANUBIA generated the motivation for this work. For this purpose the DANUBIA system was conceived as a framework-based system, consisting of a Generic Simulation Framework which is independent of the specific simulation models on the one hand, and a set of simulation models which are developed locally by several model developers on the other hand. The mentioned Generic Simulation Framework is the subject of the thesis at hand. It provides not only a solution for the particular problems of GLOWA-Danube, but rather a solution for a wide range of similar problems – in this case the coupling of simulation models from various scientific disciplines.

1.2 Modelling and Simulation

In this section we clarify some notions in the context of (environmental) modelling and simulation used throughout the thesis and exhibit how the framework under consideration fits into this context.

The following definitions are widely borrowed from [Gar01]. A *(real) system* is a part of the real world under study which can be identified from the rest of its environment for a specific purpose. A *model* is an abstract representation of a system. Usually a model is simpler than the real system, but equivalent to it in all relevant aspects. *Modelling* means the process of developing a model. Typically the first step of modelling results in an abstract *conceptual model*, which is in the next step refined to an executable *simulation model* which imitates the behaviour of the system under certain conditions. The execution of a simulation model is called a *simulation run* or *simulation* for short.

There are several categories of simulation model, the most commonly used include the following: *Physical models* like, e.g., physical devices to train pilots, or wind tunnels to study the aerodynamic properties of an aircraft. *Graphical models* which display the behaviour of the system using a graphical representation. *Mathematical models* which are represented by a set of mathematical expressions and/or logical equations expressing the relationships among the entities within the system. Mathematical models are the most flexible and most powerful types of model. They are usually implemented on a computer using software tools like MATLAB and Simulink (cf. [Beu02]) or a general purpose programming language like C++ or Java.

We consider in the following as a simulation model a computer program that simulates an environmental process over a certain time span, called the *simulation time*, with regard to a certain geographical area of the environment, called the *simulation space*. In an *integrative simulation system* several simulation models are coupled in order to analyse dependencies and feedbacks of the simulated processes.

Following [LB06] in [GJKH06] computer-based environmental simulation models can be classified with respect to their

- *basic modelling approach* (process-, data- or agent-based modelling)
- *treatment of simulation space* (spatially distributed or lumped)
- *treatment of simulation time* (discrete or continuous)

Moreover, an integrative environmental simulation system comprising a set of simulation models can be typed with respect to the

- *mode of execution* (parallel or sequential)
- *mode of coupling* between the simulation models (loose or tighten)
- *treatment of (a common) simulation space*
- *treatment of (a common) simulation time*

In this work we focus – on demand of the underlying environmental research project – on process-based simulation models which are spatially distributed and work on an arbitrary but discrete time scale. Coupling is achieved by data exchange between the parallel executed models at run time via interfaces.

1.3 Problems of Integrative Modelling

Coupling of simulation models from various disciplines is a non-trivial task. One has, among others, to cope with the following problems.

- *Different (physical) units.*
- *Different simulation approaches.* While in natural sciences a process- or data-based simulation approach is preferred, in social sciences an agent-based approach is most likely the means of choice.
- *Different spatial resolutions.* While natural science models typically use grid-based spatial resolutions ranging from a few metres to a few kilometres, social science models often consider geographical units defined by political boundaries, like, e.g., administrative districts or countries.

- *Different time scales.* The time scale of a simulation model depends strongly on the modelled processes and typically ranges from minutes or hours in natural sciences to months or years in social sciences.

Each of the problems mentioned above is addressed in a way by the Generic Simulation Framework under consideration.

1.4 Approach: a Generic Framework for Integrative Environmental Simulations

In this section we briefly introduce our approach of a generic simulation framework for integrative environmental simulations. A more comprehensive summary is provided in Chapter 2.

A framework is a set of cooperating classes that make up a reusable design for a specific class of software [GHJV95]. During an in-depth analysis of the problem domain which accommodated best practices of software engineering, like, e.g. abstraction, separation of concerns, we have identified three major requirements for an integrative simulation framework. The framework should support

1. data exchange between concurrently running simulation models,
2. consistent treatment of simulation space, and
3. coordination of simulation models according to simulation time.

These requirements are realised in a component-based simulation framework which supports independent implementability and substitutability of components by relying only on interface specifications. The framework defines *generic components* that implement common structure and behaviour, thus imposing general rules which must be respected by concrete simulation models when it comes to framework instantiation.

Orthogonally to the modularisation by components (which we do not consider in this section) the framework is split into two logical layers, the *framework core* and the *developer interface*. To explain the principle ideas behind these layers we consider Figure 1.1.

The framework core (depicted in dark blue) comprises an implementation of all features that can be handled by the framework itself like, e.g. the time coordination, the common properties of a simulation model, and the management of the spatial distribution. In contrast, the developer interface (light blue) is intended to facilitate the implementation of a simulation model. It provides a programming interface, where particular elements exhibit so-called *plug-points* (denoted by triangular gaps in the figure, cf. [DW99]) which have to be filled with appropriate *plug-ins* in order to obtain an executable system. In our context the plug-ins are provided by the concrete simulation models which are represented by the polygons labelled M1, …, M4 in the figure. The simulation models complete (or instantiate) the generic framework to a concrete simulation system.

As the figure suggests, the developer interface lies between the simulation models and the framework core, making the complex framework core transparent for the model developer. The considerable advantage of this design is that the model developer can concentrate on the scientific model code, without being concerned with administrative issues, like, e.g. model linking or time coordination, which are tasks of the framework core.

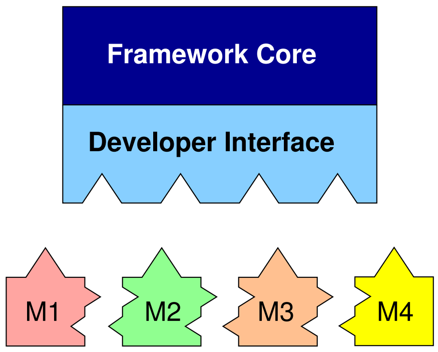


Figure 1.1: Idea of the framework-based system development

For the development and modelling of the framework a methodology mainly using the Unified Modeling Language (UML, [JBR05]) as a graphical notation and the Object Constraint Language (OCL, [WK03]) for specifying constraints has been developed. This methodology takes into account different abstraction levels and several system views. The abstraction levels considered by the methodology include *requirements*, *design* and *components*. For each level rigorous rules for the depiction of the single model elements, as well as consistency conditions between structural and behavioural models are provided. Moreover, intuitive refinement relations between the single levels are given.

Concerning the Generic Simulation Framework the following views to be considered by the methodology have been identified: *Data Exchange between Simulation Models*, *Modelling of the Simulation Space*, and *Time Coordination of Integrative Simulations*. These views correspond to the requirements stated above and are developed by extending a common *base* which is given by the basic requirements and system use cases of the framework. The complete framework architecture is obtained by the *integration* of the architecture of the single views according to the methodology.

The diagram in Figure 1.2 depicts the application of the methodology on the development of the Generic Simulation Framework (with abbreviated names of the views and of the abstraction levels). For reasons of clarity only one extension arrow ($\hookrightarrow$) is drawn from the base to each view (there should be another two, namely one for each abstraction level). The figure reflects the organisation of this thesis: the base, the views, and the integration are dealt with in a separate chapter each, while within each chapter the abstraction levels and refinement steps are discussed in subsequent sections (except for the integration chapter where only the components level is considered). We will revisit this diagram at the beginning of each chapter as a guidance for the reader through the development process.

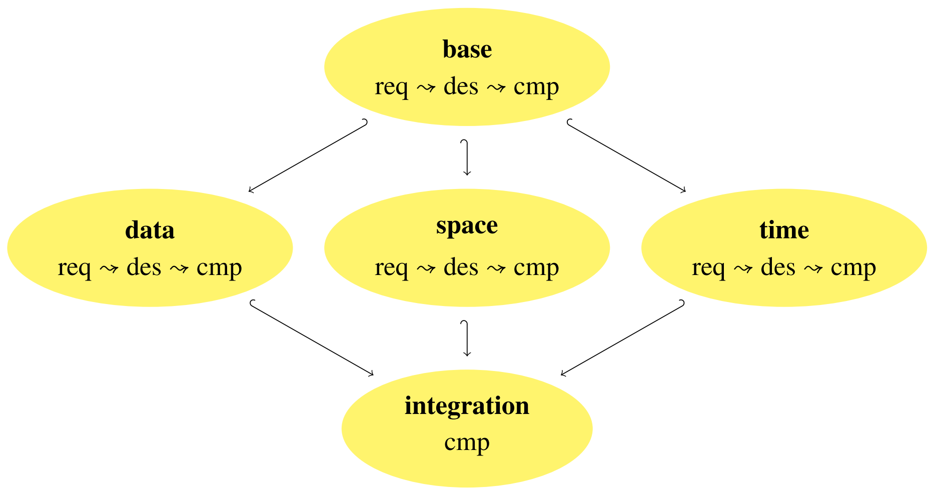


Figure 1.2: Development of the Generic Simulation Framework by different views and abstraction levels

The framework has been implemented as a distributed system in Java and successfully applied within the interdisciplinary research project GLOWA-Danube to construct the integrative simulation and decision support system DANUBIA. It is generic in that sense that it is, in principle, applicable to any kind of model which simulates environmental processes on a grid-based space with an arbitrary, but discrete time scale.

1.5 Related Approaches

When considering related approaches of our work, we have to take into account both: approaches which evolved in the field of integrative environmental simulation systems as well as approaches in software engineering which are related to our development methodology for complex systems.

Concerning the first part an overview of requirements and some examples of modelling systems is provided by [Arg04] and [RA06]. A more recent overview of environmental modelling systems and frameworks (on the regional as well as on the global scale) is given in [Jag10]. A number of other frameworks supporting integrated environmental modelling emerged since the GLOWA-Danube project started in 2001. In the field of integrated water resource management, in particular in the context of the European Water Framework Directive [Eur00], there are, e.g., the Object Modelling System OMS (cf. [KKO05]), ModCom (cf. [HBvEL03]), The Invisible Modelling Environment TIME (cf. [RSP$^+$03]), and the Open Modelling Interface OpenMI (cf. [GGW07]). While TIME is a platform for the development of stand-alone modelling tools, OMS, ModCom, and OpenMI are frameworks which support the independent development of models and allow for execution of coupled simulations. In particular, OpenMI is designed to extend existing stand-alone models by standard interfaces for coupling.

Concerning one of the main characteristics of our approach, namely the distributed and parallel execution of coupled simulation models, we observe that none of the aforementioned frameworks have all of these properties. While, for example, OpenMI allows only for sequential execution of coupled models, OMS in fact facilitates parallel execution, but only for independent (uncoupled) models. ModCom and TIME are both not designed for distributed execution.

We will reconsider the related approaches in the discussion section of each chapter concerning a view of the simulation framework. In [Kou09] a comparative study of recent environmental simulation systems and frameworks having regard to our views has been elaborated under supervision of the author.

Related approaches concerning our software engineering methodology are mentioned directly in the respective chapter.

1.6 Organisation of the Thesis

This thesis consists of twelve chapters, including an introduction (Chapter 1) and some conclusions (Chapter 12). In Chapter 2 we present an overview of the thesis in that the development and application of the integrative environmental simulation framework under consideration is exhibited in a compressed manner.

Chapter 3 describes the methodology of system development applied in the subsequent chapters, which is based on the separate treatment of different system views on different levels of abstraction. In Chapter 4 we describe the system use cases and develop a base architecture for the framework. The following chapters elaborate on the fundamental views of the framework: data exchange between simulation models (Chapter 5), modelling of simulation space (Chapter 6) and time coordination of simulation models (Chapter 7).

In Chapter 8 the previously developed design models of the different views are integrated to a design model of the framework considering all of the mentioned views. Chapter 9 describes how the framework can be applied within an integrative simulation project, i.e. how it has to be enhanced to build up a complete integrative environmental simulation system. Thereafter a reference implementation of the framework in the object-oriented programming language Java is presented in Chapter 10. Chapter 11 reports on the successful application of our framework in the integrative environmental project GLOWA-Danube where the integrative environmental simulation and decision support system DANUBIA has been developed as an instance of the framework.

2

The Generic Simulation Framework in a Nutshell

In this thesis, we report on a generic framework for computer-based environmental modelling which supports the run time coupling of various simulation models from natural science and socio-economic disciplines. The framework is generic in the sense, that it is, in principle, applicable to any kind of model which simulates spatially distributed environmental processes on an arbitrary, but discrete time scale. During an integrative simulation for some simulation period, the framework coordinates the coupled models which run in parallel exchanging iteratively data via their interfaces. For the development of the modelling framework best practices of software engineering have been applied like abstraction, separation of concerns and formal methods based on precise mathematical notations. For instance, the framework provides abstractions of simulation models and thus facilitates the development and integration of concrete simulation models of particular disciplines. Separation of concerns involves the aspects of information exchange between simulation models, consistent modelling of simulation space, and coordination of concurrently running simulation models.

Technically, for the development of the simulation framework the Unified Modeling Language UML (cf., e.g. [RJB05]) has been used in the requirements and in the design phase, while a framework implementation is programmed in Java making use of Java's Remote Method Invocation interface (RMI) for communication between distributed components. Formal methods have been applied to specify the requirements for the general life cycle each simulation model must obey and for specifying and proving the correctness of the coordination of distributed simulation models, for which the framework is responsible. For this purpose, the process algebra Finite State Processes FSP (cf. [MK06]) has been used. For the specification of requirements concerning correct simulation configurations, invariants have been stated and formalised in terms of the Object Constraint Language OCL (cf. [WK03]). The framework design follows a component-oriented approach which allows to plug in arbitrary simulation models as long as the common requirements concerning, e.g., the simulation space and the life cycle of a model are satisfied. Thus the integration of different simulation models into a coupled simulation is considerably facilitated and (most) errors occurring in simulation models caused by not meeting common requirements can be detected already at compilation time.

The framework has been developed and successfully applied to construct the integrative simulation system DANUBIA within the interdisciplinary research project GLOWA-Danube[1] ([LMN+03]), which is part of the German national initiative GLOWA(Global Change in the Hydrological Cycle) funded by the German Federal Ministry of Education and Research between 2001 and 2010. Within GLOWA-Danube, a group of researchers from various natural science and socio-economic disciplines joined forces to investigate the impact of climate change on the water cycle within the Upper Danube watershed and to support the development and evaluation of regional adaptation strategies. Actually 15 simulation models have been developed by the research groups of GLOWA-Danube, such that various simulation configurations can be built and run by our modelling framework within the DANUBIA system.

The outline of this Chapter is as follows. In Section 2.1, we identify common requirements for integrative environmental simulations. Then, in Section 2.2, we summarise the design of our framework in accordance with the given requirements. Section 2.3 introduces briefly the integrative simulation system DANUBIA obtained by framework instantiation within the context of the GLOWA-Danube project.

2.1 Requirements for Integrative Environmental Simulations

We consider in the following as a *simulation model* a computer program that simulates an environmental process over a certain time span, called the *simulation time*, with regard to a certain geographical area of the environment, called the *simulation space*. In an *integrative simulation system* several simulation models are coupled in order to analyse dependencies and feedbacks of the simulated processes. It is obvious that, besides the technique of coupling itself, the consistent treatment of simulation time and simulation space is crucial for the integration of different simulation models. Moreover, for a comprehensive environmental simulation not only processes occurring in nature but also processes reflecting human behaviour must be taken into account. Therefore we have identified three major requirements. The framework should support

1. data exchange between simulation models at run time,
2. consistent treatment of simulation space, and
3. coordination of simulation models according to simulation time.

In the following we elaborate more on each of the three requirements.

[1] `http://www.glowa-danube.de`

2.1.1 Data Exchange between Simulation Models

The coupling of simulation models is based on *interfaces*. Interfaces for data exchange specify data queries. We distinguish between *provided* interfaces specifying queries for data that is provided by a simulation model, and *required* interfaces specifying queries for data that is needed by a simulation model for its own computation. The general requirements concerning data exchange are modelled in the UML class diagram in Figure 2.1. It says that a simulation may involve arbitrarily many models, which play the role of the participating models for the simulation, and that a model may have arbitrarily many interfaces, playing the role of provided or required interfaces.

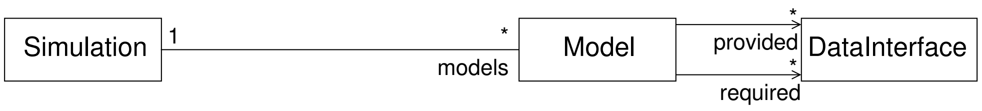


Figure 2.1: Requirements model for data exchange

A concrete example of a provided and required interface is given later on when we illustrate the framework instantiation in Figure 2.7. The following invariant expresses a consistency requirement for data exchange which must be satisfied for any integrative simulation.

Invariant for data exchange

- In an integrative simulation, for each required interface of each participating model there exists exactly one participating model which provides that interface.

This invariant can be formalised in terms of the following OCL-expression:

```
context Simulation inv:
    self.models.forAll(m |
      m.required->forAll(r |
        self.models->one(n |
          n.provided->includes(r))))
```

2.1.2 Modelling of Simulation Space

In an integrative environmental simulation the consistent treatment of the underlying simulation space is crucial. It is obvious that in spatially distributed simulations one needs geographical atoms, which in the following will be called *proxels*. The term proxel (cf. [TK99]) stems from *process pixel* and suggests that a proxel does not only model a structural element of the simulation space, but it shows also dynamic behaviour by simulating the environmental processes on this particular geographical unit. The entire simulation area is then modelled by a set of (non-overlapping) proxels.

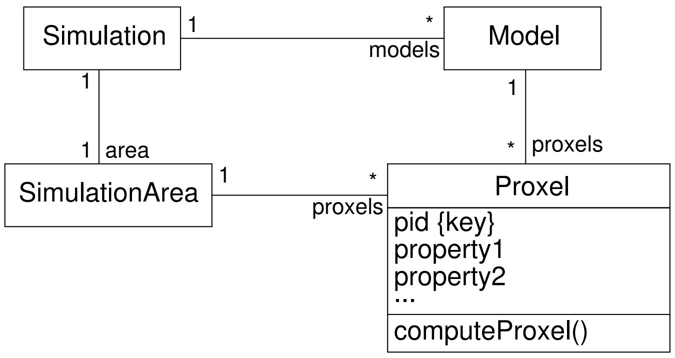


Figure 2.2: Requirements model for simulation space

The spatial requirements of an integrative simulation are described by the UML class diagram in Figure 2.2. It says that a simulation concerns always exactly one simulation area which, in turn, consists of a set of proxels. The class Proxel requires that each proxel has a unique identifier pid and an operation computeProxel() to compute the next state of a proxel in each time step. Moreover, each proxel can have a number of properties which must be common to all simulation models (like, e.g., geographical coordinates, elevation, etc.). On the other hand, each simulation model has a set of proxels, on which it operates. The following invariant requires that the models participating in an integrative simulation agree on the set of proxels determined by the area of the simulation.

Invariant for the simulation space

- In an integrative simulation, all participating models operate (only) on proxels which belong to the simulation area of the simulation.

A formalisation of this invariant in terms of an OCL constraint is given in Section 6.1. Besides the basic properties, a proxel may store domain-specific properties as illustrated for groundwater proxels in Figure 2.7 later on.

2.1.3 Coordination of Simulation Time

An important characteristics of our problem domain is the concurrent execution of different simulation models which iteratively exchange information at run time via their interfaces. In order to guarantee the consistency of data exchange during a simulation run, the single simulation models must be appropriately coordinated with respect to the progressing simulation time. The *correct* coordination is a non-trivial task since, in general, simulation models have different, individual time steps determining the model time between two consecutive computations. Model time steps depend, of course, on the simulated processes which typically range from minutes or hours, like in natural sciences, to months, like in social sciences. Hence a precise, unambiguous specification of the coordination problem is mandatory. We first describe the general life cycle which a simulation model must follow:

- *provide* initial data at the model's provided interfaces
- while not at simulation end
  - *get data* from the model's required interfaces
  - *compute* new data for the next time step
  - *provide* newly computed data at the model's provided interfaces

For the formalisation of a model's life cycle we use the process algebra Finite State Processes FSP; cf. [MK06]. The following FSP process MODEL specifies the general behaviour of a simulation model. In order to be generally applicable the process is parameterised with respect to the model's time step. The sequence of actions in line 5, getData[t] -> compute[t+Step] -> provide[t+Step], is iteratively performed with increasing time t and thus formalises the iteration in the informal description of a model's life cycle given above. Note that the computation of new data for time t+Step relies on data obtained for time t. This time difference avoids deadlocks of concurrently running models (in the case of feedback loops) but it may also lead to imprecisions whose relevance must be analysed in concrete cases and, if necessary, can be improved by using smaller time steps.

```
1 range SimTime = SimStart..SimEnd
2 MODEL(Step) = (start -> provide[SimStart] -> M[SimStart]),
3 M[t:SimTime] =
4     if (t+Step <= SimEnd)
5     then (getData[t] -> compute[t+Step] -> provide[t+Step] ->
6         M[t+Step])
7     else (finish -> STOP).
```

When several simulation models are executed in parallel, it is essential that only valid data is exchanged, i.e. data that fits to the local model time of the participating models. To specify this requirement we consider only two simulation models at a time, one, say U, acting as a user of data, and the other one, say P, acting as a data provider. From the user's point of view we obtain the coordination condition (U), from the provider's point of view the coordination condition (P).

(U) U gets data expected to be valid at time t_U only if the following holds:
The next data that P provides is valid at time t_P with $t_U < t_P$.

(P) P provides data valid at time t_P only if the following holds:
The next data that U gets is expected to be valid at time t_U with $t_U \geq t_P$.

Condition (U) ensures that the user does not get obsolete data while condition (P) guarantees that data, available at the provider's interface, will not be overwritten if it is not yet considered by the user model. If one can show that all (pairwise) combinations of all models participating in an integrative simulation considered in both roles, as user and as provider of data, satisfy the two coordination requirements, then the whole integrative simulation is coordinated correctly. We have again used FSP to formalise the coordination conditions in terms of so-called property processes. Then, in a next step, we have constructed an explicit coordination process with FSP and we have verified by model checking techniques that the coordination process provides a solution for the coordination problem of integrative simulations (which later on can be implemented in Java).

2.2 Simulation Framework

The requirements and concepts described in the previous section are realised in a component-based simulation framework (cf. [HBJL09], [HBJL10]) which supports independent implementability and substitutability of components by relying only on interface specifications.

The framework defines *generic components* that implement common structure and behaviour, thus imposing general rules which must be respected by concrete simulation models when it comes to framework instantiation. [DW99] use the notion of *plug-point*, provided by a generic component, and *plug-in*, provided by some extension which completes the generic component to an executable implementation. In the following we focus on the framework and plug-points while, in Section 2.3, we show how the framework can be instantiated by simulation models with plug-ins.

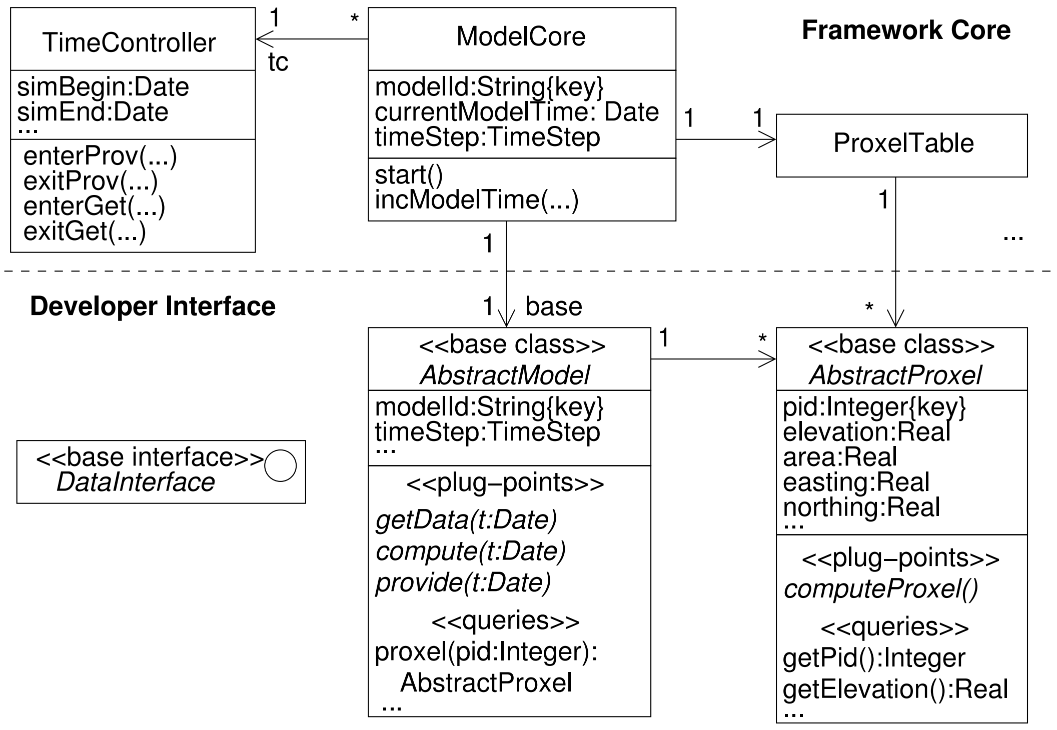


Figure 2.3: Two-layered framework architecture

The framework itself is split into two logical layers, the *framework core* and the *developer interface*. To explain the principle ideas behind these layers we consider the framework excerpt shown in Figure 2.3 (without taking into account components yet).

The framework core implements all features that can be handled by the framework itself like, e.g., the time coordination, the common properties of a simulation model (ModelCore), and the management of the spatial distribution (ProxelTable). Concerning time coordination, there exists at run time exactly one instance of the class TimeController which is a monitor object that is called by each model core instance before data is fetched from or provided to a data exchange interface. Each model core instance itself will be linked at run time to exactly one concrete simulation model which must implement the abstract operations, given by the plug-points getData, compute and provide, of the class AbstractModel of the developer interface. The UML sequence diagram in Figure 2.4 illustrates the sequence of interactions implemented in the start method by taking into account the life cycle of simulation models as described in Section 2.1.3.

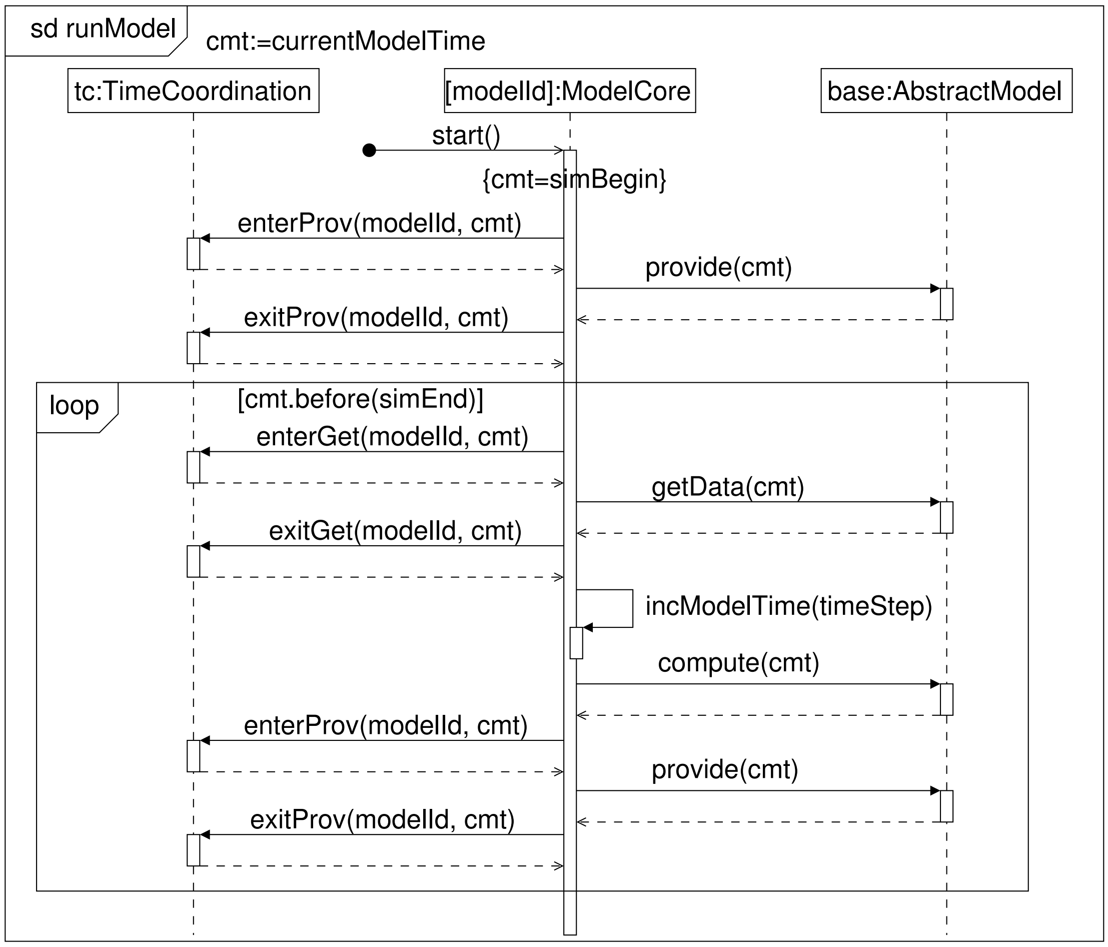


Figure 2.4: Interactions of the start method

The lower layer in Figure 2.3 constitutes the programming interface for model developers. Here one can find, corresponding to the conceptual requirements of Section 2.1, abstract base classes like AbstractModel or AbstractProxel which contain plug-points in terms of abstract operations. For example, the base class AbstractModel defines the plug-points getData, compute and provide corresponding to the actions of a simulation model within its life cycle, which have to be implemented by plug-ins, i.e. by concrete operations in a simulation model. The interface DataInterface is a marker interface for all provided and required interfaces used for data exchange.

Let us now consider how the two layers explained above fit into the component architecture of the simulation framework shown in Figure 2.5. The framework comprises several components each of them encapsulating a certain functional aspect of an integrative simulation.

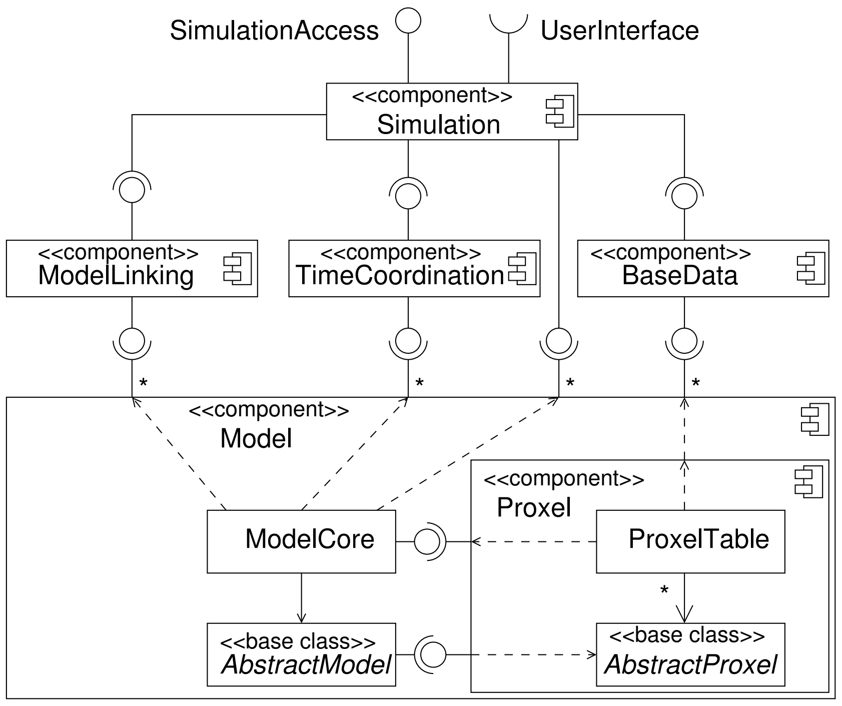


Figure 2.5: Component-based design of the simulation framework

The component Simulation is the main component for managing integrative simulations and hence provides an interface for a (graphical) user interface. This interface offers operations for starting, observing and aborting a simulation. Before starting a simulation the component Simulation checks the simulation configuration for consistency, i.e. if the invariant concerning interfaces as stated in Section 2.1.1 is fulfilled. At the beginning of a simulation the component ModelLinking establishes the links between simulation models over their interfaces for data exchange, including distribution over a network. The component TimeCoordination is responsible for the correct time coordination of the single models during a simulation run. The component BaseData reads and stores initialisation data for the basic properties of the simulation area for all simulation models. The component Model and its subcomponent Proxel are generic components which contains framework core classes as well as the classes constituting the programming interface for model developers as discussed above. In order to distinguish the elements belonging to the developer interface from those of the framework core, the former are marked with a stereotype, like, e.g. «base class», «base interface», or «data type».

We have used UML 2.0 for the complete framework design, in particular for the detailed documentation of the developer interfaces. The framework is implemented in Java SE 6 and contains approximately 25.000 lines of code.

2.3 Application of the Framework: The Danubia System

Within the GLOWA-Danube project ([LMN+03]) our simulation framework has been applied to construct the integrative simulation system DANUBIA which integrates up to 15 simulation models for natural processes (like hydrology, plant physiology, groundwater, glaciology etc.) as well as socio-economic models. The latter have been developed to model the behaviour of the involved actors in the areas of agriculture, economy, water supply, private households, and tourism based on the structure of societies and their interests.

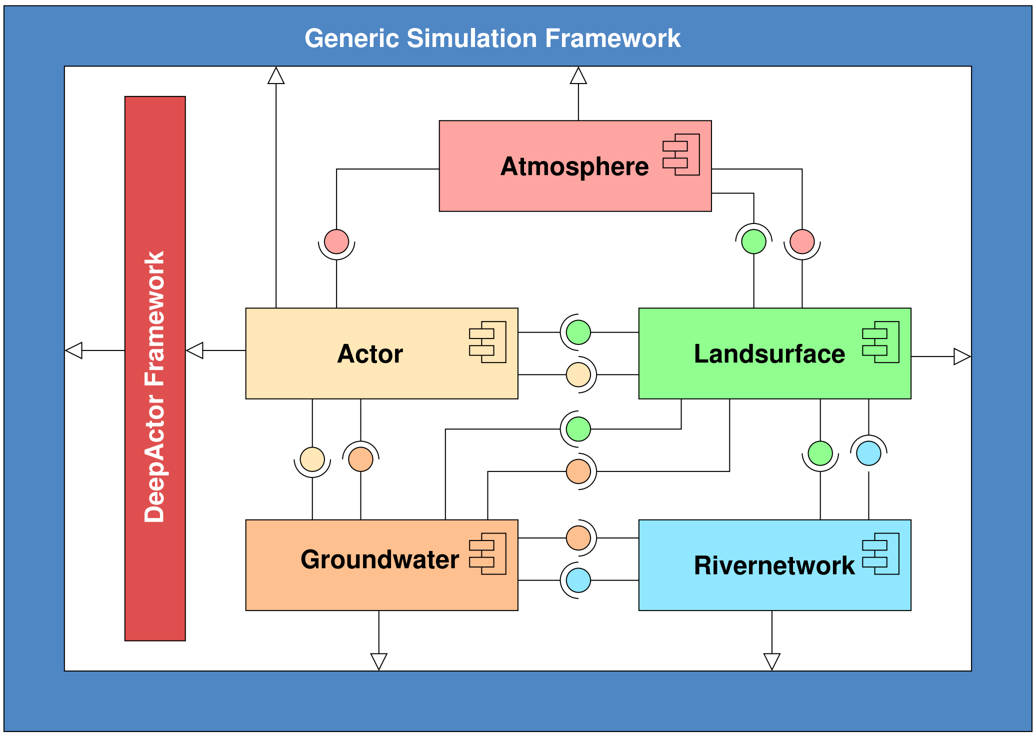


Figure 2.6: System architecture of DANUBIA

An overview of the system architecture of DANUBIA is given in Figure 2.6. The DEEPACTOR framework is an extension of the Generic Simulation Framework to address the special needs of societal simulation models (cf. [HJL10]) which is not in the scope of this thesis.

The purpose of DANUBIA is to serve as a tool for decision makers from policy, economy, and administration for the sustainable planning of water resources in the Upper Danube basin under global change conditions. DANUBIA is a distributed simulation system – the simulation models are executed in parallel on a computer cluster periodically exchanging data during a simulation run. DANUBIA was validated with comprehensive data sets of the years 1970 to 2005. It is actually in use to run and evaluate coupled simulations which are driven by climatic as well as societal scenarios for the next 50 years. DANUBIA can be flexibly configured regarding to the problem under consideration. Concerning performance, integrative simulation runs with DANUBIA over a 50 year period actually take between 36 hours and four weeks computing time, depending on the used simulation configuration.

Figure 2.7 shows in more detail, how a concrete simulation model is integrated in the framework. The upper layer of Figure 2.7 depicts (parts of) the developer interface, known from Figure 2.3, and the lower layer shows parts of a sample Groundwater model. One can see that all model classes (and interfaces) of the groundwater model extend the base classes (the base interface DataInterface resp.) of the developer interface by certain domain-specific properties, like the proxel attributes gwWithdrawal, riverLevel etc., and by providing implementations for the plug-in operations like, e.g., compute and computeProxel. Thereby the framework's core functionality concerning run time coordination, management tasks and the like is completely hidden for model developers.

Finally note that beside omitting components Figures 2.3 and 2.7 show significant simplifications with respect to the complete design model developed in the following chapters; for example, in the complete model we use extra meta data classes to describe properties like simStart, simEnd, modelId, etc.

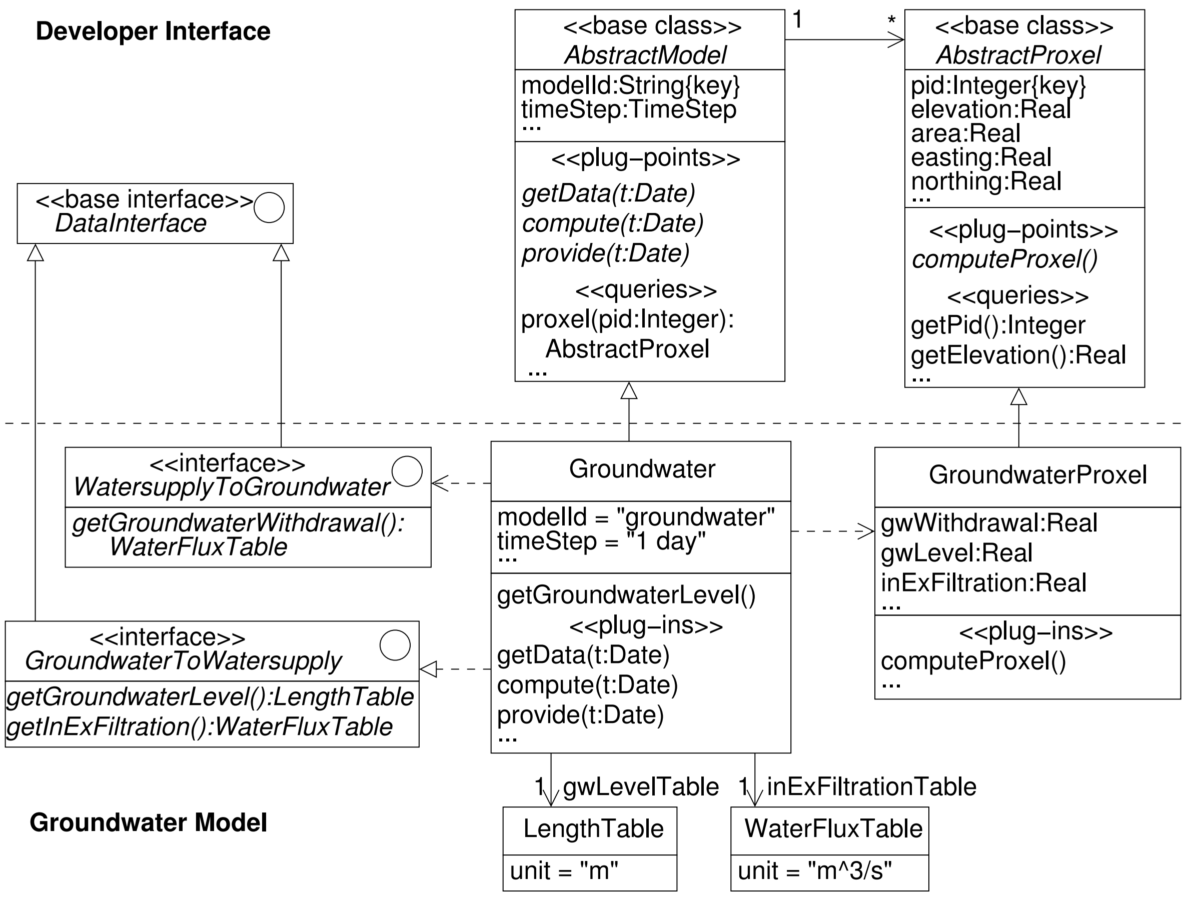


Figure 2.7: Framework instantiation

3

Methodology of System Development and Modelling

In this chapter we describe our general methodology for the development and modelling of complex software systems. Although the presented methodology emerged during the development of the generic simulation framework described in this thesis (and has of course been applied for its development), it can also be useful for the development of arbitrary systems where a separation into different system views is possible. In this context, a view is supposed to be a functional aspect of the system under consideration (not to be confused with the difference between structural and behavioural view, for instance). Examples for such views are mentioned in Section 3.5, where we describe how the development method is applied to the generic simulation framework under consideration in this thesis.[1]

The proposed methodology makes use of well known software engineering practices like object orientation, abstraction, separation of concerns, modelling techniques – in particular of the Unified Modelling Language (UML) [JBR05] and the Object Constraint Language (OCL) [WK03] – and formal software engineering methods. Its main characteristics are

- modelling on different abstraction levels (requirements, design, components),
- separate development of several independent system views on each abstraction level, and
- step-wise integration of the different views to obtain the whole system architecture.

An overview of the methodology involving two views $V1$ and $V2$ is sketched in Figure 3.1. The methodology starts with a common base view, where the abstraction levels requirements,

[1]We use the term "view" instead of "aspect" as it is recommended by the IEEE standard 1471 [IEE00]; moreover, we want to avoid confusion with the field of aspect oriented programming.

design and components are elaborated ($\rightsquigarrow$ denotes the refinement steps between these levels). Then the different views $V1$ and $V2$ are developed in that way, that the view model of each level extends the corresponding base model (denoted by $\hookrightarrow$). Finally the system views are integrated on the components level by embedding the models of the different views into an integration model for the overall system design.

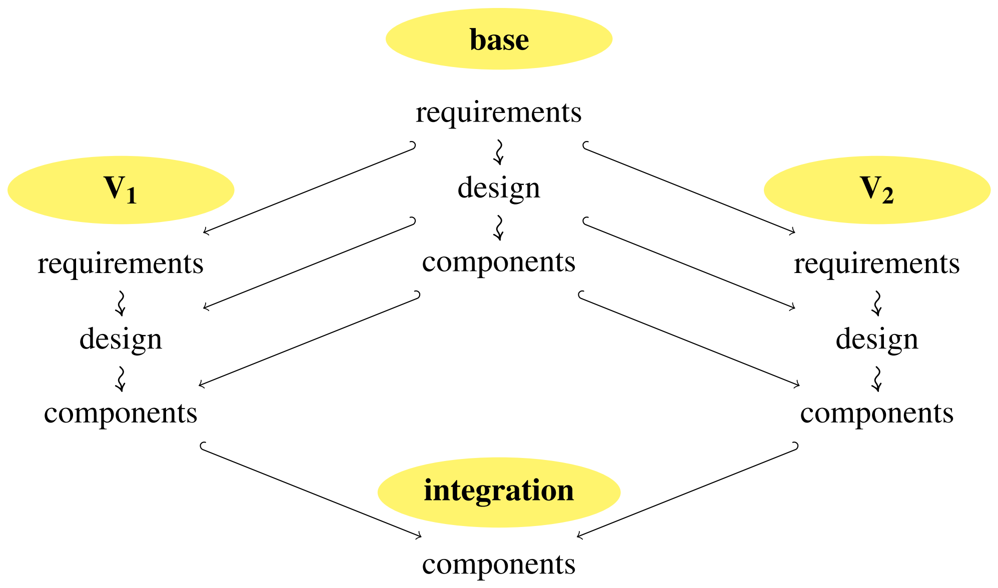


Figure 3.1: View-based development and integration on different abstraction levels

Notice, that the refinement within the views must respect the refinement of the base, so that the diagram commutes. This can be achieved in that each subdiagram of the form

$$\begin{tikzcd}
A_1 \arrow[r, hook, "②"] \arrow[d, squiggly, "①"'] & B_1 \arrow[d, squiggly, "③"] \\
A_2 \arrow[r, hook, "④"'] & B_2
\end{tikzcd}$$

is constructed as follows. We start with A_1 (e.g. base/requirements) and then build A_2 (e.g. base/design) as a refinement of A_1 (cf. ①) and B_1 (e.g. V_1/requirements) as an extension of A_1 (cf. ②), at which the order of steps ① and ② does not matter. Finally, we construct B_2 (e.g. V_1/design) as a refinement of B_1 (cf. ③), so that B_2, is also an extension of A_2 (cf. ④). Note that B_2 in general does not follow automatically from A_2 and B_1, as a refinement step usually requires additional design decisions.

The goal of the methodology is to develop a complete object-oriented system model in terms of (enhanced) UML diagrams (including components) independent from a concrete programming language. The syntax of the UML diagrams is enhanced in that way that an almost unambiguous implementation of the design model can be derived in usual object-oriented programming languages like Java, C++, etc.

The remainder of this chapter is organised as follows. First, in Section 3.1 the diagram types and modelling elements used within our methodology are provided, before in Section 3.2 the different abstraction levels used in each view are explained. Then, Section 3.3 details on view-based development and Section 3.4 on the integration of the views, and finally, in Section 3.5 we sketch how the methodology has been applied for the development of the generic simulation framework under consideration in this thesis.

3.1 Diagrams and Modelling Elements

Before we start to explain our methodology we define which UML modelling elements will be used. We do this by considering the respective excerpts of the UML superstructure specification [Obj09] (also called the UML meta model) and introduce – based on the UML meta model – a mathematical description for our models which is useful for defining refinement and extension relations as well as modelling constraints and consistency requirements later on.

The UML meta model makes strong use of class hierarchies to specify the properties of its elements which, in fact, leads to quite elegant definitions of the meta model elements, but on the other hand, it makes the meta model quite difficult to understand, because the properties of elements are often spread over a set of superclasses which in addition are most likely depicted in separate diagrams. In order to facilitate the understanding of the part of the meta model used in our methodology we show all relevant properties of a meta model element directly within or associated to this element. For example, the name of a class is represented by the attribute name of the meta class Class (cf. Figure 3.3); in the original meta model the property name is already defined within the meta class NamedElement which is the third superclass of Class.

In our methodology we use class diagrams and component diagrams for modelling structural properties, and sequence diagrams for behavioural ones. Moreover, we use the Object Constraint Language (OCL) to specify invariants and conditions. However, on the one hand we insignificantly confine the UML superstructure specification by omitting some seldom used modelling features (which will be mentioned at the respective place) in order to facilitate the definition of refinement and extension as well as the transition to an implementation in a common object-oriented programming language. On the other hand we slightly enhance the UML and OCL syntax in order to be able to better express some common programming language constructs. The extensions are explained in the following sections, too.

Notice that in our methodology every UML diagram is shown in a rectangular box which is labelled in a pentagonal box in the upper left corner of the rectangle with a stereotype indicating the respective abstraction level, the respective diagram type and the diagram name. In the remainder of this section we describe the utilised diagram types and our enhancements of the OCL.

3.1.1 Class Diagrams

On the requirements and the design level the main diagram type for structural modelling is a class diagram. A class diagram shows (a part of) the structural model on a particular detailing level (i.e. the details of some classes might be omitted or shown in a separate class diagram). Roughly speaking, a structural model consists of classes and their relationships enriched with invariants expressing constraints over the involved classes. A sample class diagram which shows the most important features is depicted in Figure 3.2. The circled numbers in the following text refer to the numbers in this diagram.

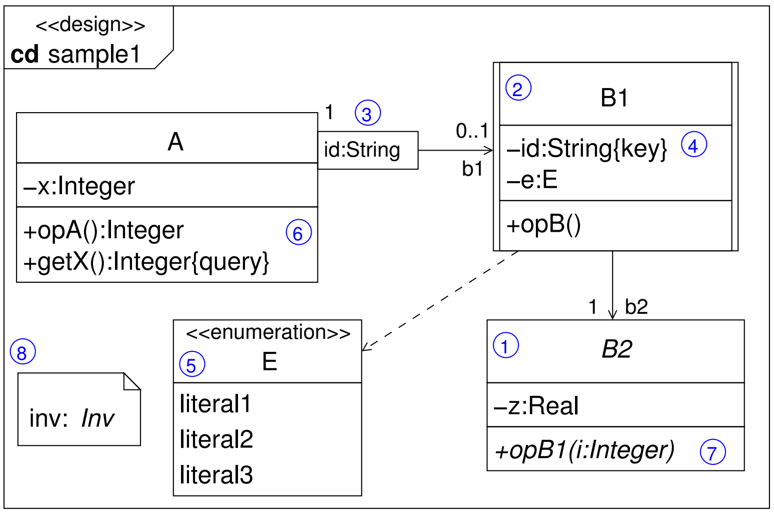


Figure 3.2: Sample class diagram showing the most important modelling features (circled numbers refer to explanations in the text)

Figure 3.3 shows the excerpt of the (adapted) UML meta model concerning classes and associations. A Class which is derived from Type always has a name and two Boolean properties which define whether the class isAbstract or isActive respectively. An abstract class (①) is depicted with an italicised name, an active class (②) with an additional vertical bar on either side of the class symbol. In our methodology, a class may have at least one superclass, i.e. we do not permit multiple inheritance for classes as this is the case in the original UML meta model. Furthermore, we require that the generalisation relation is acyclic. Then, a class possesses a set of attributes and operations. Let us first consider the attributes.

An attribute is modelled by the meta class Property which provides a name, a type as well as a multiplicity which is expressed by the attributes lower and upper, and a visibility. The single VisibilityKinds are denoted as usual with the symbols '+', '-', '#' and '~' respectively. A property may belong to an association and may have a set of qualifiers (③). Keep in mind that a qualified association end (i.e. the opposite end of the qualifier) is always multi-valued although the multiplicity is mostly 0..1. Notice that each of the aforementioned properties is optional (denoted by multiplicity 0..1 or * respectively); it depends on the actual abstraction

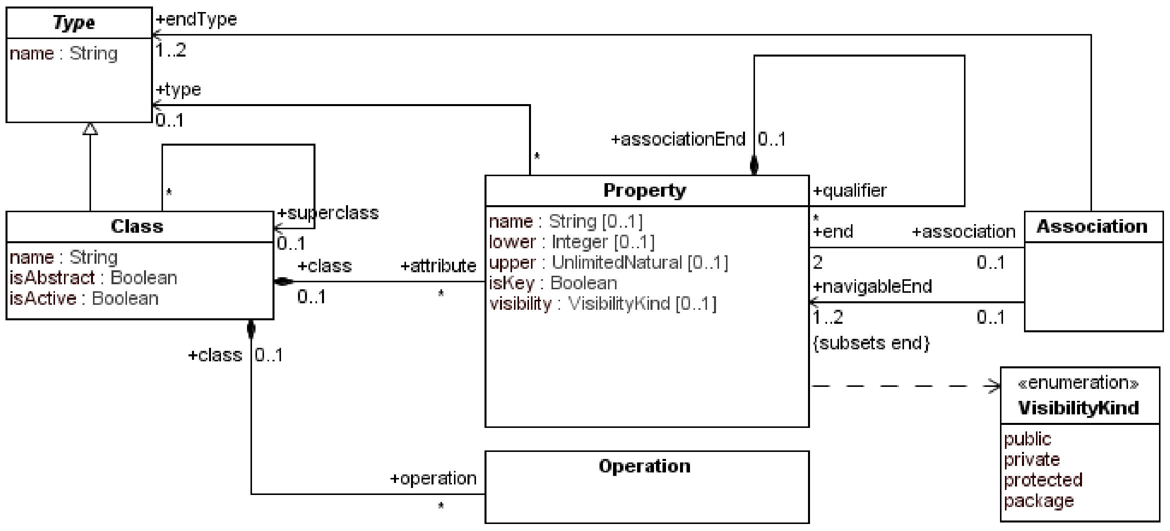


Figure 3.3: Meta model for classes and associations

level which properties must be specified. However, the following rules must be adhered to on any level:

- if the property does not belong to an association, it must have a name;
- if the lower bound is specified, then the upper bound must be greater than the lower bound or *;[2]
- if the type is a class (i.e. no data type and not unspecified) then the property belongs to an association;
- if the property belongs to an association, then it also belongs to the set of navigable ends of this association.

The attribute isKey is an extension of the original UML meta model. It denotes that the containing property is a unique (or identifying) attribute of its owning class. It is displayed as property {key} behind the attribute (④). More formally, if a class C possesses an attribute a with the property {key}, then the following invariant must be fulfilled:

```
context C
inv: C.allInstances()->forAll(c1,c2 | c1.a=c2.a implies c1=c2)
```

Concerning associations we do not support the following features: n-ary associations (for $n \geq 3$), association classes, aggregation and composition. Therefore an Association has always

[2]The type UnlimitedNatural provides the natural numbers and * for indefinite

exactly two ends, and at least one of these ends is a navigableEnd (we do not model associations which are not navigable at all). The set of types defined by endType must correspond to the types of the association ends. Moreover, we model two kinds of DataTypes: user defined Enumerations (⑤) and predefined PrimitiveTypes (Integer, Real, String, Boolean).

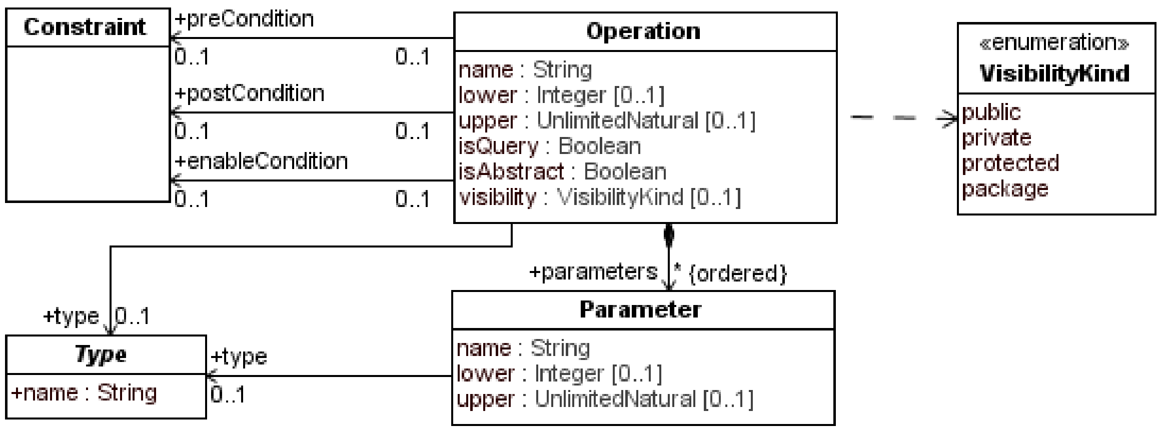


Figure 3.4: Meta model for operations

The details of the meta class Operation are depicted in Figure 3.4. An operation must always have a name, it may have a return type (if so, its multiplicity is specified by the attributes lower and upper analogously to attributes) and an (ordered) set of parameters. A parameter also must have a name and may have a type and a multiplicity. The flags isQuery and isAbstract denote whether the operation is a query operation (depicted by the property {query}, ⑥) or an abstract operation (depicted by an italicised name, ⑦) respectively. Moreover, an operation may have three kinds of conditions: a preCondition, a postCondition and an enableCondition. The enable condition is introduced in our methodology in order to express deferred behaviour. An enable condition is stated in the form enable: Q which means that the execution of the respective operation is blocked until the condition Q becomes true.

The type Constraint is supposed to be an (extended) OCL expression. Conditions of operations may be specified in the usual way in terms of OCL expressions or within a sequence diagram in the context of the respective operation. An example for the different kinds of conditions and their interpretation is provided in the following section.

Invariants concerning the classes of a class diagram are attached to the diagram within a note (⑧). For getters and setters of attributes we always assume the following implicit postconditions which ensure that these operations behave as expected. Consider a class C with an attribute a of type T. Then for the operations getA and setA the following holds.

```
context C::getA()
post: result = self.a
```

```
context C::setA(myA:T)
post: self.a = myA
```

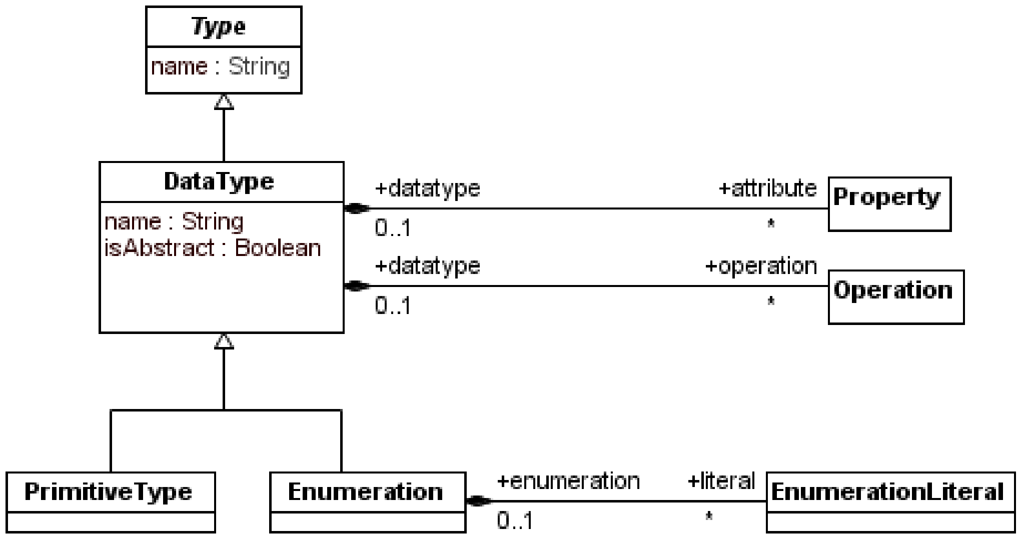


Figure 3.5: Meta model for data types

Figure 3.5 shows the meta classes concerning data types. We distinguish three kinds of data types:

- *ordinary data types* (represented by the class DataType) which may have attributes and operations;
- *enumeration types* (Enumeration) representing a user-defined enumeration consisting of a set of EnumerationLiterals;
- *primitive types* (PrimitiveType) representing primitive values like, e.g. boolean values, integers or decimals.

3.1.2 Sequence Diagrams

The behavioural model in our methodology is given by a set of sequence diagrams. A behavioural model always corresponds to a structural model which defines types and associations. As “a sequence diagram describes an Interaction by focusing on the sequence of Messages that are exchanged” ([Obj09, p. 506]), we identify a sequence diagram with the interaction it describes. In Figure 3.6 a sequence diagram is shown which exemplifies the features described in the following. Again the circled numbers in the text refer to those in the diagram.

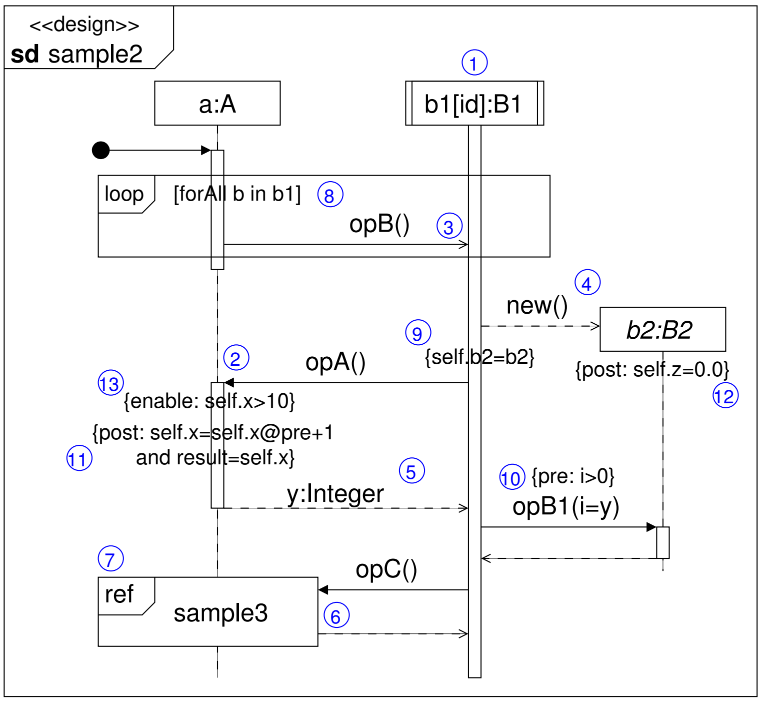


Figure 3.6: Sample sequence diagram showing the most important modelling features (circled numbers refer to explanations in the text)

The excerpt of the (adapted) UML meta model concerning interactions is depicted in Figure 3.7. Thus an Interaction possesses a name and a set of parameters and consists of a set of Lifelines, a set of Messages and a (partially) ordered set of InteractionFragments. A lifeline represents a ConnectableElement, i.e. an instance of a property of the containing classifier of the interaction. The name of the lifeline is of the form

[<*element-name*>[‘[’<*selector*>‘]’]] : <*class-name*>,

where <*element-name*> is the name of the represented ConnectableElement , e.g. a role name occurring in a class diagram, <*selector*> is the expression representing the selector (①), and <*class-name*> is the name of the type referenced by the ConnectableElement. The selector determines a particular instance in the case that there exist multiple instances of the corresponding connectable element. If the selector is omitted an arbitrary instance is chosen. If a lifeline represents an abstract class then its name is italicised. In this case for execution a concrete subtype of the abstract class has to be substituted. A lifeline head of an active class possesses two additional side-bars (analogous to the class symbol).

A Message is of a certain sort which is given by the enumeration MessageSort[3]. The different message sorts are depicted in the usual notation. A message of sort synchCall (②),

[3]Note that we omit the message sort asynchSignal as it is not necessary for our modelling purposes

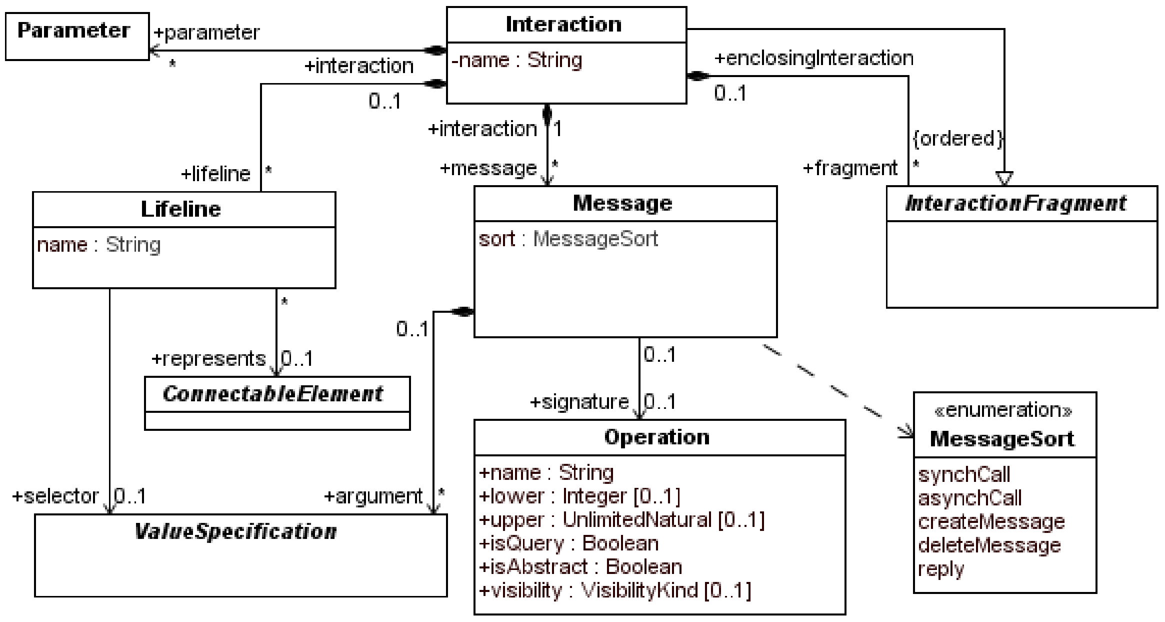


Figure 3.7: Meta model for interactions

asynchCall (④) and createMessage (④) is associated with an Operation expressed by the property signature. In the case of a createMessage which is accompanied by the keyword new we consider the constructor of the corresponding class as the associated operation. Moreover, a message may carry a set of arguments which must correspond to the parameters of the operation (or the constructor respectively). A reply message which corresponds to an operation with return type different from void is equipped with a variable containing the return value of the operation; for convenience reasons often the type of the return value is displayed, too (⑤).

"An InteractionFragment is an abstract notion of the most general interaction unit. An interaction fragment is a piece of an interaction. Each interaction fragment is conceptually like an interaction by itself." ([Obj09, p. 487]). Furthermore, an interaction fragment covers a set of lifelines. As InteractionFragment is an abstract class (cf. Figure 3.8) we have to consider its concrete subtypes for modelling: MessageEnd[4], StateInvariant, CombinedFragment, and InteractionUse. In the following we discuss those subtypes.

MessageEnd and StateInvariant are subsumed as basic interaction fragments as they cover exactly one lifeline. A MessageEnd is associated to its corresponding message which in turn defines the message ends as sendEvent and receiveEvent respectively. A message with a missing sendEvent is called *found message* and a message with a missing receive event is called *lost message*. A special kind of a MessageEnd is a Gate which "is a connection point for relating a

[4]In the UML specification the class MessageEnd is abstract, but we prefer to use this class instead of its subclass MessageOccurrenceSpecification

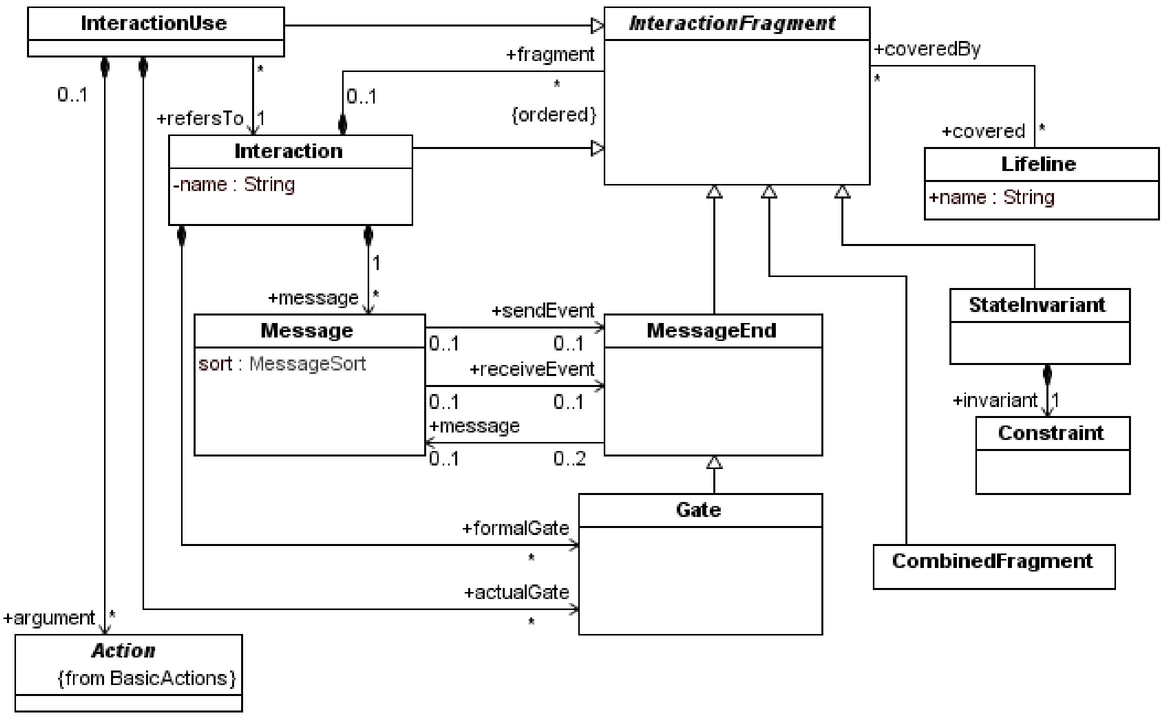


Figure 3.8: Meta model for interaction fragments

Message outside an InteractionFragment with a Message inside the InteractionFragment" (⑥) [Obj09, p. 482]. A StateInvariant defines a Constraint in terms of an OCL specification. In our models a state invariant is depicted in curly braces on a lifeline.

Now let us consider the further kinds of interaction fragments which we call advanced fragments, as they may cover more than one lifeline of an interaction and comprise a number of basic interaction fragments. Advanced fragments are InteractionUse and CombinedFragment.

An InteractionUse (⑦) is nothing else than a (syntactic) placeholder for the interaction it refers to. Hence an interaction containing one ore more interaction uses is also called a *hierarchical interaction*. It is equivalent to the interaction which is gained by flattening, i.e. by expanding the respective referred interaction at the place an interaction use. Hence with an InteractionUse it is possible to decompose an interaction (cf.,e.g., [Bow06]), or, the other way round, define hierarchical interactions by referencing a previously defined interaction. An interaction use is displayed by a fragment (i.e. a rectangular box within the sequence diagram) with the operator ref in the upper left corner, and the name of the referred interaction in the middle of the box. The name is followed by a list of arguments which must correspond to the parameters of the referred interaction. An interaction use covers all lifelines of the enclosing interaction which appear within the referred interaction. If the interaction use has a set of actualGates (⑥) then these gates must match the formalGates of the referred interaction.

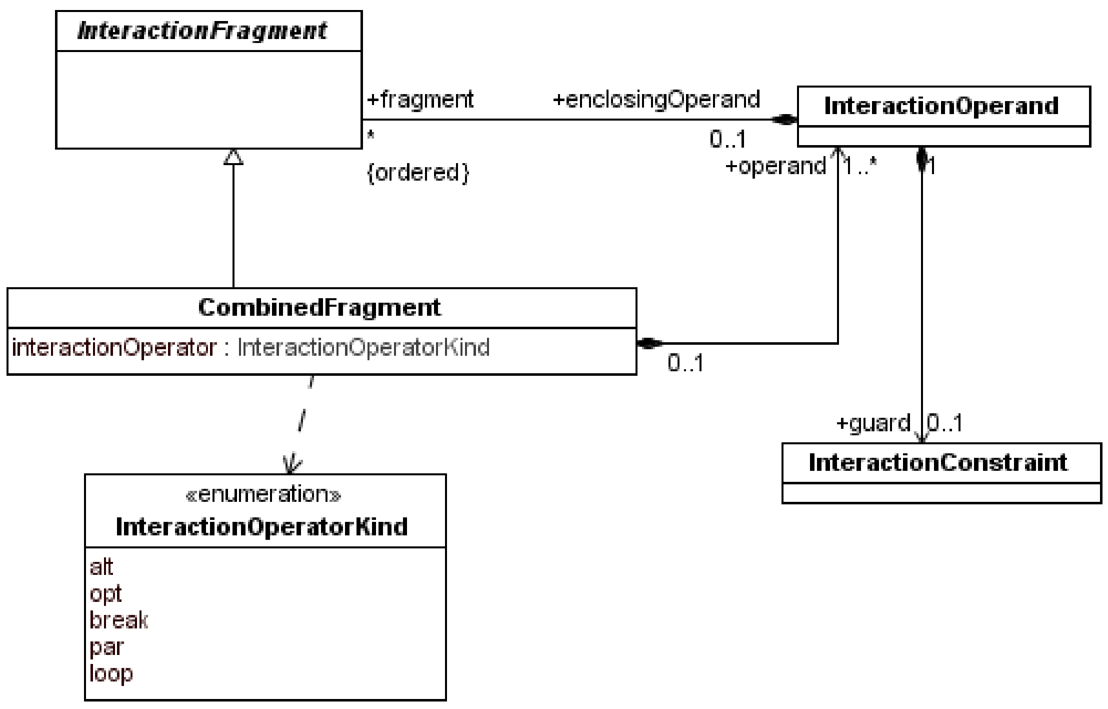


Figure 3.9: Meta model for combined fragments

The meta model excerpt concerning combined fragments is shown in Figure 3.9. A CombinedFragment has an interactionOperator which is one of the enumeration literals defined in InteractionOperatorKind[5]. A combined fragment comprises one or more InteractionOperands, each of which contains a (partially) ordered set of InteractionFragments and may be guarded by an InteractionConstraint. The kind of operator defines the semantics of the fragment; for a semantics definition we refer to the UML superstructure specification for an informal definition [Obj09, p. 470], and to Cengarle and Knapp [CKM09] for a formal definition.

The interaction operator also determines the number of interaction operands as well as the kind of guards for the single operands. A par fragment does not allow for a guard at all; an opt and a break fragment has exactly one operand with one guard in form of a Boolean OCL expression. While an opt fragment represents an optional behaviour, a break fragment is intended to capture an exceptional situation (like, e.g., a missing file). If the condition of the break fragment results to true, the execution continues with the interaction within this fragment, whereas the interactions after the fragment are ignored. Unlike the UML standard we assume that the remainder of the whole interaction is ignored and not only the remainder of the embedding interaction fragment. This approach of exception handling has been proven to be sufficient for our purposes, because the execution of a simulation always terminates after

[5]In contrast to the UML specification we omit the operator kinds critical, neg, assert, strict, seq, ignore, and consider which are not relevant for our modelling purposes.

an exception is raised. A more sophisticated treatment of exceptions is, for example, proposed in [HH06].

An alt fragment may have two or more operands each of them having a guard in form of a Boolean OCL expression. The last operand may have the guard else which denotes the negation of the conjunction of all other guards of the fragment. We require that in each alt fragment exactly one guard results to true; thus we avoid indeterminism. Finally we have to consider the loop fragment which models sequential iteration of the contained interaction fragments. A loop fragment comprises exactly one interaction operand. While the UML specification allows only for specifying a lower and an upper bound for the number of iterations, we extend this by allowing also OCL expressions as guards which iterate over a set or sequence of objects (⑧). The number of iterations is then given by the size of the set or sequence respectively.

Combined fragments are presented in a sequence diagram by a rectangular box which stretches across the lifelines covered by the fragment. The interaction operator is contained in a pentagonal box in the upper left corner, a guard is presented within square brackets. Interaction operands are separated by a dashed horizontal line.

Similar to other software development methodologies (like, e.g. there is one presented in [Rum04]) we use OCL constraints within sequence diagrams in different situations. The constraints aim to enforce an unambiguous implementation of the operations from the model. In our methodology OCL expressions may occur in the following situations.

- A *state invariant* (⑨) is typed in curly braces on a lifeline. An invariant concerning a particular attribute or variable must hold for the remainder of the interaction or until a new state invariant concerning the respective attribute or variable appears. A state invariant is seen as an instruction for the implementor of the corresponding operation; the implementor is responsible to fulfil the state invariant (this should always be possible).

- A *precondition* (⑩) of an operation is written in curly braces above a call message with the keyword pre. The caller of the message is responsible to fulfil the precondition. If the precondition is not true, the operation is not blocked, but its behaviour is undefined and may lead to divergence. This conforms to the classical interpretation of preconditions, like, e.g. in Z (cf. [MBD00]).

- A *postcondition* (⑪) is placed on a lifeline typically before the respective operation call returns. To distinguish a postcondition from a state invariant it carries the keyword post. The usage of extended OCL expressions (including @pre) is allowed in postconditions. A postcondition directly below the head of a lifeline is supposed to be the postcondition of the constructor of the object (⑫).

- An *enable condition* (⑬, for a definition see above) is also placed on the lifeline, typically at the beginning of an operation call; it is marked with the keyword enable. The implementor of the operation is responsible to block the execution of the operation until the enable condition becomes true.

In general there exists no dependency between the precondition and the enable condition of an operation. The keyword enable is borrowed from CSP-OZ [Fis97], where an operation may comprise an *enable schema* which defines the guard of the operation; i.e. the operation call is blocked if the enable condition is false. This conforms to our interpretation of the enable condition. However, as the sequence diagrams in our approach are more implementation-oriented (rather than specification-oriented) the enable condition is positioned *after* the message call on the lifeline where the operation is executed. That means, the operation may be called, but is immediately blocked if the enable condition is false. In contrast, in a process-algebraic specification the transition corresponding to the message call is blocked.

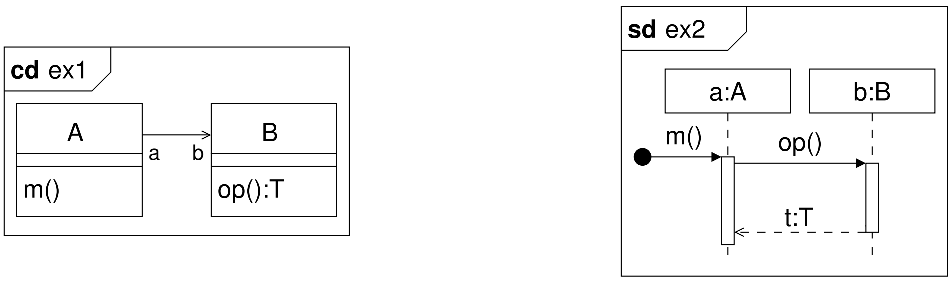


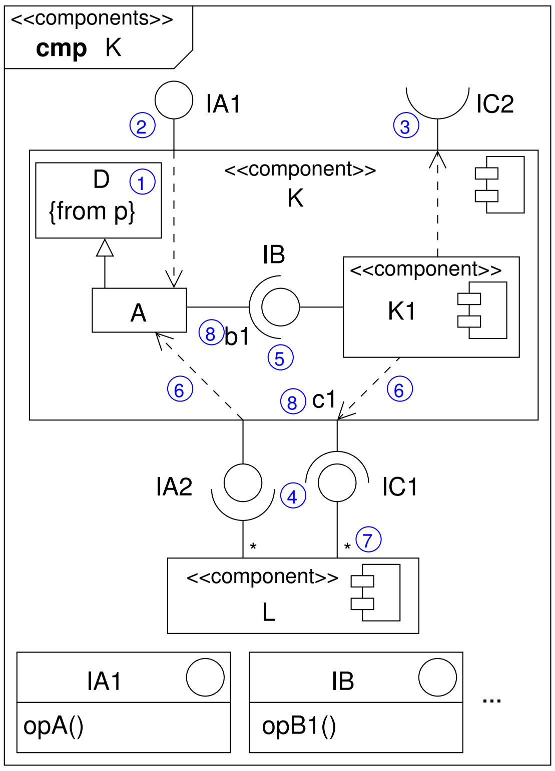
Figure 3.10: Example for specifying a synchronous message call by means of a sequence diagram

In the end let us say some words about specification of behaviour with OCL. Although it is possible to express calling (and returning) of a (synchronous) message in a postcondition this is quite involved as the following example shows. Consider the situation in the class diagram on the left hand side of Figure 3.10. If we want to specify that the operation op of class B is called within the operation m, and op returns a value t of type T, we do this by means of the sequence diagram on the right hand side of the figure. While an asynchronous message call can be specified by means of the OCL *isSent* operator (denoted as ^), for a synchronous message call the OCL *message* operator (^^) is necessary (cf. [FM04]). An OCL specification utilising this operator and expressing the situation of Figure 3.10 then could read as follows:

```
context A::m()
post: let messages : Sequence(OclMessage) = b^^op() in
    messages->size() = 1 -- op was sent once
  and
    messages->first().hasReturned() -- op returned
  and
    messages->first().result() = t -- t is the return value
```

3.1.3 Component and Package Diagrams

On the components level we introduce components and packages for structuring the classes of the design model in order to obtain a component-based system architecture. A component is depicted in a component diagram by a rectangular box with the stereotype «component» and the UML component symbol; a package is notated in the usual UML notation within a package diagram. Figure 3.11 shows an example of a component and a package diagram. The features denoted by circled numbers will be explained in the following by means of the UML meta model. Rules for the creation of component diagrams within our methodology are provided in Section 3.2.3.



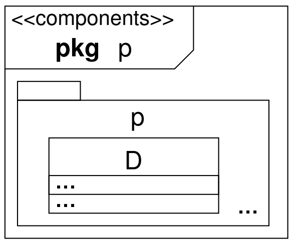


Figure 3.11: Sample component (left) and package (right) diagram

The UML specification distinguishes between two kinds of components, basic components and packaging components. While the first one is intended to model executable elements of a system, the latter one focuses on defining a component as a coherent group of elements and is therefore our means of choice. The excerpt of the UML meta model defining packaging components is depicted in Figure 3.12. A Component has a name and consists of a set of PackageableElements. See Figure 3.13 for some concrete subtypes of this abstract class; notice that Component is a subtype of Class and hence a subtype of PackageableElement which permits the definition of hierarchical (or nested) components. Moreover, a component may import elements – modelled by the set of ElementImports, each of which referencing again a packageable element. That means, that each element referenced by a component is either owned by this component or imported. However, we permit only importing from packages that are defined outside of a component. An element which is imported from a package p is

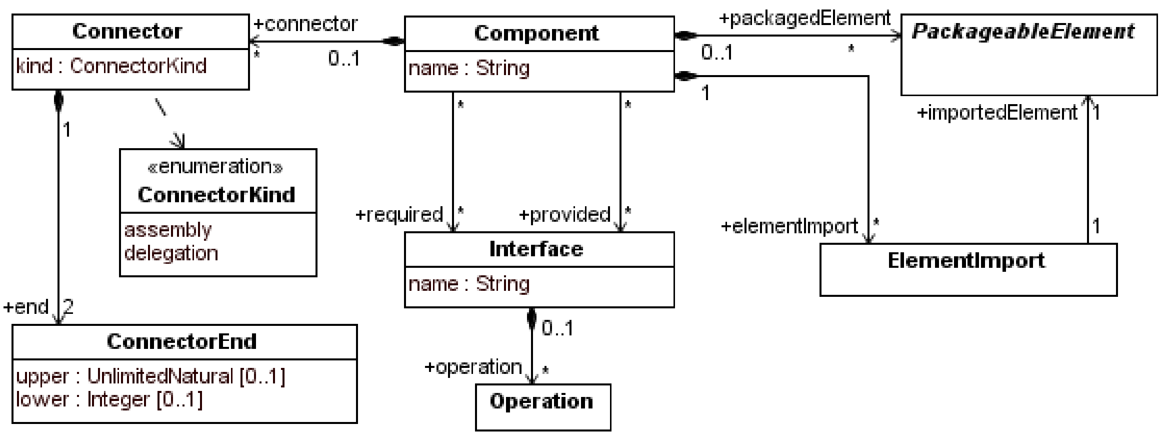


Figure 3.12: Meta model for components

denoted by the property {from p} (①). Elements of other components may only be accessed via the interfaces of the component.

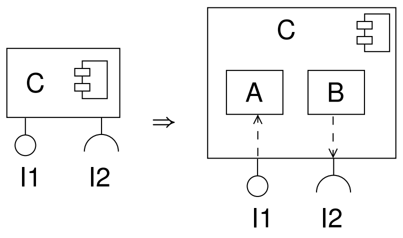
A component possesses a set of provided (②) and required (③) Interfaces, each interface defining a set of Operations. Furthermore, a component comprises a set of Connectors; there are two kinds of connectors, *assembly connectors* and *delegation connectors*. While an assembly connector associates the provided or required interfaces of a component with a required or provided interfaces of another component on the same hierarchy level (④), or with elements of the enclosing component (⑤), a delegation connector connects the provided or required interfaces of a component to the elements of this component that realize or require them respectively (⑥).[6]

In our setting a connector has always two ConnectorEnds (the UML standard permits two or more). A connector end may be equipped with a multiplicity (⑦) like an association end (default value is 1); we require that the multiplicity is equal for each connector end which is attached to the same connectable element. As an extension to the UML standard we permit to attach a role name to a connector end (⑧). Within our methodology this is for example useful for tracing back a connector to the association it originates from.

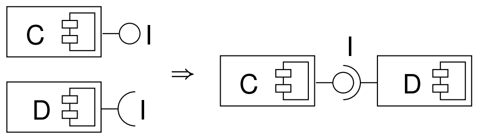
Of course there is a relationship between the connectors and the provided and required interfaces of a component. To achieve consistency of a component model, the following conditions must hold (the diagrams illustrate the preceding condition):

- for each provided (required) interface of a component C there is a delegating connector from (to) an element of the component which realises (uses) the interfaces;

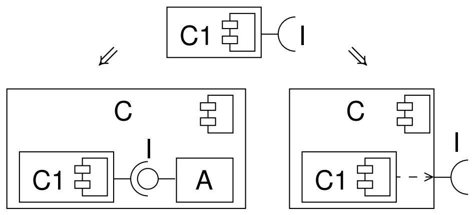
[6]Unfortunately the modelling tool MAGICDRAW which is used for creating most of the UML diagrams in the thesis does not allow for drawing delegating connectors like in Figure 3.11 – except for ports which we do not use. In this case we help ourselves by connecting a provided or required interface of the component directly with the internal element which realises or uses the interface.



- if an interface *I* is a provided interface of a component *C* and also a required interface of a component *D* then there is an assembly connector between the components *C* and *D* defined by the interface *I*;



- if *I* is a provided (required) interface of a component *C*1 which is a subcomponent of a component *C* then there is
  - either an assembly connector between *C*1 and an element of the containing component *C*,
  - or *I* is also a provided (required) interface of *C* and there is a delegating connector to (from) *C*1 from (to) *I*.



Packages (cf. Figure 3.13) are – similarly to packaging components – intended to group coherent model elements, but – in contrast to components – do not possess interfaces and connectors.

3.1.4 OCL Enhancements

In this section we define some OCL operations which we will use later in order to express a kind of reflection within OCL. The operation getType on the type OclAny shall return the concrete type of an instance of OclAny and is defined as follows.

```
context OclAny
def: getType() : OclType
    = OclType.allInstances()->any(t:OclType | self.oclIsTypeOf(t))
```

Moreover, we introduce the operation getName():String on the type OclType which shall provide the name of the given type.

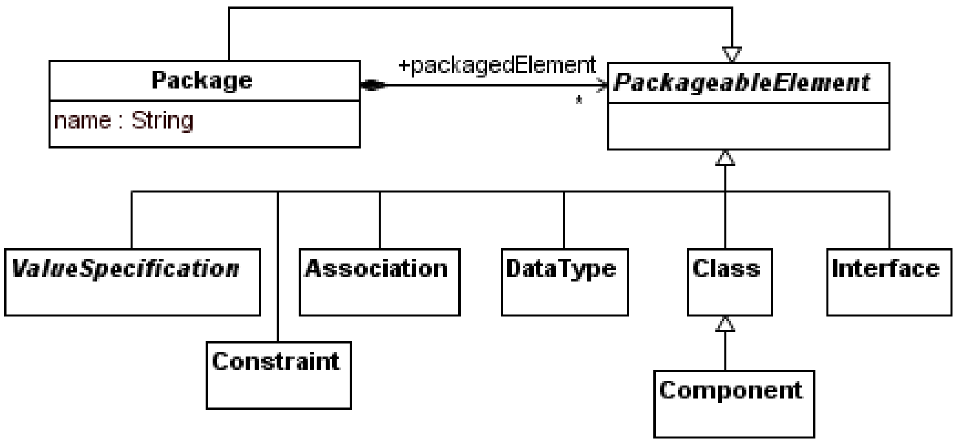


Figure 3.13: Meta model for packages and packageable elements

3.2 Abstraction Levels and Refinement

In this section we describe the abstraction levels used by the methodology which involve the following refinement steps (denoted by $\rightsquigarrow$):

$$\text{requirements} \rightsquigarrow \text{design} \rightsquigarrow \text{components}$$

It is out of the scope of this thesis to provide a formal refinement relation between the different levels; instead we use an intuitive notion of refinement.

The following sections are separated into several parts: first, we describe the modelling constraints of the respective level, then the consistency conditions between structural and behavioural models and, finally (except for the requirements level) the refinement relation from the previous level to the actual one.

3.2.1 Requirements

On the requirements level the concepts and requirements of the respective view are stated in terms of class diagrams and sequence diagrams.

Modelling Constraints

Structural Model. The structural model consists of a set of classes and associations. As the requirements level is the most abstract level in our methodology we do not have to specify many details. Usually we do not model attribute types and visibility, navigability of associations (all association ends are considered navigable), inheritance, operations, abstract or active classes, but we may specify invariants which the requirements classes must adhere to. As an example for a class diagram on the requirements level consider the class diagram on the top of Figure 3.14.

Behavioural Model. The requirements for the behaviour of the classes are given by interactions in terms of sequence diagrams. On the requirements level we usually do not specify OCL constraints; the names of lifelines consist only of the type of the represented class. Activation bars are not shown to simplify matters. For an example of a sequence diagram on the requirements level consider the upper diagram in Figure 3.15.

Consistency between Structural and Behavioural View

The problem of consistency between structural and behavioural view is treated only informal here: we say that a behavioural model is consistent with its corresponding structural model (on the requirements level) if

- the type of each lifelines in a sequence diagrams corresponds to a class of the structural model, and
- messages are exchanged only between lifelines the corresponding classes of which have an association in between.

A formal approach dealing with the consistency between class and sequence diagrams can be found in [BB07] or in [EEEP08].

3.2.2 Design

On the *design* level the models from the requirements level are refined. The design model is again expressed in terms of class diagrams and sequence diagrams for the structural and the behavioural view respectively.

Modelling Constraints

Structural View. For the structural view we have the following constraints. A type must be specified for each attribute of a class (including multiplicity, default is 1), each association must be directed (bidirectional associations are directed in both directions) and for each association end a multiplicity (default 1) and for each navigable association end a role name must be specified. Role names have to be distinct in the sense that all association ends with the same role name are attached to the same class.

Query operations, i.e. operations which do not change the state of its object, are marked with the property {query}. Moreover, a visibility has to be specified for each element of a class (default is private for properties and public for operations).

To improve readability often an overview diagram is provided which shows the classes (without details) and their relations. The details of the classes are then depicted in separate diagrams.

Behavioural View. For sequence diagrams the following modelling constraints hold. The kind of each message call must be specified by an appropriate arrow type (either asynchronous call, synchronous call, create, delete or reply). A synchronous message call is always followed by a corresponding reply message. Activation bars are displayed on the design level.

Pre- and postconditions of operations can be attached to the diagram in a note, or directly inserted into the diagram (cf. Figure 3.15). Remind that we stated implicit postconditions of getters and setters. The return value (including its type) is attached to a reply message. The name of a lifeline must contain an object name (usually derived from a role name of a class diagram).

Consistency between Structural and Behavioural View

Consistency conditions are the following. The types of the lifelines must correspond to the classes of the design level structural model. The object name of a lifeline is derived from the (distinct) role name in the class diagram. As already stated for the requirements model, messages are exchanged only between lifelines the corresponding classes of which have an association in between, but additionally, the signatures of operation calls must fit to the operation signatures in the class diagrams.

Refinement

The refinement relation between the requirements and the design level is described informally and illustrated by means of an example.

Structural View. A structural model SM_2 is a refinement of a structural model SM_1, denoted by $SM_1 \rightsquigarrow SM_2$, if

- for each class A in SM_1 there exists a non-empty set Cor_A of corresponding classes in SM_2,
- for each attribute of a class in SM_1 there is a corresponding attribute in one of the refining classes of that class,
- for each association in SM_1 between two classes A and B there exists an association in SM_2 between two classes in Cor_A and Cor_B respectively, and
- for each invariant Inv there exists an invariant Inv' with $Inv' \Rightarrow Inv$.

For associations additionally the following condition must hold: if a multiplicity was already specified in SM_1, then its value in SM_2 may not be contradictory. Note that the refinement relation as defined above in particular allows for the renaming and the splitting of classes, as well as for the specification of new classes and associations in SM_2. The refinement relation between class diagrams is given by the restriction of the above definition to the elements of the structural model contained in the class diagram.

As an example consider the class diagrams in Figure 3.14 where the lower diagram is a (possible) refinement of the upper one. The dashed arrows denote the corresponding classes, associations and invariants. Obviously the refinement relation as defined above is fulfilled.

The operations of the classes in the lower diagram are mainly derived from the sequence diagrams of the requirements level (like e.g. opA, opB, opB1) or are getters and setters for attributes (getX, getY). The association between the requirements classes A and B has been refined to a qualified association between A and B1. Note that a qualifier reduces the multiplicity on the opposite end to 0..1.

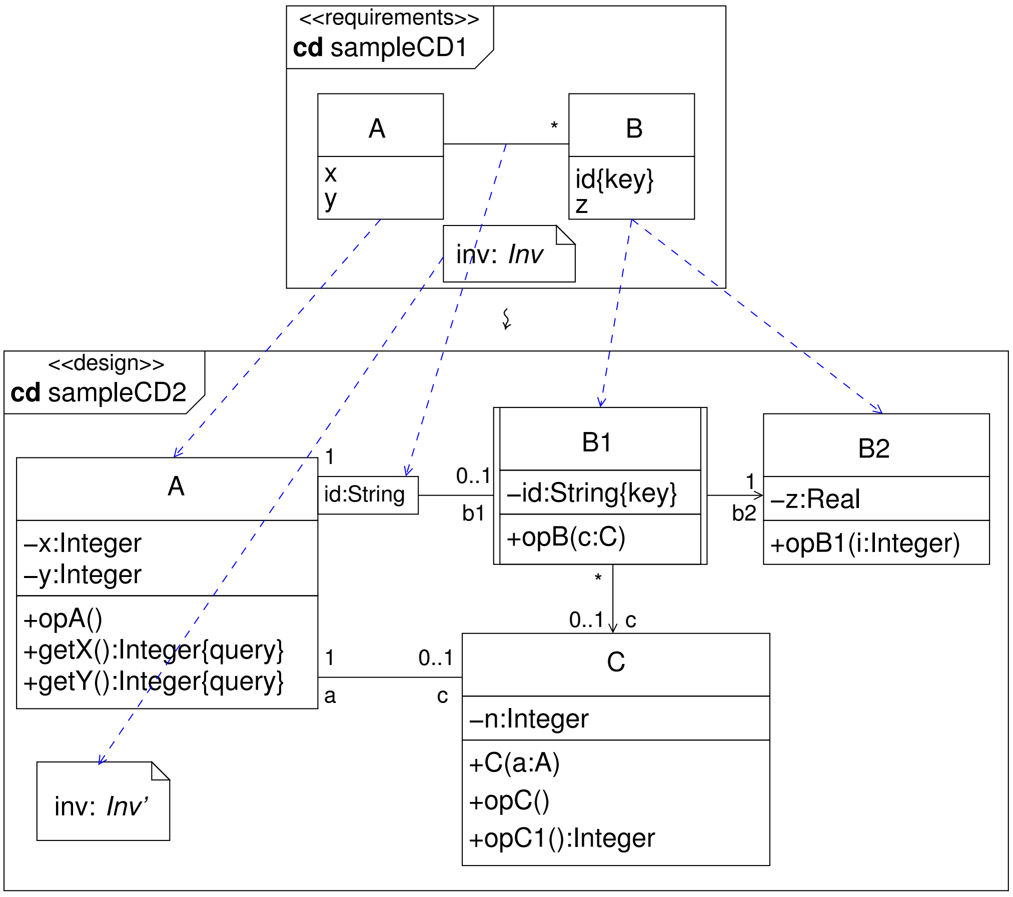


Figure 3.14: Example of a class diagram refinement

Behavioural View. Let SD_1 and SD_2 be sequence diagrams with corresponding structural models SM_1 and SM_2 respectively. Then SD_2 is a refinement of SD_1, denoted by $SD_1 \rightsquigarrow SD_2$, if

- $SM_1 \rightsquigarrow SM_2$,

- for each lifeline L in SD_1 with type $A \in SM_1$ there is a non-empty set Cor_L of corresponding lifelines in SD_2 where the type of each $L\prime \in Cor_L$ is in Cor_A , and
- for each interaction fragment in SD_1 exists a corresponding interaction fragment in SD_2, and the (partial) order of the interaction fragments of SD_1 is reflected by the (partial) order of the corresponding interaction fragments in SD_2.

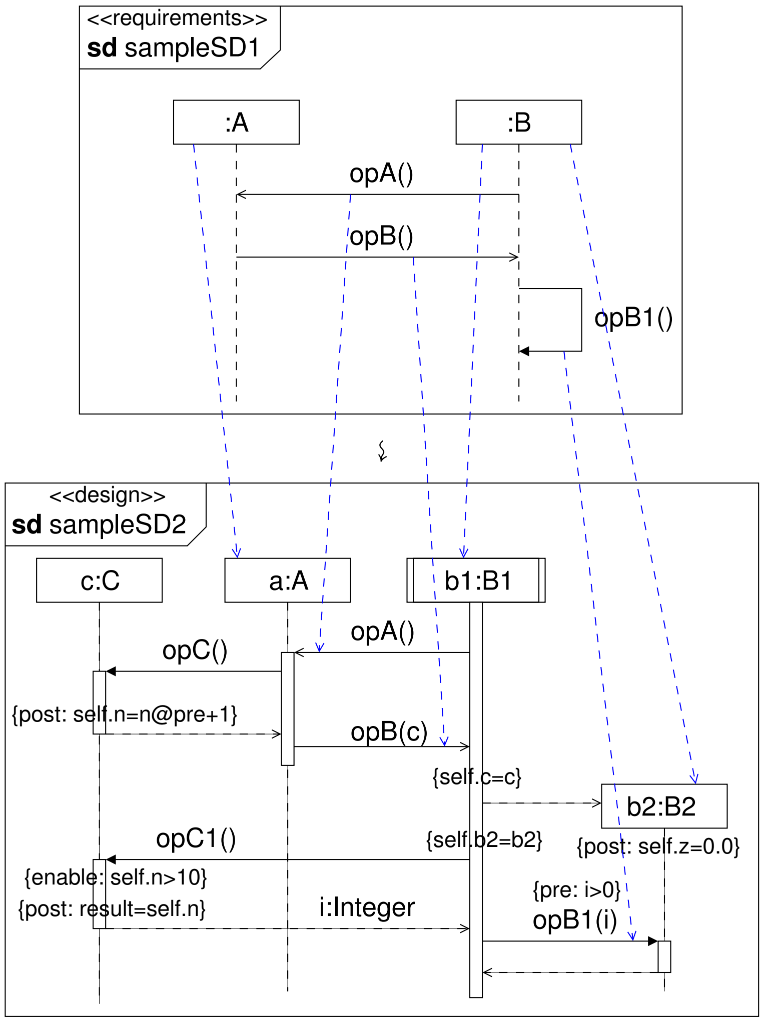


Figure 3.15: Example of a sequence diagram refinement

An example of a sequence diagram refinement is depicted in Figure 3.15. The sequence diagrams in this figure correspond to the class diagrams in Figure 3.14. The dashed arrows denote again the refinement relation of the single entities. It is easy to proof that the refinement conditions stated above are fulfilled by these diagrams. Note that in sampleSD2 there are two lifelines b1:B1 and b2:B2 which are a refinement of the lifeline :B.

Several approaches for refinement of sequence diagrams can be found in the literature, most of them expressing refinement on the semantics level (like, e.g., the approach of Cengarle and Knapp [CKM09], or the STAIRS approach [HS03, HHRS05]).

3.2.3 Components

On the *components* level we introduce components and packages as structural elements to which the elements of the design level are assigned.

Modelling Constraints

For the modelling on the components level we utilise the partitioning of the systems by (hierarchical) components. That means, that for each component occurring in the system a component diagram is depicted which shows

- the component as a rectangular box with the stereotype «component», the UML component symbol and the component name,
- the provided and required interfaces in ball-and-socket notation attached to the component,
- the internal structure of the component given by a component diagram, comprising subcomponents without their internal structure,
- the connectors between the provided and required interfaces of the component and the internal elements which realize or utilise them respectively.
- Moreover, the interfaces which occur in assembly connectors in the internal structure are shown in detail within the component diagram.

For each package of the system a package diagram is depicted showing the internal structure of the package. For component diagrams as well as for package diagram the following holds: usually classes are depicted with details, but if the diagrams thereby would grow too large, the details of a class are depicted in a separate diagram.

The whole system can be modelled, when we apply the following top-down approach. We consider the system itself as a component of the top-most level (we omit the component frame for the system). The internal structure of this component comprises the system architecture. Then recursively each subcomponent is depicted in a separate diagram. As an example consider Figures 3.16 and 3.17. While the former diagram shows a possible system architecture, consisting of the components K and L, the latter diagram shows the details of component K. For modelling the entire system we would also have to detail the components L, K1 and each possible subcomponent of them. We omit this for lack of space here.

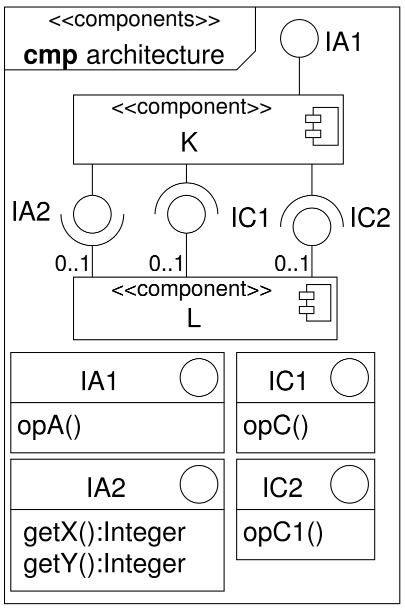


Figure 3.16: Sample architecture component diagram

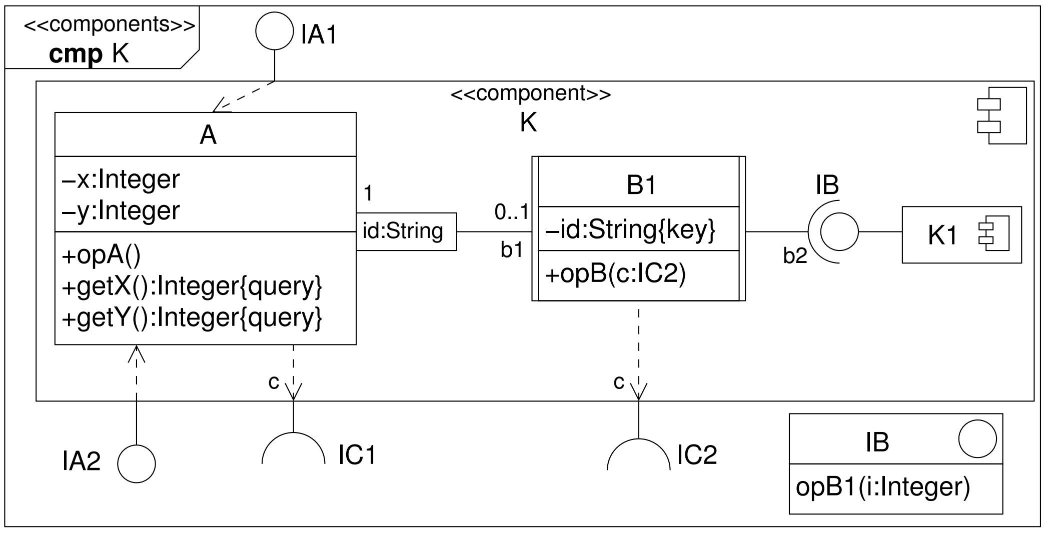


Figure 3.17: Sample component diagram

Consistency between Structural and Behavioural View

The transition from the design to the components level in the structural view is just a partitioning of the design level elements into components and packages (see the refinement section below), i.e. no new modelling elements are introduced which would have influence on the behaviour. Therefore we can reuse the sequence diagrams from the design level, and the consistency problem is reduced to the consistency problem of the design level. To build a component level sequence diagram one would just have to replace the class type of a lifeline with an interface type where necessary.

Refinement

We define the refinement relation between the structural model *SM* of the design level and the component model *CM* which consists of a set of components and packages. *CM* is a refinement of *SM*, denoted by $SM \rightsquigarrow CM$, if

- for each class of *SM* there is an equivalent class (for a definition see below) in either a component or a package of *CM*,
- for each association between two classes *A* and *B* in *SM* there is
  - an equivalent association (for a definition see below) in *CM*, if the equivalent classes of *A* and *B* both reside in the same component, or at least one of them reside in a package;
  - a sequence of connectors between the equivalent classes of *A* and *B* if these classes reside in different components (cf. Figure 3.18; the operations of the interfaces are derived from the messages exchanged between *A* and *B*)

We still have to provide the definition of equivalent classes and associations. Two classes *A* and *B* are equivalent, if they have the same name, the same attributes, operations and associations, at which, however, an association may be replaced by a connector in the following way:

- in the case of a navigable association (i.e. the association end at *B* is navigable) by a connector associated to a required interface which specifies at least the operations called from *A* on *B*;
- in the case of a non-navigable association (i.e. the association end at *A* is navigable) by a connector associated to a provided interface which specifies at least the operations called from *B* on *A*; in this case *A* implements this interface.

Two associations are equivalent if their respective ends have the same multiplicity, navigability and role name.

For an example of a refinement between the design level and the components level consider Figure 3.19. Obviously the refinement relation defined above is fulfilled by the two diagrams.

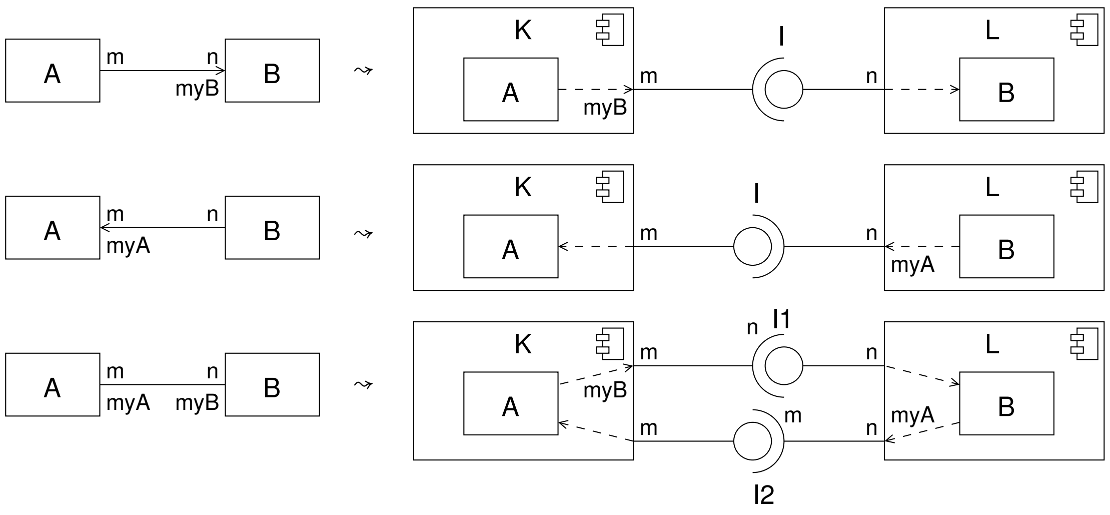


Figure 3.18: Refinement of associations by connectors taking into account role names (myA, myB), multiplicities (m, n) and navigability

The refinement of the associations which result in connectors are pointed out by the dashed arrows. Note that the depiction in this Figure does not match the rules stated by the modelling constraints, as too many details are displayed within the components.

The main tasks of the system developer on the component level are first to identify appropriate components and packages and then assign each class and each interface[7] of the class design level to one of the components or one of the packages. The rule to decide whether a class has to be put into a component or into a package is the following: classes which represent data are put into packages, all other classes and interfaces into components. An interface which has no implementor within the component becomes a required interface of the component.

Once this has been done, the transition from the class design to the component design level is straight forward: each association between classes assigned to different components has to be resolved by appropriate connectors; all other relations remain unchanged and are assigned to the respective component.

In the end let us say some words about the advantages of using components in our approach, although the system seems to be fully specified already on the design level. The main advantages are first, that we obtain a clear picture of the system architecture of the complex system under development, and second, that a single component can easily be replaced by another component with the same interfaces. Furthermore, by using interfaces between components the access to the public operations of classes can be restricted, which is not the case when using associations only.

[7]Interfaces obtained from the class design level may not be confused with the provided/required interfaces of the component

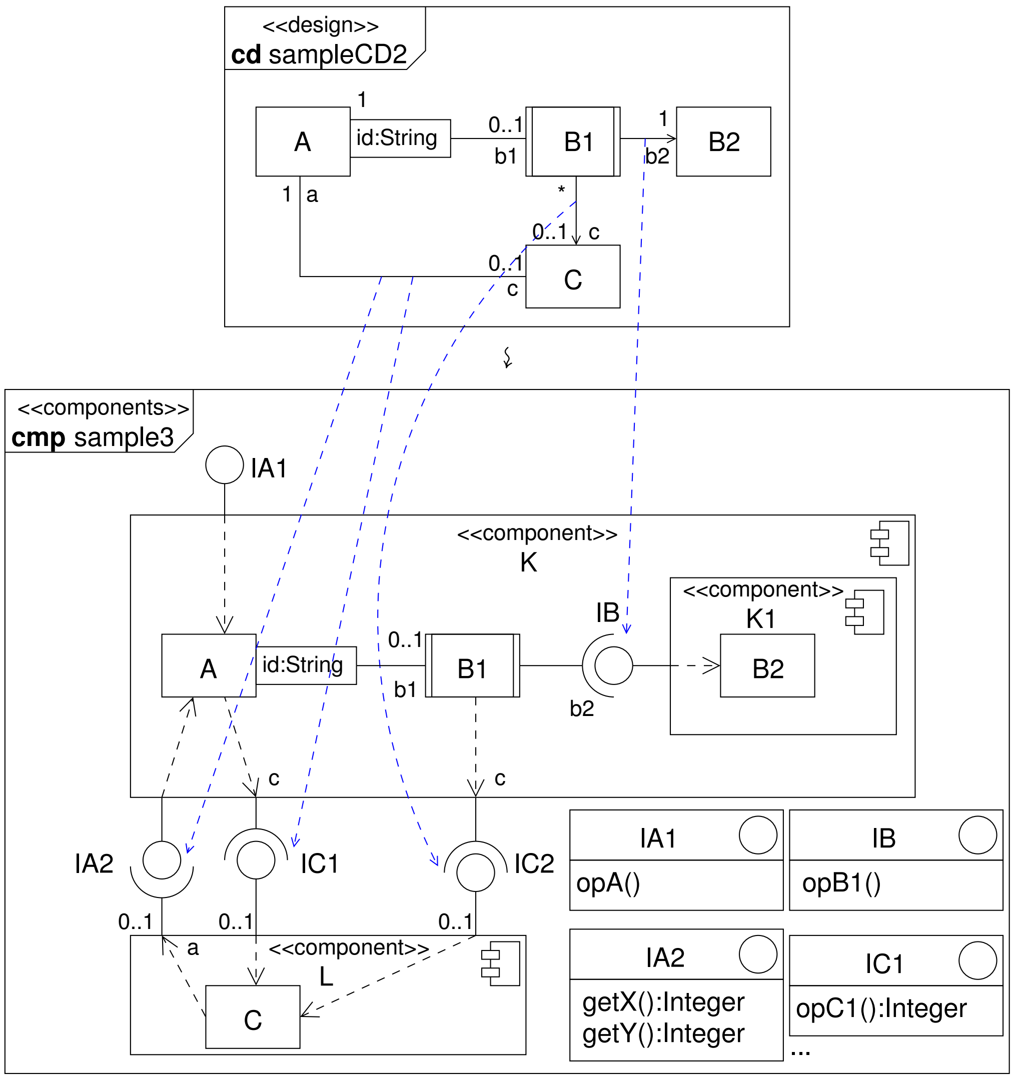


Figure 3.19: Example of a refinement from the design to the components level

3.3 View-based Development by Extension

In this section we sketch how we develop a complex system by considering different views of the system on the abstraction levels discussed in the previous section. Regarding to Figure 3.1 on page 22, this means that we now consider the extension relation $\hookrightarrow$ where the base view is on the left hand side. As mentioned before, the relation $A \hookrightarrow B$ denotes an *extension* of A by B, where A and B are each a structural or a behavioural model element.

The idea of this development method is, that we first construct a base architecture of the system under consideration by analysing some basic system use cases and stating some basic requirements, and then, emanating from this, develop an architecture for each identified view of the system. Of course, in order to be able to obtain a system architecture by simply integrating the different views, some prerequisites have to be considered when identifying the base and the different views, hence this is one of the crucial tasks of system development.

In short, the extension relation is a special kind of refinement. In contrast to the refinement relation defined in the previous section it does not permit renaming and splitting of classes, thus allowing only for introducing new classes and associations or extending the existing ones by adding new attributes, associations or operations. Moreover, operations may be extended by increasing their parameter list. These restrictions of the general refinement relation are necessary in order to facilitate the integration of the views later on.

In the remainder of this section we define the extension relation for the structural, behavioural and component model respectively and sketch the methodology by means of an example where a base model is extended into two different views $V1$ and $V2$ on each abstraction level. The integration of the views will be discussed in Section 3.4.

As the extension relation is quite more restrictive than the refinement relation we can be a bit more precise when providing a definition. We thereby regard the UML meta model presented in Section 3.1. In the following we use the dot notation known from OCL to refer to properties of modelling elements (e.g. $C.name$ to refer to the name of class C). All used properties are defined in the meta model. If the multiplicity of a property is greater than one, then the reference to it denotes a set of the respective type (e.g. $C.attribute$ denotes the set of attributes of class C).

3.3.1 Structural Model

According to the UML meta model presented in Section 3.1.1 we have to consider *classes*, *data types* and *associations* for defining the structural model. Properties are either attributes of classes or association ends. The generalisation relation is implicitly given by the superclass property of a class. The structural model hence consists of a set of classes $\mathcal{C}$, a set of data types $\mathcal{D}$, and a set of associations $\mathcal{A}$. In the following we define the extension relation of structural models.

Definition 3.1. A structural model $SM_2 = (\mathcal{C}_2, \mathcal{D}_2, \mathcal{A}_2)$ is an *extension* of a structural model $SM_1 = (\mathcal{C}_1, \mathcal{D}_1, \mathcal{A}_1)$, denoted by $SM_1 \hookrightarrow SM_2$, if

- for all classes $C \in \mathcal{C}_1$ there exists exactly one class $C' \in \mathcal{C}_2$ with $C \hookrightarrow C'$ which means that
  - $C.name = C'.name$,
  - $C.isAbstract = C'.isAbstract$,
  - $C.isActive = C'.isActive$,
  - $C.superclass.name = C'.superclass.name$,
  - $C.attribute.name \subseteq C'.attribute.name$[8]
  - for all attributes $a \in C.attribute$ and $a' \in C'.attribute$ the following holds: $a.name = a'.name \Rightarrow a = a'$ (for the definition of equality on properties see below)
  - $C.operation.name \subseteq C'.operation.name$
  - for all operations $o \in C.operation.name$ and $o' \in C'.operation.name$ the following holds: $o.name = o'.name \Rightarrow o \hookrightarrow o'$ (for the definition of extension of operations see below),
- for all data types $D \in \mathcal{D}_1$ there exists exactly one data type $D' \in \mathcal{D}_2$ with $D.name = D'.name$ and if D is a(n)
  - ordinary data type then D' is an ordinary data type and $D \hookrightarrow D'$ as defined for classes;
  - enumeration type then D' is an enumeration type and $D.literal \subseteq D'.literal$;
  - primitive type then D' is a primitive type.
- for all associations $A \in \mathcal{A}_1$ there exists exactly one association $A' \in \mathcal{A}_2$ with $A.end_i = A'.end_i$, $i \in \{1,2\}$, where end_i denote the association ends (for the definition of equality on properties see below).

We still have to provide the definition of equality on properties.

Definition 3.2. A property p *is equal to* a property p', denoted $p = p'$ if

- $p.name = p'.name$
- $p.type = p'.type$
- $p.lower = p'.lower$
- $p.upper = p'.upper$
- $p.isKey = p'.isKey$
- $p.visibility = p'.visibility$

[8] $C.attribute.name$ is an abbreviation for $\{a.name | a \in C.attribute\}$

- $p.qualifier$ is equal to $p'.qualifier$

A set P of properties is equal to a set P' of properties, if $|P| = |P'|$ and for each $p \in P$ there exists a $p' \in P'$ with $p = p'$.

The extension relation on structural models also requires an extension relation on operations which we provide in the following.

Definition 3.3. An operation o' is an *extension* of an operation o, denoted by $o \hookrightarrow o'$, if

- $o.name = o'.name$
- $o.type = o'.type$
- $o.lower = o'.lower$
- $o.upper = o'.upper$
- $o.isQuery = o'.isQuery$
- $o.isAbstract = o'.isAbstract$
- $o.visibility = o'.visibility$
- $o'.preCondition \Rightarrow o.preCondition$
- $o'.postCondition \Rightarrow o.postCondition$
- $o'.enableCondition \Rightarrow o.enableCondition$
- $o.parameter.name \subseteq o'.parameter.name$,
- for all parameters $p \in o.parameter$ and $p' \in o'.parameter$ the following holds: if $p.name = p'.name$ then
  - $p.type = p'.type$,
  - $p.lower = p'.lower$,
  - $p.upper = p'.upper$,

In other words, an operation may be extended only with respect to its parameter list.

As an example for the extension of class diagrams on the requirements level consider Figure 3.20. The diagrams are arranged so that the base is on the top of the figure and the views V_1 and V_2 on the bottom left and bottom right respectively. Moreover, the new elements are coloured in the view diagrams: blue in view V_1 and red in view V_2.

Note that the diagrams are related to the base or one of the views by their names in that `^base`, `^V1` or `^V2` is contained in the name. A diagram name like `sample1^V1` should be read

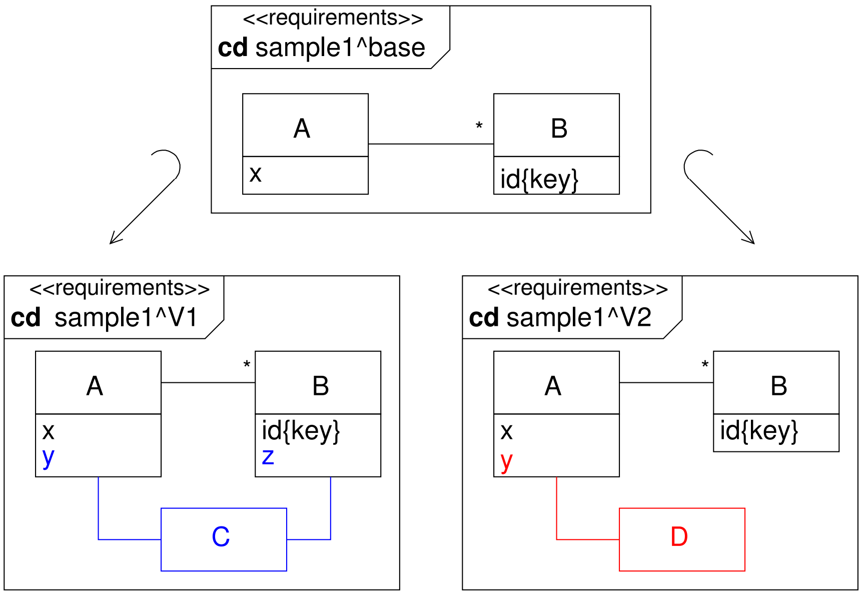


Figure 3.20: View-based development on the requirements level – structural view

as “diagram sample1 under the view V1”. One can easily proof that the extension relation $\hookrightarrow$ is fulfilled for the respective diagrams.

Figure 3.21 shows a possible extension of class diagrams on the design level. Unfortunately, due to their size, they cannot be arranged similarly to the requirements models, but rather one below the other. Again the elements added or extended in the different views are coloured, additionally new or extended elements are marked with the property {new} or {ext} respectively. While the colouring is only done for explanation purposes here the marking with properties is also used in the application of the methodology. The properties facilitate the integration of the views as we will see later.

Finally note that in the example, besides the extension relation which relates the diagrams on a particular abstraction level, the refinement relation defined in the previous section relates corresponding diagrams on different views.

3.3.2 Behavioural Model

The behavioural model is given by a set of interactions, where an interaction is described by a sequence diagram.

Definition 3.4. A behavioural model BM_2 is an *extension* of a behavioural model BM_1 if

- the corresponding structural models are in extension relation, and
- each sequence diagrams describing an interaction of BM_1 has a corresponding extended sequence diagram in BM_2.

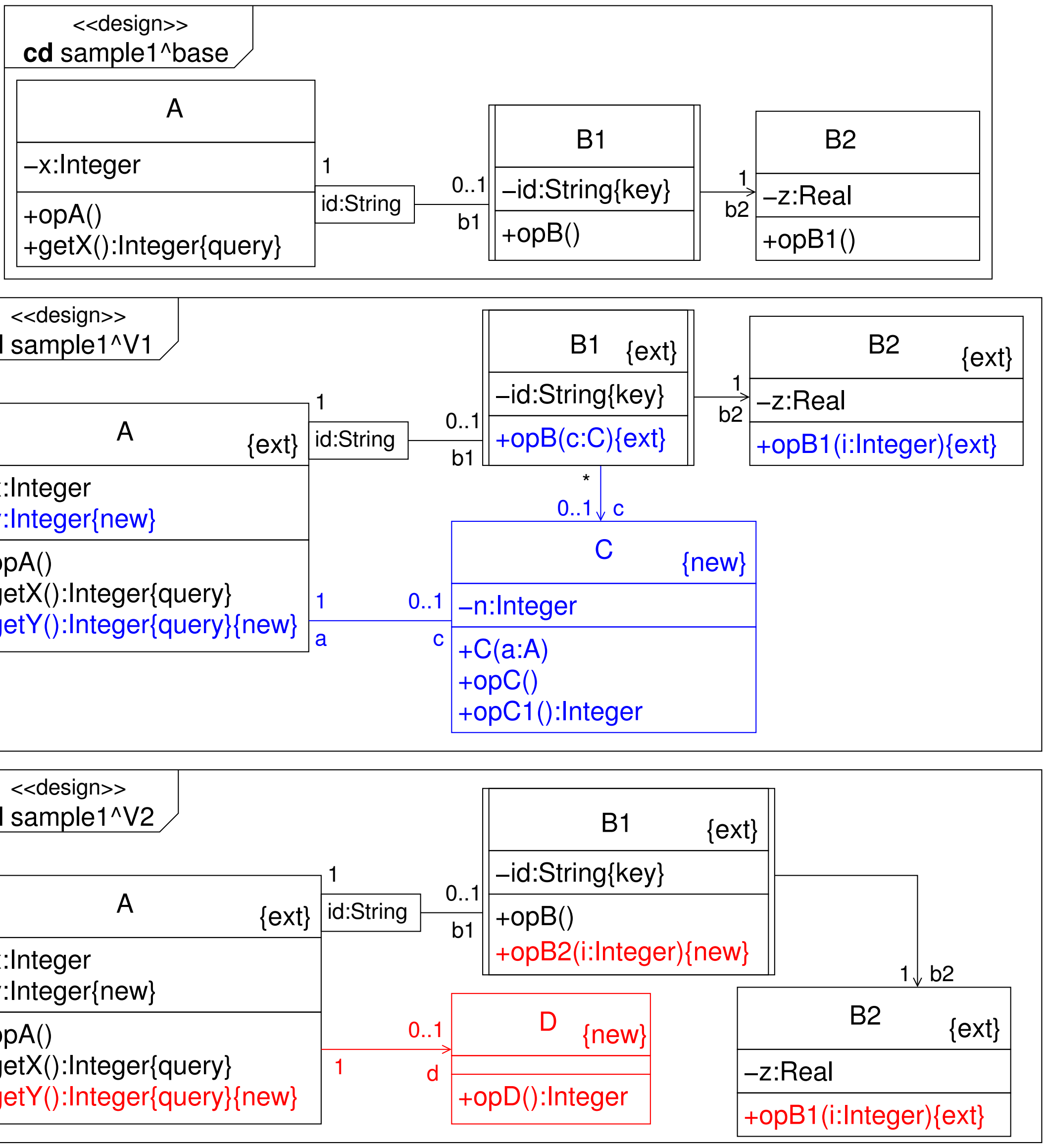


Figure 3.21: View-based development on the class design level – structural view

A sequence diagram *SD* consists of (cf. Figure 3.7)

- a set $\mathcal{L}$ of *lifelines* representing the instances taking part in the interaction,
- a set $\mathcal{M}$ of *messages*, and
- a (partially) ordered set $\mathcal{F}$ of *interaction fragments*.

An *interaction fragment* $F \in \mathcal{F}$ (cf. Figure 3.8) is either

- a message end (i.e. a *send event* or a *receive* event),
- a state invariant,
- a *combined fragment* which in turn consists of a set of interaction operands each referring to a (partially) ordered set of interaction fragments, or
- an interaction use.

With these preliminaries we are able to define the extension relation on sequence diagrams.

Definition 3.5. A sequence diagram $SD' = (\mathcal{L}', \mathcal{M}', \mathcal{F}')$ is an *extension* of a sequence diagram $SD = (\mathcal{L}, \mathcal{M}, \mathcal{F})$, denoted by $SD \hookrightarrow SD'$, if

- $\mathcal{L} \hookrightarrow \mathcal{L}'$, that means for each $L \in \mathcal{L}$ there exists an $L' \in \mathcal{L}'$ with
  - $L.name = L'.name$,
  - $L.selector = L'.selector$, and
  - $L.represents \hookrightarrow L'.represents$ (where $\hookrightarrow$ is the extension relation on classes as defined above);
- $\mathcal{M} \hookrightarrow \mathcal{M}'$, that means for each message $M \in \mathcal{M}$ there exists a message $M' \in \mathcal{M}'$ with
  - $M.sort = M'.sort$
  - if $M.signature$ exists, then $M'.signature$ exists and $M.signature \hookrightarrow M'.signature$ (where $\hookrightarrow$ denotes the extension relation on operations as defined above);
- $\mathcal{F} \hookrightarrow \mathcal{F}'$, that means for each interaction fragment $F \in \mathcal{F}$ there exists an interaction fragment $F' \in \mathcal{F}'$ with $F \hookrightarrow F'$ (for the definition of $\hookrightarrow$ see below) and $\hookrightarrow$ preserves the partial order of $\mathcal{F}$, i.e. if $F_1 \leq F_2$ in $\mathcal{F}$, and $F_1 \hookrightarrow F_1'$, $F_2 \hookrightarrow F_2'$ then $F_1' \leq F_2'$ in $\mathcal{F}'$.

The extension relation $\hookrightarrow$ on interaction fragments is defined as follows. $F \hookrightarrow F'$ if F and F' are of the same type (message end, state invariant, combined fragment, or interaction use), $F.covered \hookrightarrow F'.covered$ (i.e. the set of covered lifelines are in extension relation as defined above) and if F is a(n)

- message end, then $F.message \hookrightarrow F'.message$,

- state invariant, then $F' \Rightarrow F$,
- combined fragment, then
  - $F.interactionOperator = F'.interactionOperator$,
  - for each $o \in F.operand$ exists an $o' \in F'.operand$ with
    - $\forall f \in o.fragment\ \exists f' \in o'.fragment : f \hookrightarrow f'$, and
    - $o'.guard \Rightarrow o.guard$,
- interaction use, then $F.refersTo \hookrightarrow F'.refersTo$.

Note that in the case of combined fragments and interaction uses the definition is recursive. The property *refersTo* of an interaction use refers to an interaction, i.e. a sequence diagram.

As an example for the extension of sequence diagrams consider Figure 3.22.Again, the different colours denote the extensions made in the different views.

3.3.3 Component Model

A component model CM is given by a set of components $\mathcal{K}$[9] and a set of packages $\mathcal{P}$. We assume that the consistency conditions concerning connectors and provided/required interfaces of components stated in Section 3.1.3 hold.

Definition 3.6. A component model $CM' = (\mathcal{K}', \mathcal{P}')$ is an *extension* of a component model $CM = (\mathcal{K}, \mathcal{P})$, denoted $CM \hookrightarrow CM'$, if

- for each component $C \in \mathcal{K}$ there is a component $C' \in \mathcal{K}'$ with $C \hookrightarrow C'$, which means
  - for each element $p \in C.packagedElement$ there exists $p' \in C'.packagedElement$ with $p \hookrightarrow p'$ (where $\hookrightarrow$ is the extension relation for the specific type of p),
  - for each imported element $i \in C.elementImport$ there is an imported element $i' \in C'.elementImport$ with $i \hookrightarrow i'$ (where $\hookrightarrow$ is the extension relation for the specific type of i),
  - for each provided interface $ip \in C.provided$ there exists $ip' \in C'.provided$ with $ip \hookrightarrow ip'$ (for the definition of $\hookrightarrow$ see below),
  - for each required interface $ir \in C.required$ there exists $ir' \in C'.required$ with $ir \hookrightarrow ir'$,
  - for each connector $c \in C.connector$ there exists a connector $c' \in C'.connector$ with $c.kind = c'.kind$ and the respective connector ends of c and c' are in extension relation;

[9] We use $\mathcal{K}$ for the set of components, because $\mathcal{C}$ is already occupied for the set of classes.

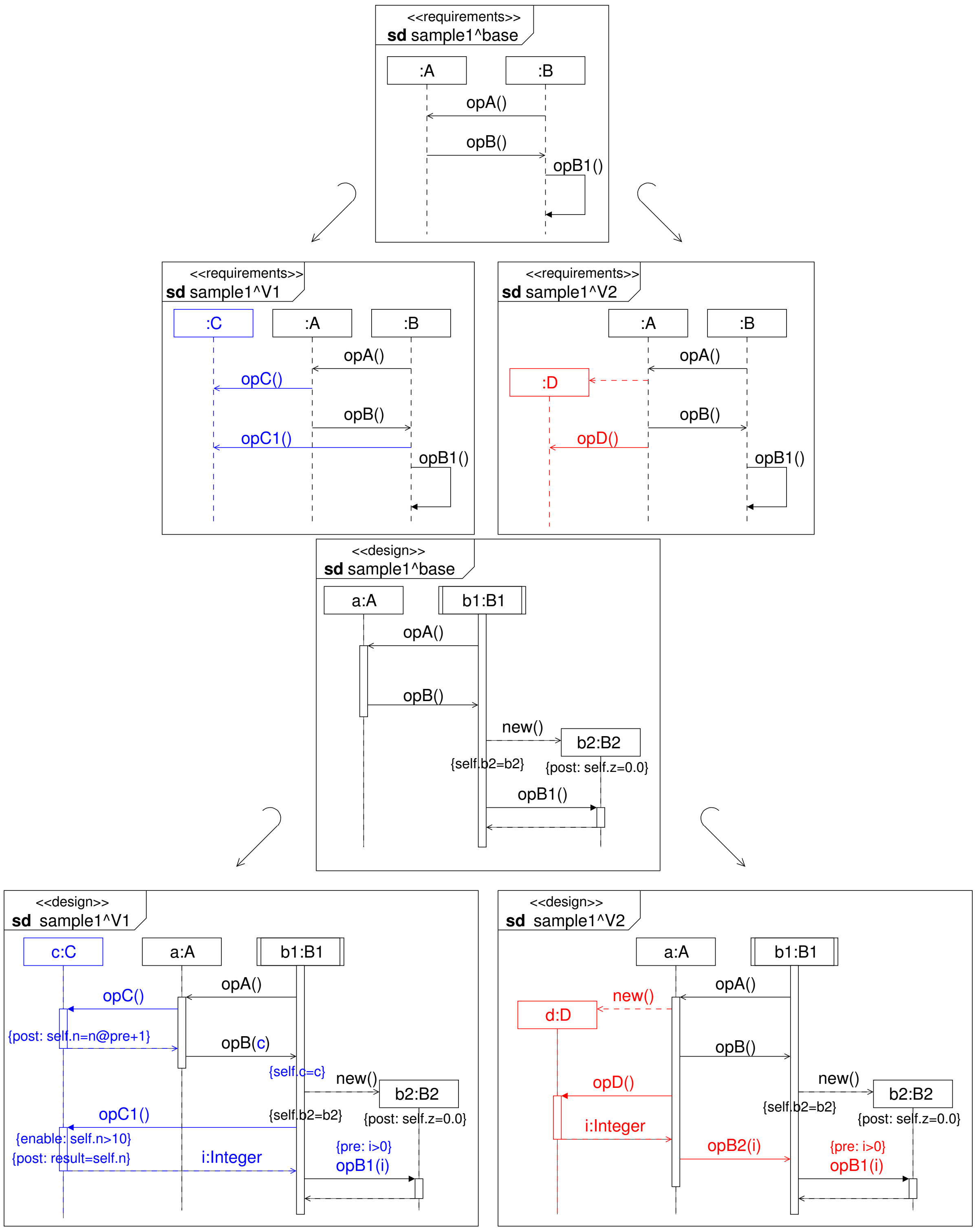


Figure 3.22: Extension of sequence diagrams – on the requirements level (top) and on the design level (bottom)

- for each package $P \in \mathcal{P}$ there is a package $P' \in \mathcal{P}'$ with $P \hookrightarrow P'$ which means that for each element $p \in P.packagedElement$ there is an element $p' \in P'.packagedElement$ with $p \hookrightarrow p'$ (where $\hookrightarrow$ is the extension relation for the specific type of p).

The extension relation $\hookrightarrow$ on interfaces is a restriction of the extension relation on classes considering operations only.

As an example for the extension relation of components consider Figures 3.23 and 3.24. While the former shows an architecture diagram which is extended in the views V_1 and V_2, the latter focusses on the extension of the component K. Again, the extensions are coloured blue (view V_1) and red (view V_2) respectively.

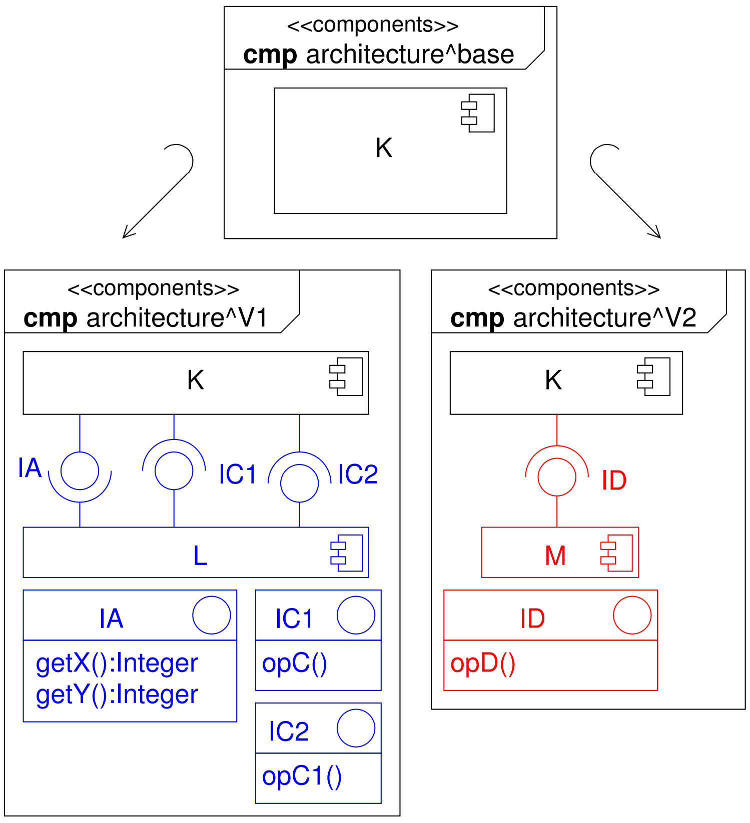


Figure 3.23: View-based development on the components level – architecture view

3.4 Integration of the Views

In this section we explain how the models of different views are integrated under consideration of their common base. The basic idea is to embed the component model of each view into an

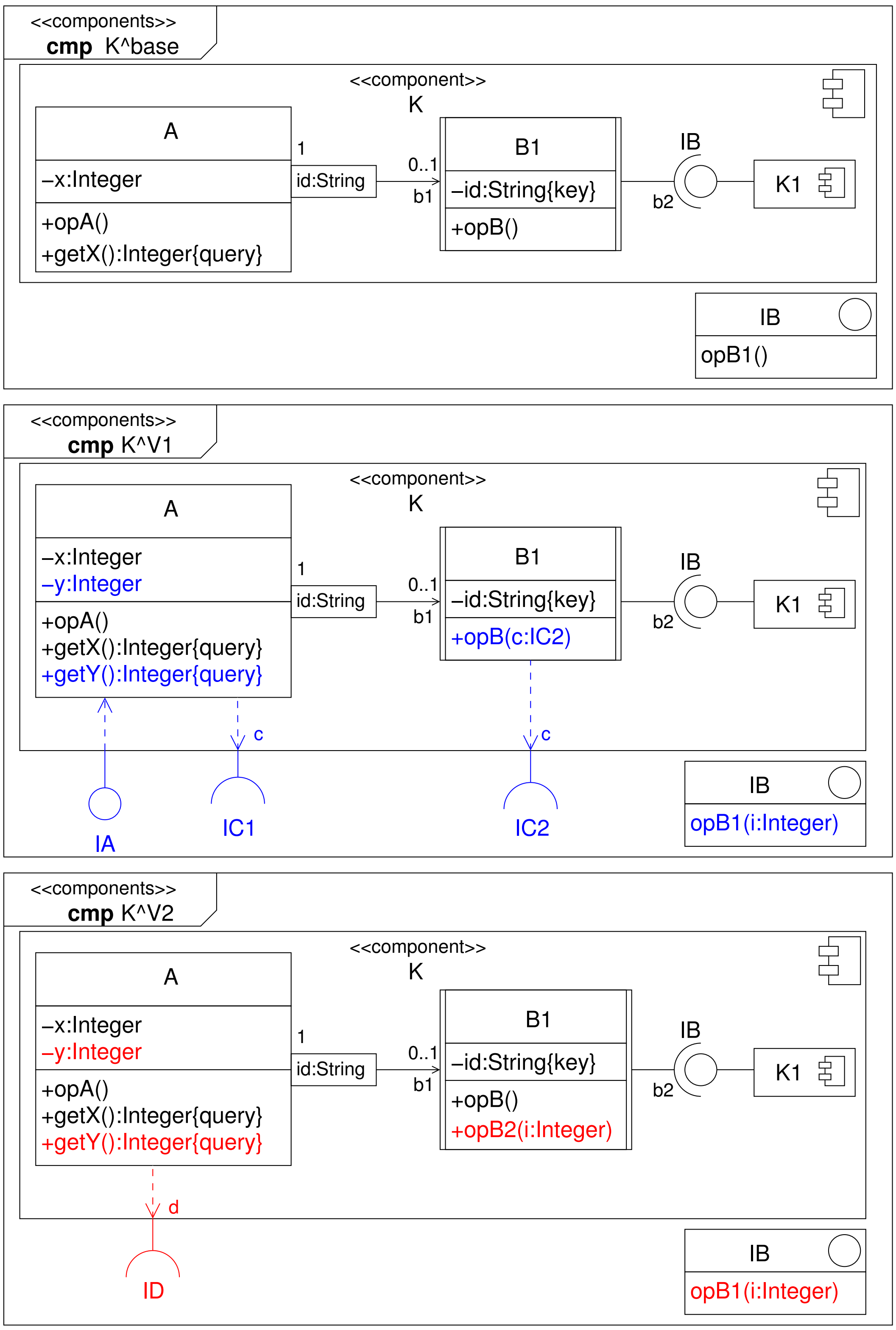


Figure 3.24: View-based development on the components level – component K

integration component model in an appropriate way, so that the integration is an extension of each view:

$$\begin{tikzcd}
& \mathit{base} \arrow[dl, hook'] \arrow[dr, hook] & \\
V_1 \arrow[dr, hook] & & V_2 \arrow[dl, hook'] \\
& \mathit{integration} &
\end{tikzcd}$$

Note that this construction corresponds to a push-out in category theory. Hence the integration procedure is automatable and its result is unique except for renaming. In the case of more than two different views integration is performed step-by-step, i.e. the first view is integrated with the second, the third with the integration of the first and the second and so on. Thereby the order of integration does not matter as we will see later.

In the following the structural integration of component diagrams (Section 3.4.1) and the behavioural integration of sequence diagrams (3.4.2) is defined and illustrated by means of the example elaborated in the previous sections. Finally we briefly discuss the problem of integrability.

3.4.1 Structural Integration

In the structural view we have to integrate two component models CM_1 and CM_2 with a common base model CM_0, i.e. we have to integrate components, packages and their respective elements. The integration process described in the following is widely adopted from the UML *package merge* procedure which is explained, e.g. in [HKKR05]. The integrated component model corresponds to the effective contents of the receiving package in a package merge, while the component models of the views to be integrated regard to a merged package each.

Suppose we want to integrate two models M_1 and M_2 under a common base model M_0, i.e. we have $M_0 \hookrightarrow M_1$ and $M_0 \hookrightarrow M_2$. Then for the integration we have to consider in particular equal elements of M_1 and M_2, and the elements of M_0 which are extended by corresponding elements of M_1 and M_2 respectively. While the first matter can be coped with renaming of equal elements, the second matter has to be treated individually.

Having this in mind, the integrated model M_{int} consists of

- the elements which have been added in M_1, i.e. $M_1 \smallsetminus M_0$,
- the elements which have been added in M_2 and are different from those of M_1, i.e. $M_2 \smallsetminus M_1$,
- renamed equivalents of the elements of M_2[10] which are equal to elements of M_1, i.e. $M_2 \cap M_1 \smallsetminus M_0$, and

[10]One could also rename the elements of M_1 instead of those of M_2.

- the elements of M_0 integrated with their corresponding elements in M_1 and M_2.

In the following we provide a mathematical definition for the integration of sets, which is useful for defining the integration of modelling elements later on.

Definition 3.7. Let A_0, A_1 and A_2 be subsets of a domain D with $A_0 \subseteq A_1$ and $A_0 \subseteq A_2$. Then we define the integration of A_1 and A_2 under the common base A_0 by

$$A_{int} := A_1 \overset{A_0}{\uplus} A_2 := A_1 \setminus A_0 \cup A_2' \cup A_0',$$

where

$$A_0' := \{\mu(a) \mid a \in A_0\},$$

$\mu : A_0 \to D$ is an injective mapping which is specifically defined for each kind of set (called the *integration mapping*),

$$A_2' := A_2 \setminus A_1 \cup \{\rho(a) \mid a \in (A_1 \cap A_2) \setminus A_0\},$$

and $\rho : D \to D$ denotes an injective renaming function so that $\rho(a) \notin A_1 \cap A_2$ for all $a \in A_1 \cap A_2$.

We are now able to define the integration of components, packages and their elements (classes, interfaces, associations). To facilitate the notation let in the following $i \in \{0,1,2\}$ be an index variable and for each i-indexed element E_i hold that $E_0 \hookrightarrow E_1$ and $E_0 \hookrightarrow E_2$.

Components. Let $\mathcal{K}_i$ be sets of components. Then the integrated set $\mathcal{K}_{int}$ is given by

$$\mathcal{K}_{int} := \mathcal{K}_1 \overset{\mathcal{K}_0}{\uplus} \mathcal{K}_2,$$

where the integration mapping μ is defined for each $C_0 \in \mathcal{K}_0$ as follows.

- $\mu(C_0).name = C_0.name$,
- $\mu(C_0).required = I_1^{req} \overset{I_0^{req}}{\uplus} I_2^{req}$, where $I_i^{req} := C_i.required$,
- $\mu(C_0).provided = I_1^{prov} \overset{I_0^{prov}}{\uplus} I_2^{prov}$, where $I_i^{prov} := C_i.provided$,
- $\mu(C_0).connector = C_0.connector$,
- $\mu(C_0).packagedElement = P_1 \overset{P_0}{\uplus} P_2$, where $P_i := C_i.packagedElement$,
- $\mu(C_0).elementImport = O_1 \overset{O_0}{\uplus} O_2$, where $O_i := C_i.elementImport$,

Packages. For the integration of packages we resort to the integration of components restricted to the name and the packaged elements (first and fourth item in the above definition).

Classes. Let $\mathcal{C}_i$ be sets of classes. Then the integrated set $\mathcal{C}_{int}$ is given by

$$\mathcal{C}_{int} := \mathcal{C}_1 \overset{\mathcal{C}_0}{\uplus} \mathcal{C}_2,$$

where the integration mapping μ is defined for each $C_0 \in \mathcal{C}_0$ as follows.

- $\mu(C_0).name = C_0.name$,
- $\mu(C_0).superclass = C_0.superclass$,
- $\mu(C_0).attribute = A_1 \overset{A_0}{\uplus} A_2$, where $A_i := C_i.attribute$ and the integration mapping on attributes is the identity,
- $\mu(C_0).operation = O_1 \overset{O_0}{\uplus} O_2$, where $O_i := C_i.operation$ and the integration mapping μ_o is defined for each $o_0 \in C_0.operation$ as follows.
  - $\mu_o(o_0).name = o_0.name$,
  - $\mu_o(o_0).type = o_0.type$,
  - $\mu_o(o_0).lower = o_0.lower$,
  - $\mu_o(o_0).upper = o_0.upper$,
  - $\mu_o(o_0).isQuery = o_0.isQuery$,
  - $\mu_o(o_0).isAbstract = o_0.isAbstract$,
  - $\mu_o(o_0).visibility = o_0.visibility$,
  - $\mu_o(o_0).preCondition = o_1.preCondition \wedge o_2.preCondition$,
  - $\mu_o(o_0).postCondition = o_1.postCondition \wedge o_2.postCondition$,
  - $\mu_o(o_0).enableCondition = o_1.enableCondition \wedge o_2.enableCondition$,
  - $\mu_o(o_0).parameter = P_1 \overset{P_0}{\uplus} P_2$, where where $P_i := o_i.parameter$ and the integration mapping is the identity.

Interfaces. For the integration of interfaces we resort to the integration of classes restricted to the name and the operations (first and fourth item in the above definition).

Associations. Let $\mathcal{A}_i$ be sets of associations. The integrated set $\mathcal{A}_{int}$ is then given by

$$\mathcal{A}_{int} = \mathcal{A}_0 \cup \mathcal{A}_1 \cup \mathcal{A}_2.$$

With the definition above we have in fact that integrated component model CM_{int} is an extension of CM_1 and CM_2, i.e.

$$\begin{tikzcd}
 & CM_0 \arrow[dl, hook] \arrow[dr, hook] & \\
CM_1 \arrow[dr, hook] & & CM_2 \arrow[dl, hook] \\
 & CM_{int} &
\end{tikzcd}$$

which can be easily proven by applying the definition of extension provided in Section 3.3.

The integration process is exemplified in Figure 3.25 for an architecture diagram and in Figure 3.26 for an individual component which contains classes and interfaces. The elements which have been added or extended in different views are coloured blue (for view V_1) and red (for view V_2) respectively. To get the full picture (including extension of the base) one has to consider Figure 3.23 and Figure 3.24 respectively.

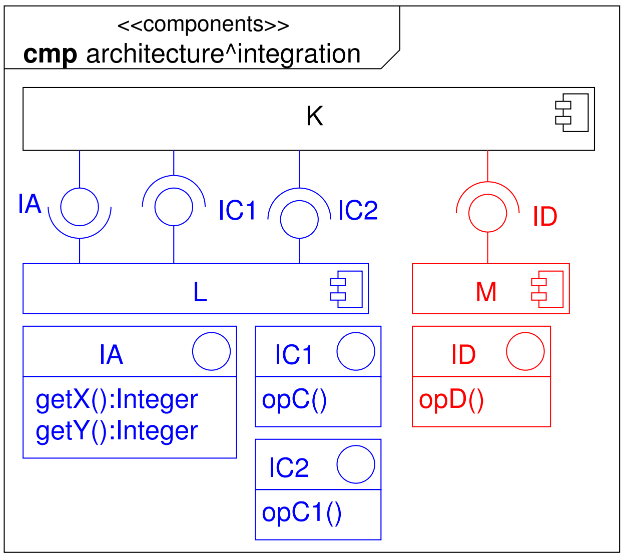


Figure 3.25: Integrated component architecture

3.4.2 Behavioural Integration

Concerning the behavioural view we have to define how two sequence diagrams SD_1 and SD_2 which extend a common base sequence diagram SD_0 are integrated. Note that in our approach no separate sequence diagrams are constructed on the components level as the ones from the

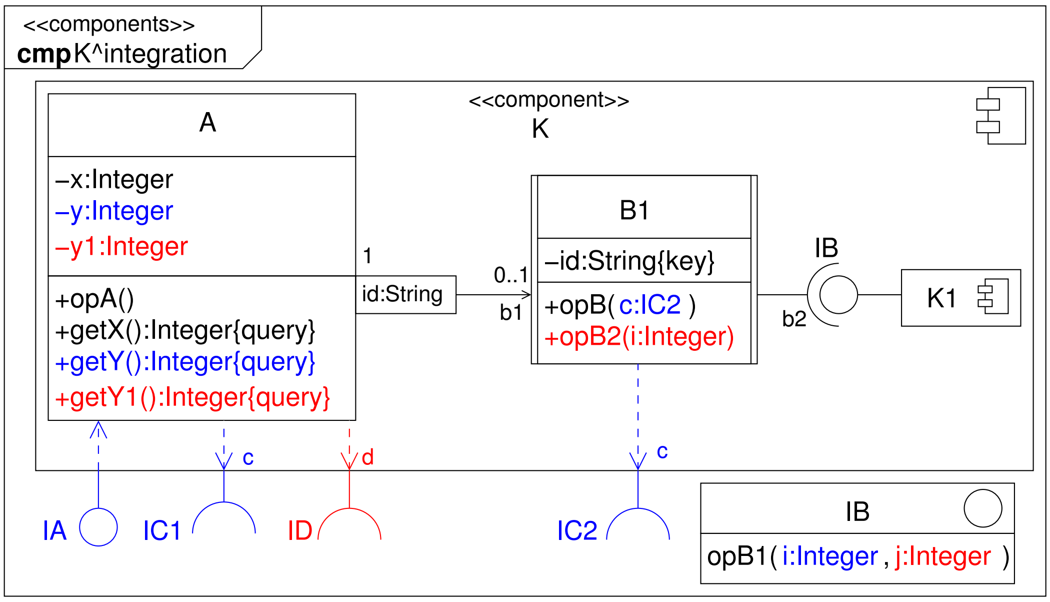


Figure 3.26: Integrated sample component K

design level can be reused (cf. Section 3.2.3). Hence in the following the sequence diagrams correspond to a structural model of the design level.

The integrated sequence diagram SD_{int} is defined as follows:

- the corresponding structural model of SD_{int} is the integration of the structural models of SD_1 and SD_2 as defined in the previous section;
- The set of lifelines of SD_{int} is given by

$$\mathcal{L}_{int} = \mathcal{L}_1 \overset{\mathcal{L}_0}{\uplus} \mathcal{L}_2;$$

- the set of messages of SD_{int} is given by

$$\mathcal{M}_{int} = \mathcal{M}_1 \overset{\mathcal{M}_0}{\uplus} \mathcal{M}_2;$$

- the set of interaction fragments of SD_{int} is given by

$$\mathcal{F}_{int} = \mathcal{F}_1 \overset{\mathcal{F}_0}{\uplus} \mathcal{F}_2$$

  and the partial order defined on $\mathcal{F}_{int}$ (if existing, see below) preserves the partial orders on $\mathcal{F}_1$ and $\mathcal{F}_2$.

According to the definition above, an integrated sequence diagram comprises the lifelines, messages and interaction fragments of its constituent parts. The order of interaction fragments is constructed as follows in order to comply with the requirement of preserving the partial orders: the interaction fragments of the base diagram act as synchronisation points whereas the other interaction fragments of the extension diagrams are arranged in separate operands of a par fragment between the interaction points.

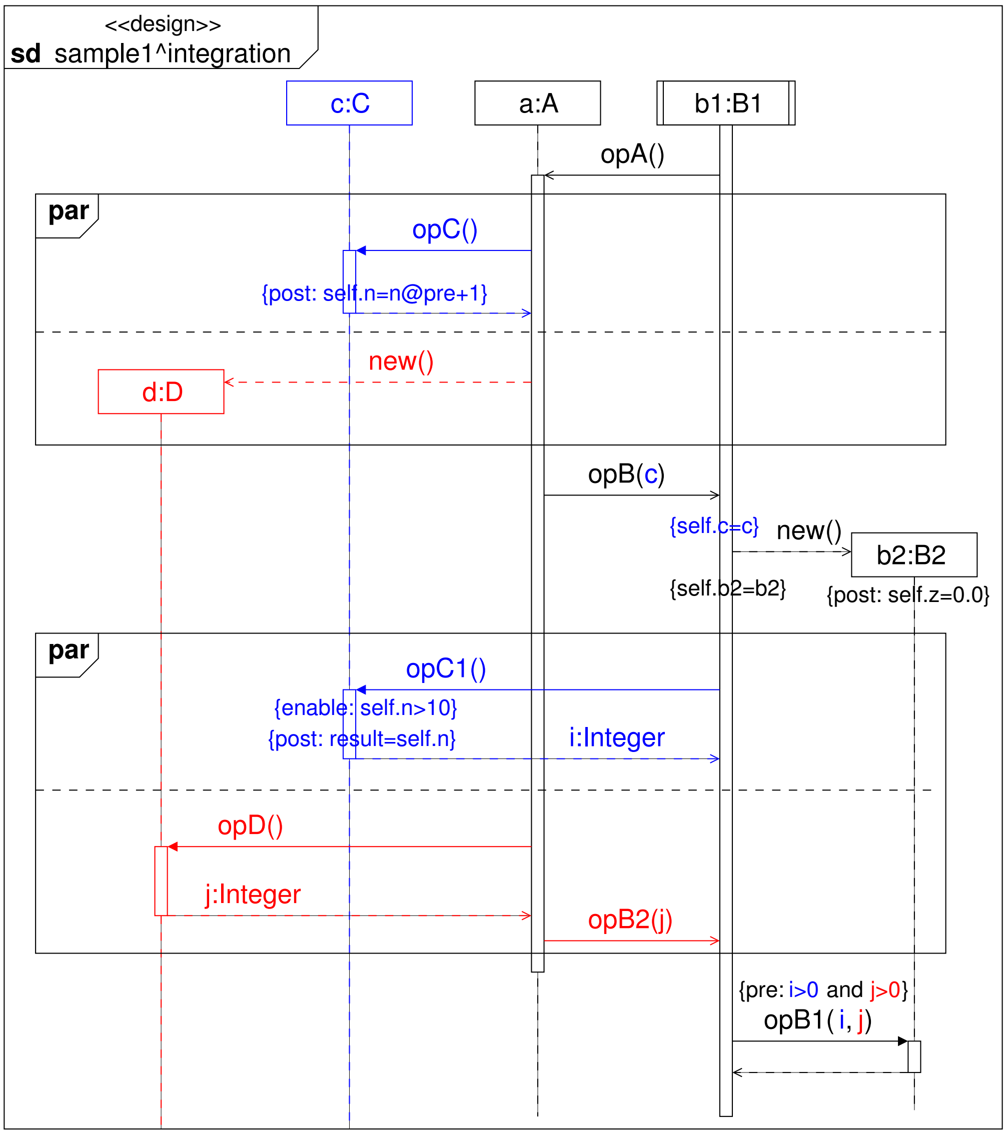


Figure 3.27: Integrated sequence diagram on the design level

As an example consider the sequence diagram sample1^integration in Figure 3.27 which integrates the sequence diagrams in Figure 3.22 (bottom). Note that the colours used for depicting the extensions of different views are retained in the integration in order to illustrate the integration process. While the black elements denote the synchronisation points, i.e. the elements of the base diagram, the blue and red elements are extensions added in the views. If there is an extension in both views between two consecutive synchronisation points then these extensions are put in different operands of a par fragment.

We still have to settle the matter of *integrability*. We say that two sequence diagrams SD_1 and SD_2 are integrable under a common base diagram SD_0 if the resulting sequence diagram SD_{int} as defined above is well-formed, i.e. there exists a partial order on the set of interaction fragments.

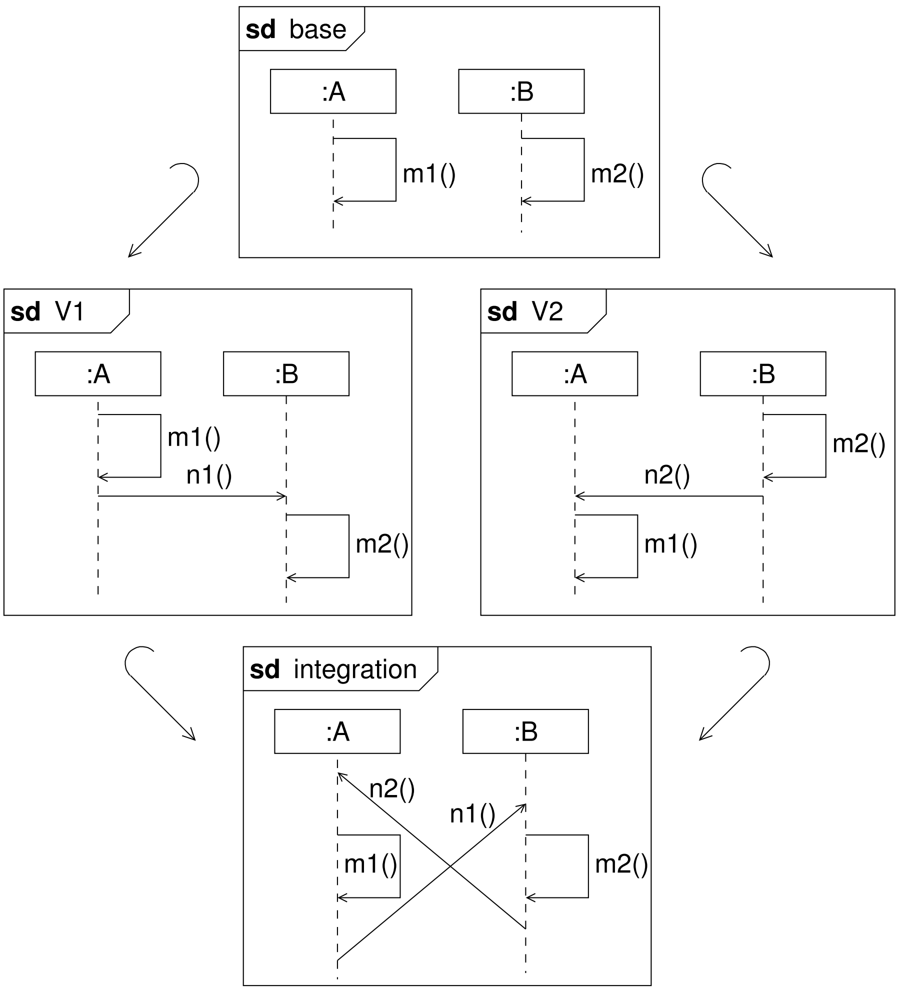


Figure 3.28: Example of a not well-formed integration

Note that the integration of two well-formed sequence diagrams is not always well-formed, as the counterexample in Figure 3.28 demonstrates (cf. [KCH04]): obviously the order of interaction fragments in the resulting diagram is not a partial order. It is out of the scope of this thesis to provide necessary and sufficient conditions for the integrability of sequence diagrams. Instead we refer to the relevant literature.

For instance, in [KCH04] the integration of Message Sequence Charts (MSCs, cf. [ITU96]) which are a kind of predecessor of UML sequence diagrams is discussed. This approach has been adapted to sequence diagrams in [KFJ07] in the context of aspect orientation. The integration of two sequence diagrams is formally described by means of an *amalgamated sum* there, but conforms largely with our approach: the diagrams there defined as point-cut correspond to the base view in our approach, while the base and advice diagrams are related to the views. The resulting diagram is then the counterpart of our integration model.

An integration process similar to ours, but without using a common base defining the synchronisation points is presented in [BB07]. The approach is based on a categorical construction using labelled prime event structures [WN95]: the synchronisation points of two (probably independent) sequence diagrams are calculated as a pull-back, and the integration as push-out.

Finally we want to note that the order of integration does not influence the integrated model. This is due to the fact that the interaction operator par is associative and commutative, which has been proven for example in [MB09] by means of a co-algebraic semantic framework for interactions.

3.5 Application of the Methodology within the Thesis

The subject of this thesis is a generic framework for integrative environmental simulations. For this kind of system, considering the requirements of the enclosing research project, the following fundamental system aspects were identified each of them giving rise to a particular system view in the sense of our development methodology:

- *Data exchange* between simulation models,
- consistent treatment of *simuation space*, and
- *time coordination* for parallel running simulation models with individual time steps.

Each single view extends a *base architecture* which describes a fundamental simulation framework for the parallel execution of a number of (independent) simulation models. The development methodology described in this section is applied to the generic simulation framework as shown in Figure 3.29. In this figure all necessary extension and refinement arrows are depicted. In the remainder of this thesis we describe the single steps from the base architecture (Chapter 4) over the different views (Chapters 5, 6, and 7) to the integration (Chapter 8) which comprises the complete architecture of the framework.

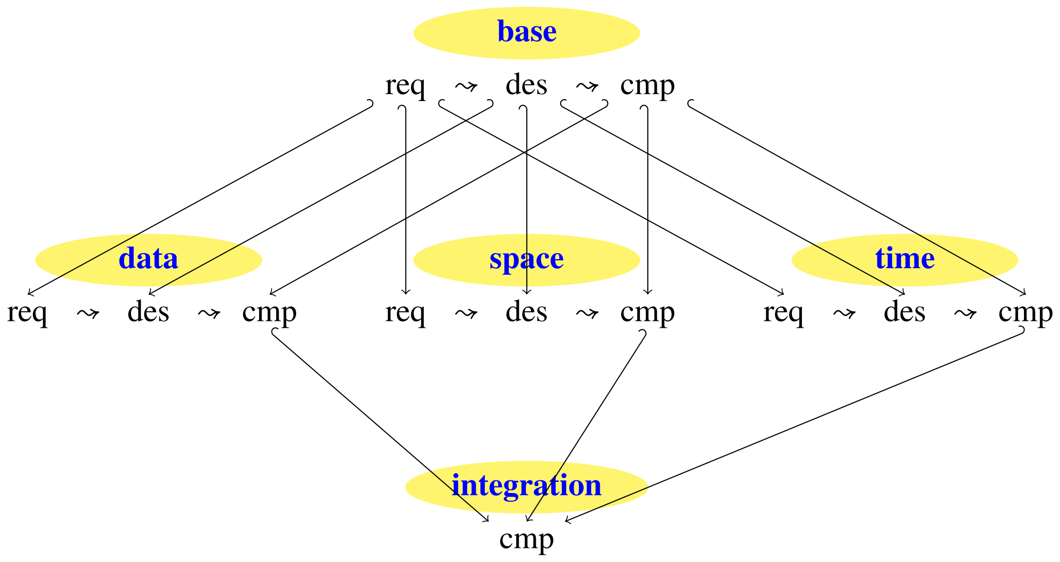
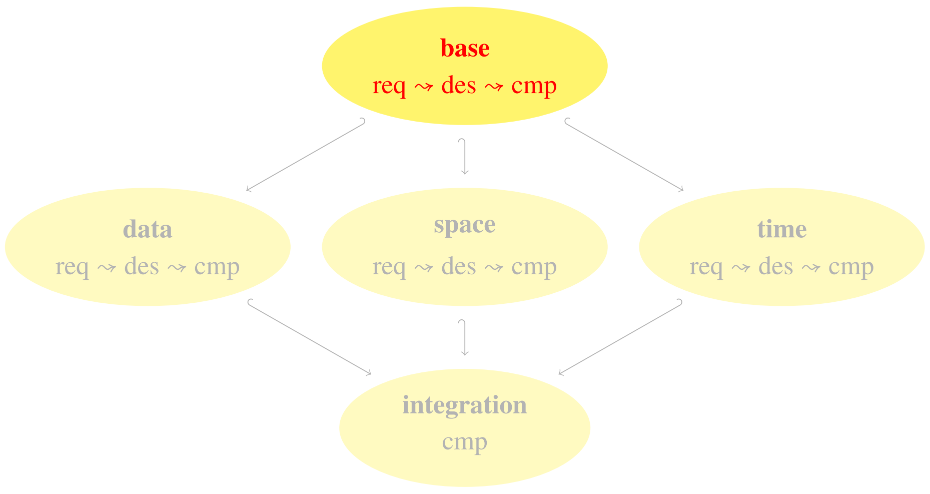


Figure 3.29: Application of the methodology to the generic simulation framework

4

System Use Cases and Base Architecture of the Simulation Framework



With this chapter the course through our development methodology starts with elaborating a base architecture of the framework. In the above figure, the red label of the uppermost ellipse denotes the task under consideration. In the remainder of this chapter we first describe the system use cases (Section 4.1) and then develop a base architecture of the simulation framework. The development of the architecture is structured according to the methodology described in Chapter 3: the requirements are specified in Section 4.2, a design model is presented in Section 4.3, and the component architecture derived from the design model is shown in Section 4.4.

4.1 System Use Cases

The only system use case executeSimulation of the simulation framework is depicted in Figure 4.1. The actors which are connected to this use case are a (graphical) user interface on the one hand, and a number of simulation models on the other hand. The framework, the user interface and the models form the integrative simulation system. In the remainder of this thesis we focus on the simulation framework and show how the other actors can be connected.

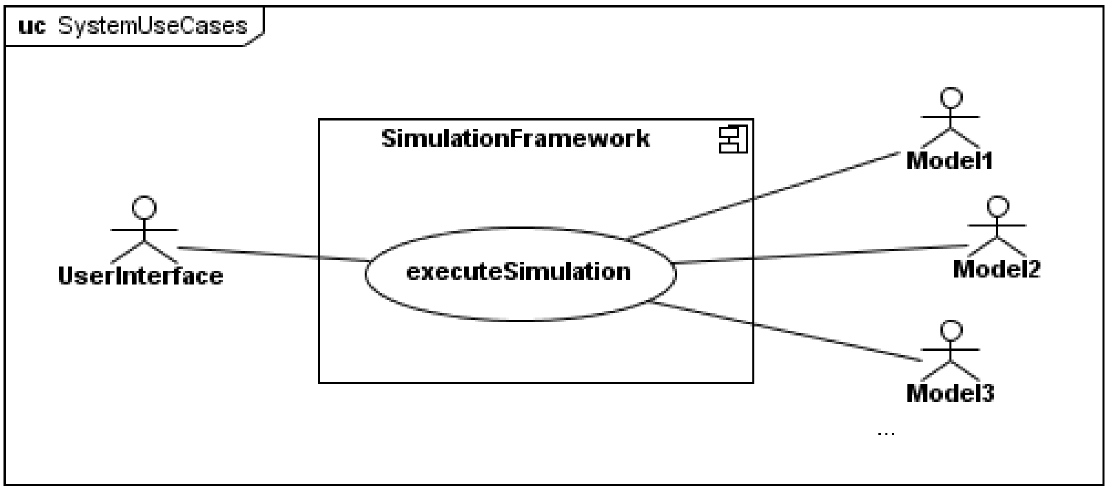


Figure 4.1: System Use Case for the Simulation Framework

4.2 Requirements

We introduce the concept of an *integrative simulation* (in the following often called *simulation* for short) by the class Simulation which is associated to a set of participating simulation models; an arbitrary *simulation model* (in the following often called *model* for short) is represented by the class Model (cf. Figure 4.2). A simulation or a model can be identified by a unique simulationId or a modelId respectively (cf. Section 3.2.2 for the meaning of the property {key}).

The sequence diagram in Figure 4.3 shows the typical course of action when an integrative simulation is started. By means of an appropriate user interface a Simulation is created and started. Then instances for all participating models have to be created and started as well (denoted within the loop fragment). Typically, after a model is started it first has to be initialised (cf. message init) and then performs its computation (compute). Before terminating, some final actions like, e.g., closing of open files or database connections, may be executed (finalize) and the Simulation object is notified that the model has finished its course of action. If this

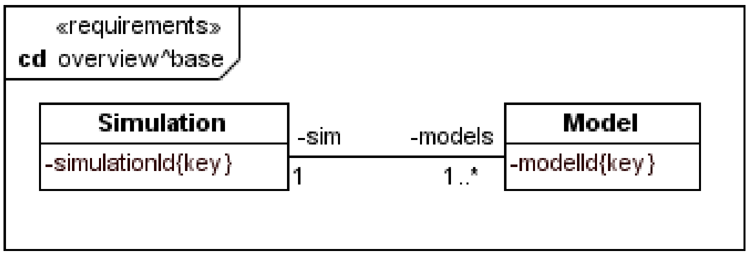


Figure 4.2: Concepts Class Diagram

notification has arrived from all participating models the end of the simulation is signalled to the UserInterface by the message finished as well.

4.3 Design

In this section we develop a structural and a dynamic design model from the requirements model presented in the previous section.

4.3.1 Structural Design Model

An overview of the structural design model is depicted in Figure 4.4. Let us first explain how the classes of the design model emerged from the requirements model and show that the design model is a refinement of the requirements model according to our methodology (cf. Section 3.2). The requirements class Simulation was split into the two classes SimulationAdmin and SimulationConfiguration, i.e. the set of corresponding classes is given by $Cor_{\mathsf{Simulation}} = \{\mathsf{SimulationAdmin}, \mathsf{SimulationConfiguration}\}$. While a SimulationAdmin instance is supposed to act as a management entity for an integrative simulation (and is therefore designed as an active class), the class SimulationConfiguration covers the description of the simulation (like, e.g., the identifier simulationId). This complies with the refinement requirement that there exists a corresponding attribute in one of the corresponding classes.

The concept class Model has been split into the design classes ModelCore, AbstractModel and ModelMetadata, i.e. $Cor_{\mathsf{Model}} = \{\mathsf{ModelCore}, \mathsf{AbstractModel}, \mathsf{ModelMetadata}\}$. While ModelMetadata is a class for storing meta data of a simulation model (like, e.g., the model identifier modelId, the classes ModelCore and AbstractModel represent a simulation model itself. The dispartment into two classes results from the *framework principle* explained in Section 2.2: while ModelCore belongs to the framework core and implements the (general) life cycle of a simulation model within the method start (and therefore is designed as an active class), the class AbstractModel is part of the developer interface of the framework. It constitutes a *base class* (depicted by the corresponding stereotype) for the development of an individual simulation model by (object oriented) inheritance.

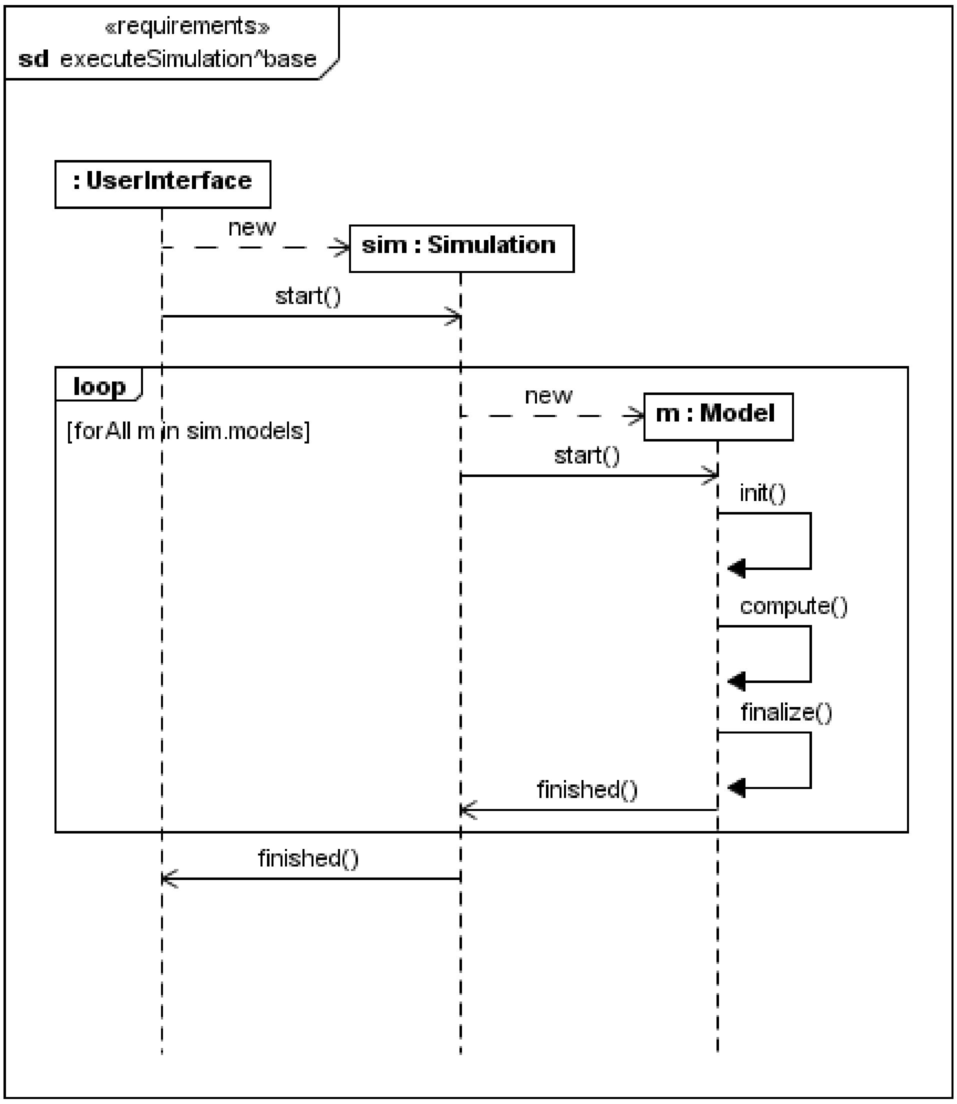


Figure 4.3: Concepts Sequence Diagram

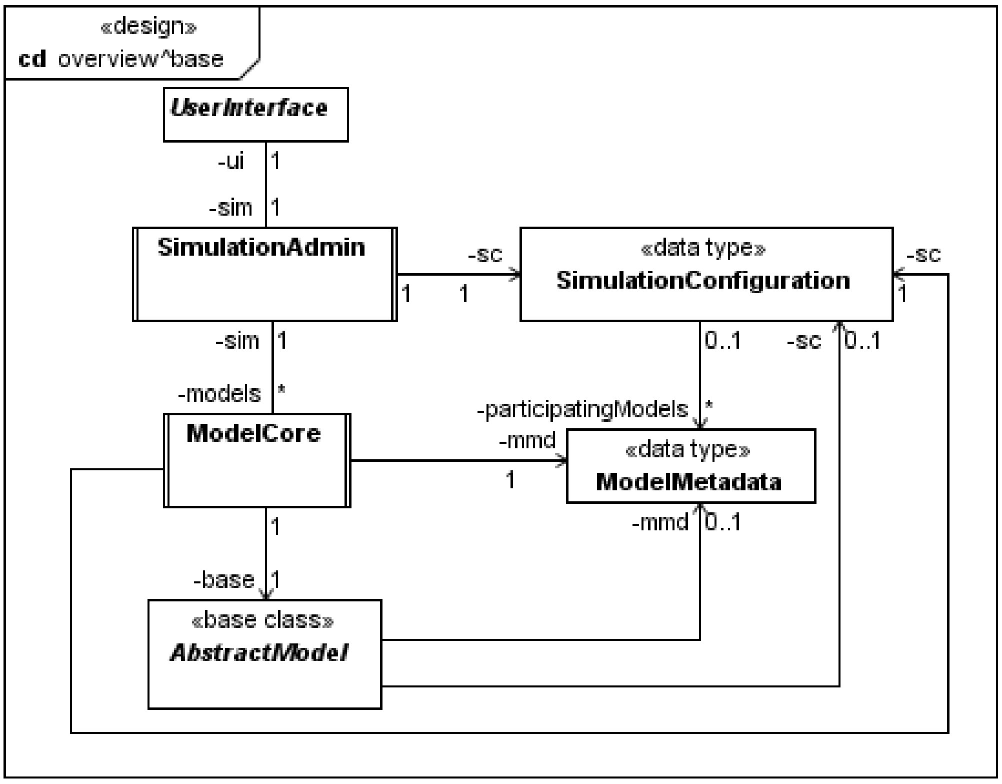


Figure 4.4: Design model of the base view (overview)

The association between Model and Simulation the requirements model has been refined to several associations between corresponding classes. As the association between SimulationAdmin and ModelCore is no longer navigable at the SimulationAdmin end, but the ModelCore (and AbstractModel) instances have to know about the actual SimulationConfiguration, new (directed) associations between these classes are necessary. The directionality of the association between ModelCore and AbstractModel evolves from the framework principle: a framework core class should never be navigable from a base class.

The abstract class UserInterface has been introduced as a new class and represents the user interface for the simulation system which is not part of the framework. It is modelled as an abstract class (instead of an interface) because it does not only receive messages from the SimulationAdmin, but also sends messages to it.

Note that in the design model (as well as in the components model) all classes which belong to the developer interface are marked with an appropriate stereotype (either «base class», «base interface», or «data type»). This is the only possibility to distinguish the developer framework classes from those of the framework core, as the separation of the framework into core and developer interface is only a logical separation which is not reflected by the components later on.

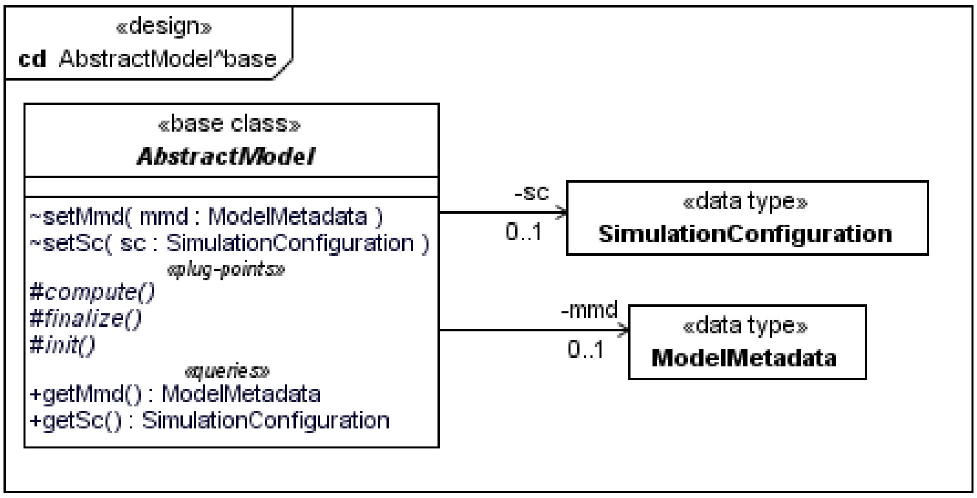


Figure 4.5: Details of the class AbstractModel

The classes depicted in Figure 4.4 are detailed in Appendix A.1.1. In the following we describe only the base class AbstractModel (cf. Figure 4.5) which represents a generic simulation model. As a base class – which is denoted by the corresponding stereotype – it belongs to the developer interface and has to be extended in order to implement a concrete simulation model. The class holds references to the actual SimulationConfiguration and ModelMetadata which can be set by appropriate setters. The setters are package private[1] in order to prevent simulation model developers from changing the actual values by calling these setters within their model implementation – this should only be done by the corresponding ModelCore instance. Access to the SimulationConfiguration and ModelMetadata objects is granted by the respective public queries getSc and getMmd.

Operations of design level classes are mainly derived from the behavioural requirements model or are setters and getters of attributes. In a base class the operations which are dedicated to a simulation model developer – either plug-points or queries – are listed in the operations department headed by the respective stereotype. The class AbstractModel provides three plug points in form of the abstract operations compute, finalize, and init. The plug-points have to be implemented appropriately in a concrete subclass. The visibility of plug-points is protected, because plug-points are designed to be overridden in a subclass. Finally, all queries of AbstractModel are public so that all classes of a model implementation can access the conveyed information.

We have shown that the structural model of the design level (cf. Figure 4.4) is indeed a refinement of the structural model of the requirements level (cf. Figure 4.2).

[1]Although components and packages are not modelled on the design level, we assume that classes representing a concrete simulation model reside in a component or package outside the framework

4.3.2 Behavioural Design Model

An overview of the behavioural design model is depicted in Figure 4.6 in an interaction overview diagram.

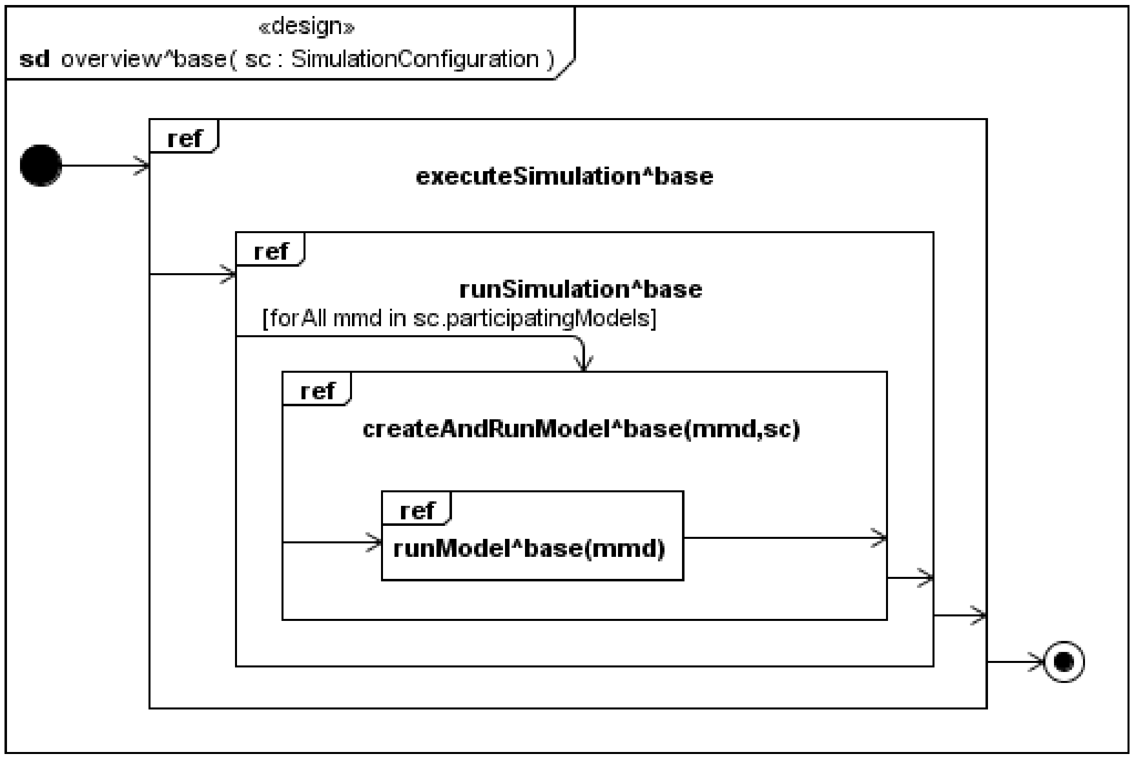

«design»
sd overview^base(sc : SimulationConfiguration)
ref
executeSimulation^base
ref
runSimulation^base
[forAll mmd in sc.participatingModels]
ref
createAndRunModel^base(mmd,sc)
ref
runModel^base(mmd)

Figure 4.6: Interaction overview diagram in the base view

The single diagrams referenced in Figure 4.6 are detailed in Appendix A.1.2. In the following we briefly describe the interactions contained in the single diagrams. The course of interactions starts with the diagram executeSimulation^base in which a SimulationConfiguration object and a SimulationAdmin instance are created. The creation of the SimulationAdmin is guarded by a precondition which says that the SimulationConfiguration object which is passed as parameter to the constructor has to be valid, i.e. its query isValid has to result to true. The result of the query isValid is given by the following postconditions:

```
context SimulationConfiguration::isValid()
 pre:  true
 post: result = self.simulationId <> ""
   and self.participatingModels <> null
   and self.participatingModels->forAll(m | m.isValid())

context ModelMetadata::isValid()
```

```
pre:  true
post: result = self.modelId <> "" and self.modelClass <> ""
```

Informally speaking, a SimulationConfiguration is valid if their attribute simulationId does not equal to the empty string, the property participatingModels is not null and the for each element of the set of participating models the respective ModelMetadata object is valid, too. A ModelMetadata object is again valid, if its attributes modelId and modelClass are not equal to the empty string.

After creating the SimulationAdmin instance and calling the operation start the interactions are specified by the diagram runSimulation^base. This diagram contains a loop in which the diagram createAndRunModel^base is referenced for each ModelMetadata object which is referenced by the current SimulationConfiguration. Note that the loop is denoted in the interaction overview diagram (Figure 4.6) by the guard forAll mmd in sc.participatingModels.

As the diagram createAndRunModel^base is the most interesting diagram of the current view, it is depicted in Figure 4.7). In the following we go into the details of this diagram.

The first event in this diagram is to create a ModelCore instance. As there are several instances of ModelCore in the system, we use the model identifier modelId defined in the ModelMetadata object as selector. Note that the following start message is asynchronous so that the SimulationAdmin gets back control flow after this message and can continue with the creation of the other models of the current simulation configuration (remember that the interaction createAndRunModel is performed within a loop fragment). After the start message has been received by the ModelCore instance it creates an AbstractModel instance. Of course, the abstract class AbstractModel cannot be instantiated, but a concrete subclass that represents a particular simulation model. The type of this class is given by the attribute modelClass of ModelMetadata and the state invariant self.base.getType().getName()=mmd.modelClass on the lifeline of the ModelCore object shall ensure that an object of the correct type has been created.

As it is possible that this condition might not have been fulfilled (e.g. when the required type could not be found), but its validity is crucial for the continuation of the simulation, it is necessary to check the condition within the following break fragment. In the case that the object base is not of the required type, the course of actions continues with the interactions specified in handleException. Note that we deviate from the UML standard in two ways when using a break fragment. Firstly, in our approach a break fragment covers only the lifeline on which the break condition is evaluated instead of all lifelines of the enclosing interaction; secondly, we consider the remainder of the whole interaction after a break fragment as ignored, and not only the remainder of the enclosing interaction. Taking this into account, the system behaves in the case the condition of the break fragment evaluates to true as follows. The ModelCore instance sends the message exception to the actual gate of the interaction use handleException (cf. Figure 4.8). This diagram shows that this message will be received by the SimulationAdmin which in turn sends the message error to the user interface. In this case the simulation terminates with an error. We will reuse the diagram handleException for exception handling later in this thesis.

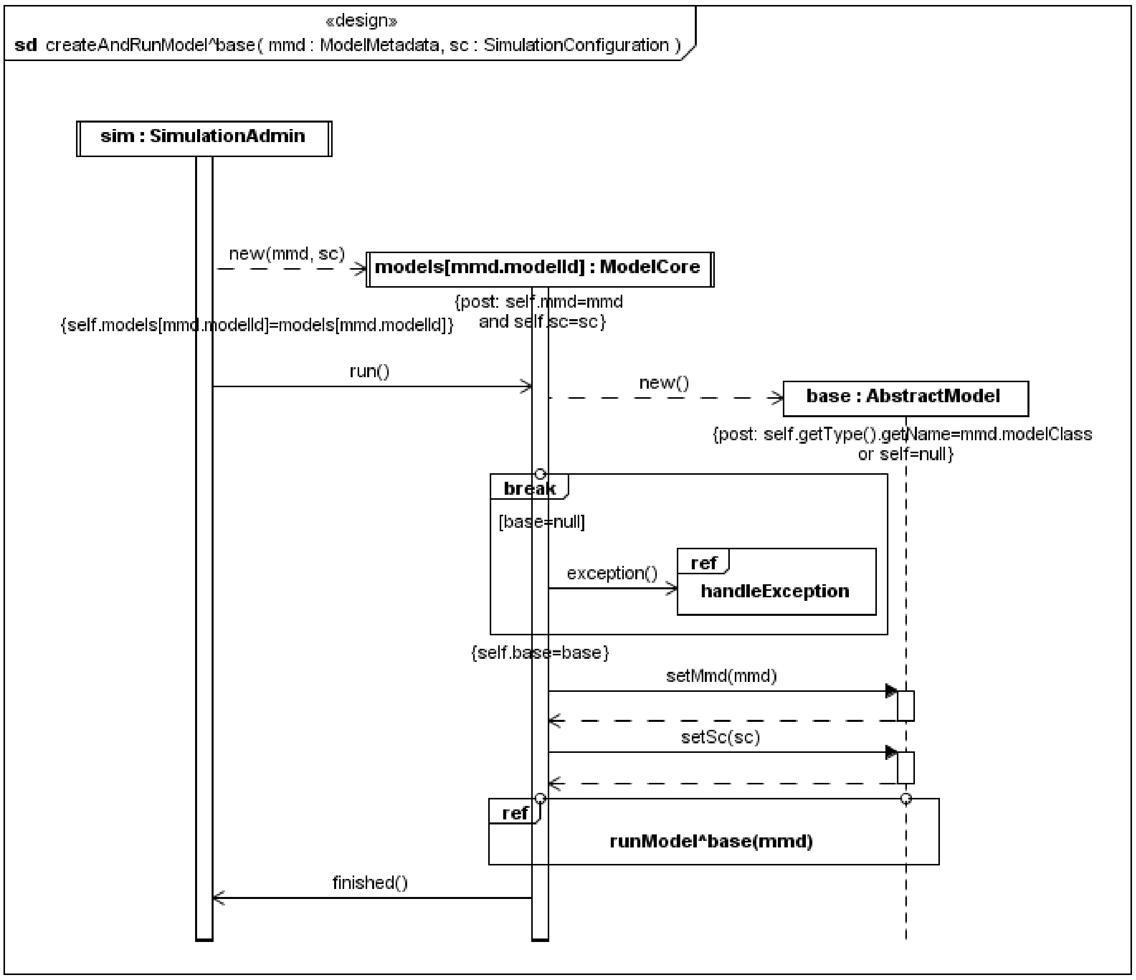

«design»
sd createAndRunModel^base(mmd : ModelMetadata, sc : SimulationConfiguration)
sim : SimulationAdmin
new(mmd, sc)
models[mmd.modelId] : ModelCore
{post: self.mmd=mmd and self.sc=sc}
{self.models[mmd.modelId]=models[mmd.modelId]}
run()
new()
base : AbstractModel
{post: self.getType().getName=mmd.modelClass or self=null}
break
[base=null]
exception()
ref
handleException
{self.base=base}
setMmd(mmd)
setSc(sc)
ref
runModel^base(mmd)
finished()

Figure 4.7: Sequence diagram createAndRunModel in the base view

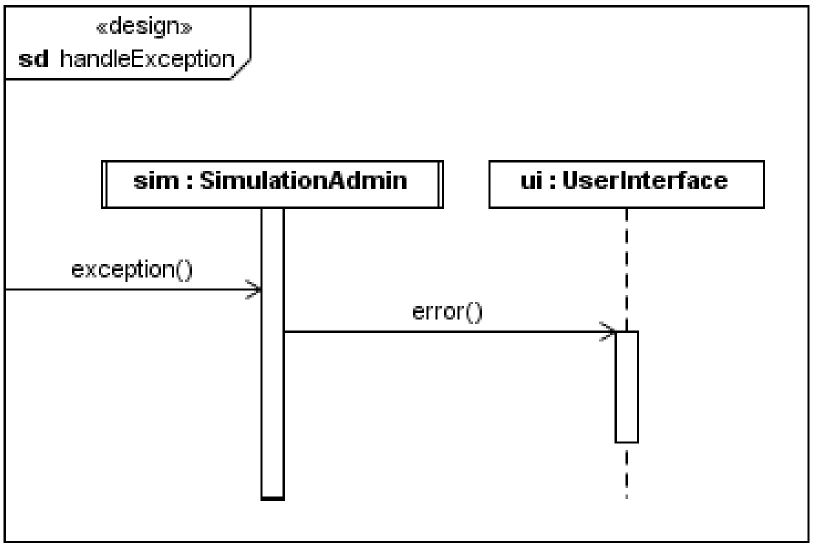


Figure 4.8: Interaction handleException

Finally, in the diagram runModel^base we find the typical actions a model performs within a simulation which have already been stated in the concepts model: first, the simulation model is initialised, then executes its computation and finally, performs a finalisation (e.g. closing open files). In contrast to the concepts model where these actions have been called from the model on itself, on the design level the calls stem from the ModelCore instance and are executed on the AbstractModel instance, i.e. on the concrete simulation model instance at run time.

It is easy to proof, although not evidently, as one has to consider a couple of nested sequence diagrams, that the behavioural design model is a refinement of the behavioural requirements model.

4.4 Components

In this section we provide the component design which provides the basic architecture of the generic integrative simulation framework under consideration. The architecture is given by the component diagram in Figure 4.9. Two components have been identified, the component Simulation on the one hand, and the component Model on the other hand. The former covers the concepts of an integrative simulation, while the latter represents a generic[2] simulation model. Note that the connector ends at the component Model exhibit multiplicity * which indicates that at run time arbitrary many instances of this component may exist.

[2]generic means that at run time a concrete model instance has to be bound to this component

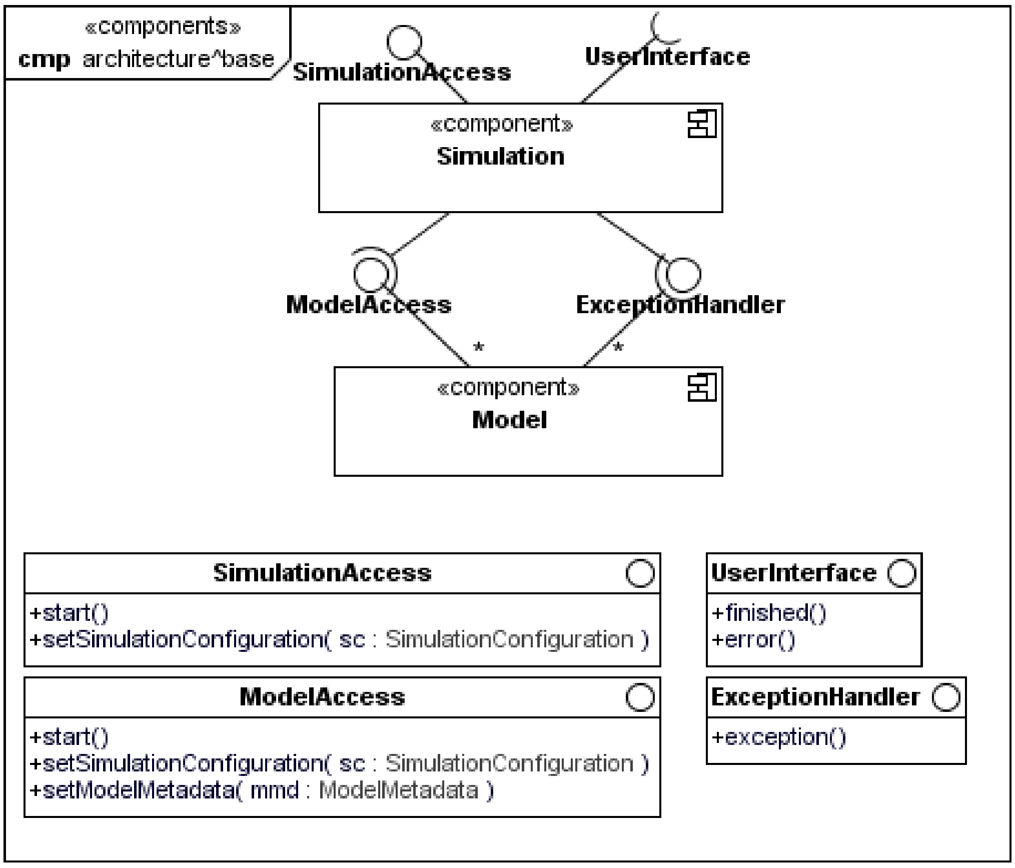


Figure 4.9: Architecture of the base view

The components provide and require several interfaces which are also depicted in Figure 4.9. The interfaces depicted above the component Simulation serve the connection with the user interface; while the provided interface SimulationAccess receives calls from the user interfaces, like e.g. start, the required interface UserInterface notifies the user interface about changes in the system state by calling finished or error. Note that an implementation of a user interface is not part of the framework, hence the interfaces to the user interfaces remain without a counterpart.

The remaining interfaces, ModelAccess and ExceptionHandler, are intended for the communication between the administrative component Simulation on the one hand and the simulation models represented by the component Model on the other hand. For the details of the components Simulation and Model we refer to Appendix A.1.3.

4.5 Conformance with the Development Methodology

In order to show that the architecture developed so far conforms with the development methodology presented in Chapter 3, we have to show that

$$\mathrm{req}^{base} \rightsquigarrow \mathrm{des}^{base} \rightsquigarrow \mathrm{cmp}^{base},$$

where l^v denote the models of abstraction level l under the view v (using abbreviations for level and view names). The refinement step from requirements to design has already been considered in Section 4.3. In the following we concentrate on the refinement from the design to the components level.

According to the rules stated in Section 3.2.3 we have to show that for each class of the design model there exists an equivalent class in one of the components or packages of the component model (except for the abstract class UserInterface which is by definition not part of the framework). The following table summarises, where the equivalent classes can be found in the component model.

The class	has an equivalent class in
SimulationAdmin	component Simulation
AbstractModel	component Model
ModelCore	component Model
SimulationConfiguration	package metadata
ModelMetadata	package metadata

As each class of the design model possess an equivalent class in one of the components or packages of the component model we still have to show that each association of the design model is represented in the component model by either an equivalent association, or by a sequence of connectors between equivalent classes. In order to be able to examine the associations properly we revisit the structural design model given in Figure 4.4. In Figure 4.10 this diagram is shown again, but with numbered associations.

With this numbering in mind, we can state the following.

- The bidirectional association ① is represented in the components model by two sequences of connectors, comprising the interfaces SimulationAccess and UserInterface respectively. In fact, as one of the associated classes does not belong to the framework (by definition), the assembly connectors containing the interfaces SimulationAccess and UserInterface are incomplete in the components model. This situation fulfils the requirements for refinement anyhow.

- The bidirectional association ② is represented in the components model by two sequences of connectors, comprising the interfaces ModelAccess and ExceptionHandler respectively.

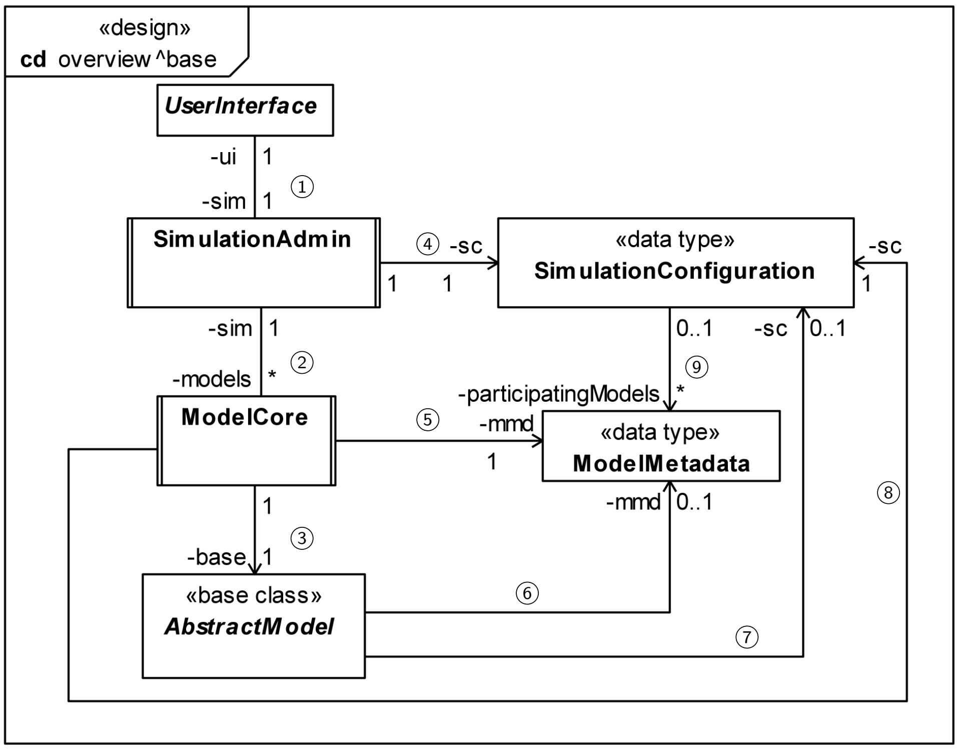


Figure 4.10: The design class model revisited

- The associations ③ to ⑨ possess an equivalent association in the components model, as the associated classes either reside in the same component or package (associations ③ and ⑨), or one of the associated classes belongs to a package and is imported into a component (associations ④ to ⑧).

This completes the proof of the refinement step.

4.6 Discussion

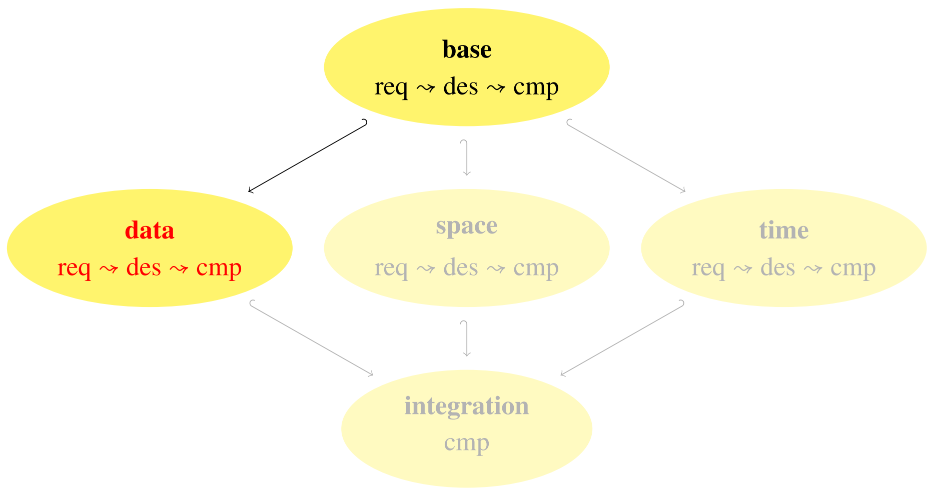
Finally we want to discuss our framework approach in the context of other frameworks in the field of environmental simulation and modelling.

The usage of (abstract) base classes to facilitate the implementation of simulation models is a common approach in environmental modelling frameworks. For instance, this approach is used in OMS (cf. [KKO05]) and TIME (cf. [RSP$^+$03]), while in ModCom (cf. [HBvEL03]) and OpenMI (cf. [GGW07]) just interfaces are defined which must be implemented in order to make an entity compliant with the framework. Moreover, ModCom provides components offering numerical services like integrators which can be uses when a simulation model is given in form of ordinary differential equations.

The particular advantage of our approach is the strict separation of the framework core and the developer interface which alleviates the task of the model developer significantly as he or she is only concerned with the absolutely necessary code (and documentation) for the development purpose.

5

View “Data exchange between Simulation Models”



In this chapter we discuss the view “Data exchange between simulation models” according to the development methodology provided in Chapter 3. Hence, in Section 5.1 we state the concepts and requirements of this view, after that, in Section 5.2 a class design is developed from the requirements model. The component model of the view is provided in Section 5.3. The chapter closes with a brief discussion of related approaches for data exchange between simulation models in Section 5.5.

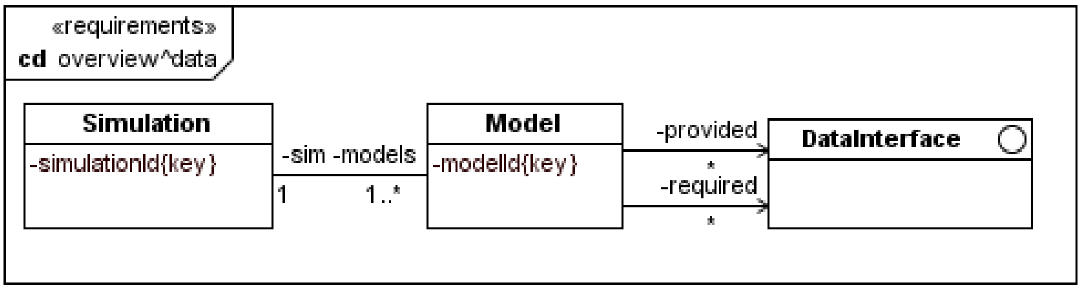


Figure 5.1: Requirements model for data exchange

5.1 Requirements

The coupling of simulation models in our approach is based on *interfaces*. Interfaces for data exchange specify data queries. We distinguish between *provided* interfaces specifying queries for data that is provided by a simulation model, and *required* interfaces specifying queries for data that is needed by a simulation model for its own computation. The general requirements concerning data exchange are modelled in the UML class diagram in Figure 5.1 which is an extension of the base class diagram in Figure 4.2. It says that a simulation may involve arbitrarily many simulation models, which play the role of the models for the simulation, and that a model may have arbitrarily many interfaces, playing the role of provided or required interfaces.

The usage of interfaces has several advantages. First of all, low coupling between the simulation models is achieved, i.e., one model does not need to know about implementation details of the other, but only knows the public interface. This leads directly to the advantage that simulation models can easily be exchanged as long as they provide and require the same interfaces. Furthermore, an interface always specifies a contract. The model which implements an interface is obliged to comply with the contract (i.e. implement all methods of the interface), and the model that uses an interface can rely on the contract (i.e. only use methods that are specified in the interface). Thus, for the model developers and scientific researchers the interface is the point for discussions about the data to be exchanged.

A concrete example of a provided and required interface is given later on when we illustrate the application of the framework in Chapter 9. The following invariant expresses a consistency requirement for data exchange which must be satisfied for any integrative simulation.

Invariant for data exchange

- In an integrative simulation, for each required interface of each participating model there exists exactly one participating model which provides that interface.

This invariant can be formalised in terms of the following OCL-expression:

```
context Simulation inv:
```

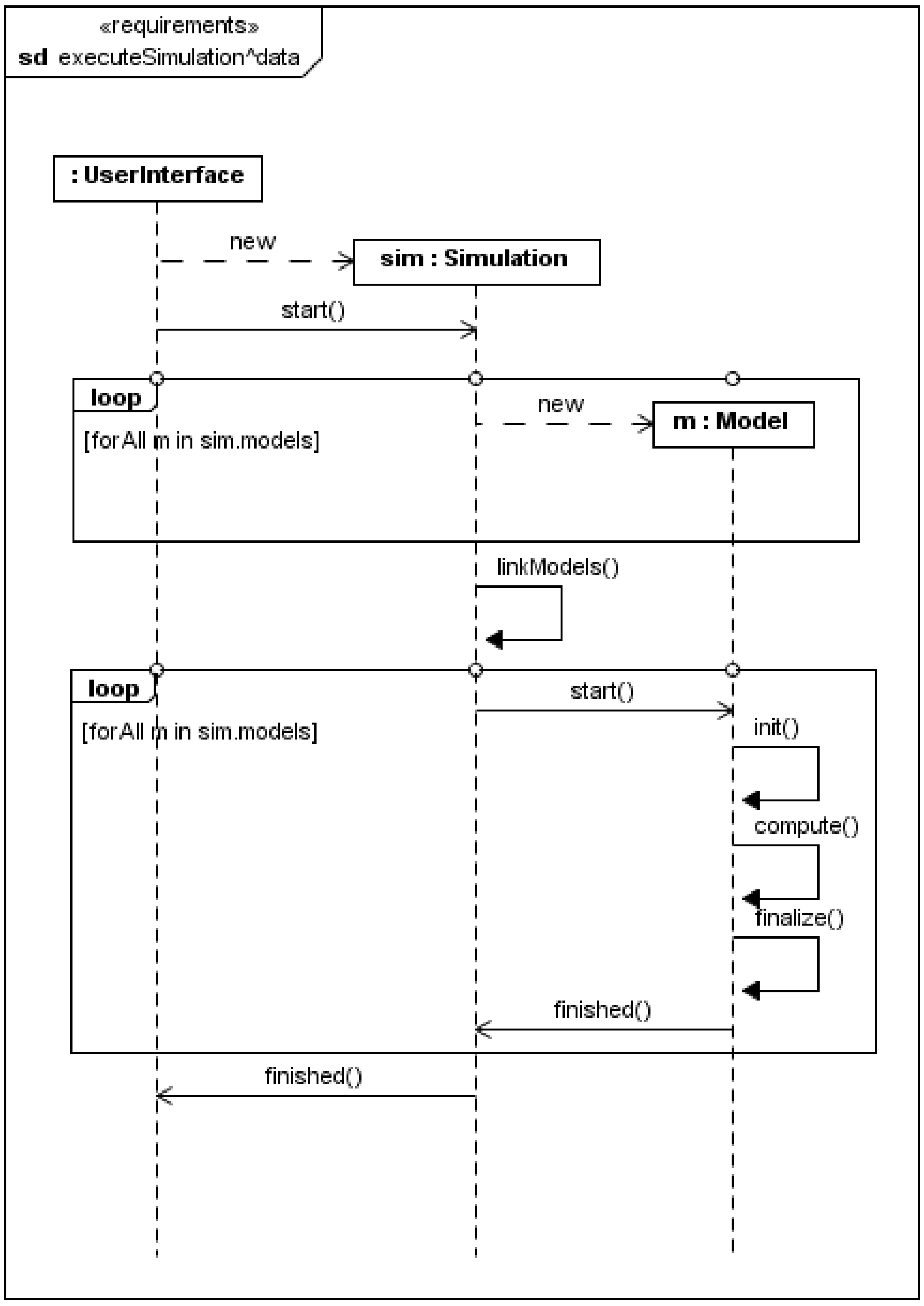


Figure 5.2: Dynamic concepts model for data exchange

```
self.participatingModels.forAll(m |
  m.required->forAll(r |
    self.participatingModels->one(n |
      n.provided->includes(r))))
```

Of course, besides the static concepts model, we have also to extend the dynamic concepts model in order to consider data exchange. As the simulation models are equipped with provided and required interfaces, it is task for the simulation to link these interfaces correctly. For this purpose, the message linkModels has been included into the dynamic model (cf. Figure 5.2) just after creation of all participating models and before the life cycle of the single models is activated by the start message.

5.2 Design

In this section we provide a design model for the view “data exchange” which evolved from the requirements model.

5.2.1 Structural Design

An overview diagram of the static design model is depicted in Figure 5.3. In the diagram the classes which have been added in the view “data exchange” are denoted by the property {new} whereas the classes which have been extended with respect to the base are denoted with the property {ext}. All other classes remained unchanged with respect to the base view.

New elements are the class LinkAdmin and the interface DataInterface which is marked with the stereotype «base interface» to denote that this interface belongs to the developer interface of the framework. All other classes in Figure 5.3 except for UserInterface have been extended in a way.

The interface DataInterface is just a marking interface (i.e. it does not define own operations) to provide a common supertype for all interfaces destined for data exchange. In the following we concentrate on the classes AbstractModel, LinkAdmin and ModelMetadata which are the most interesting ones of the current view. All other classes depicted in Figure 5.3 are detailed in Appendix A.2.1.

AbstractModel

The class AbstractModel (cf. Figure 5.4) is extended by a qualified association to the interface DataInterface. The associated set of objects of type DataInterface is the set of objects which provide import data for the particular simulation model. Each object is qualified by the name of the interface which defines the data to be exchanged. For setting an import object for a particular interface name, the operation setImport is provided. The visibility of this operation is package private because it is designed to be called from framework classes only. In contrast, the corresponding query getImport which expects an interface name as parameter and delivers

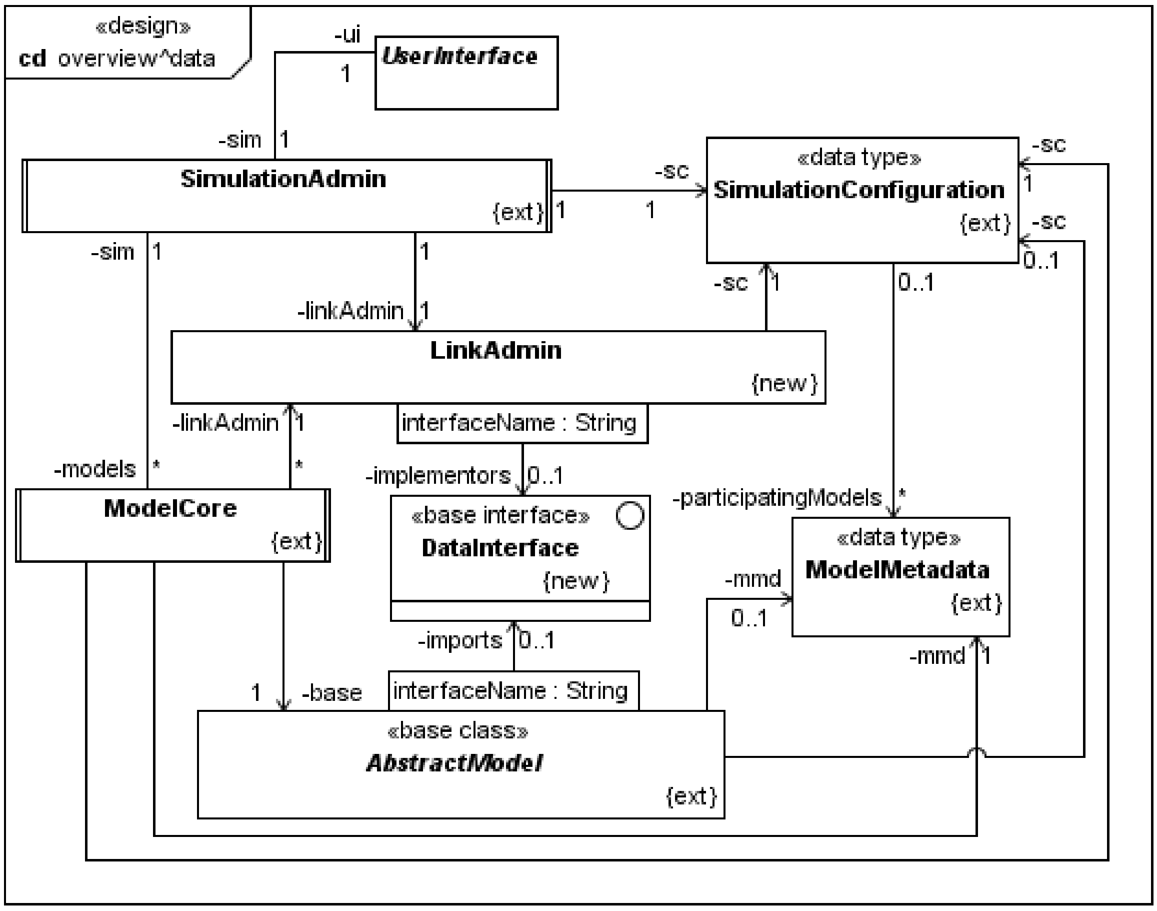


Figure 5.3: Overview of the class design model in the view “Data Exchange”

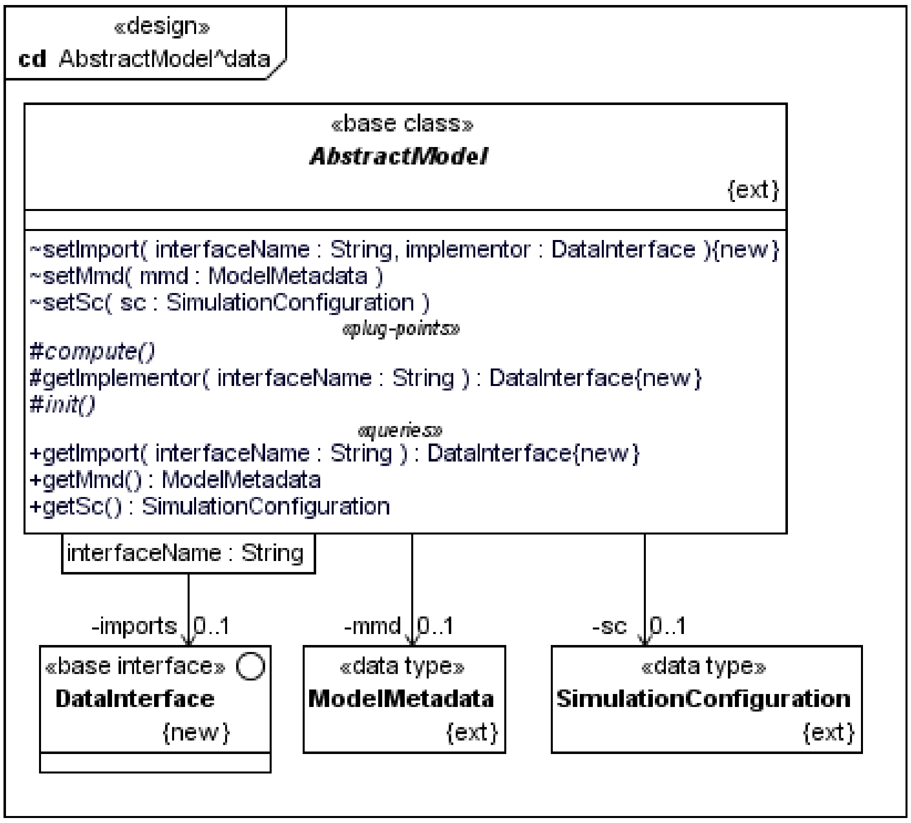

«design»
cd AbstractModel^data
«base class»
AbstractModel
{ext}
~setImport(interfaceName : String, implementor : DataInterface){new}
~setMmd(mmd : ModelMetadata)
~setSc(sc : SimulationConfiguration)
«plug-points»
#compute()
#getImplementor(interfaceName : String) : DataInterface{new}
#init()
«queries»
+getImport(interfaceName : String) : DataInterface{new}
+getMmd() : ModelMetadata
+getSc() : SimulationConfiguration
interfaceName : String
-imports 0..1
-mmd 0..1
-sc 0..1
«base interface»
DataInterface
{new}
«data type»
ModelMetadata
{ext}
«data type»
SimulationConfiguration
{ext}

Figure 5.4: Details of the class AbstractModel

the respective import object has public visibility. In order to obtain a valuable result from this query, the following precondition must hold:

```
context AbstractModel::getImport(interfaceName:String)
pre:  self.mmd.importInterfaces->exists(s | s = interfaceName)
post:  result <> null
       and result.getType().getName() = interfaceName
```

In other words, the interface name for which an implementing object is requested must be contained in the list of import interfaces of the respective ModelMetadata object.

In the view “data exchange” the optional[1] plug point getImplementor is introduced. The plug point is called from the framework core to obtain an implementing object for an export interface of the particular model.

LinkAdmin

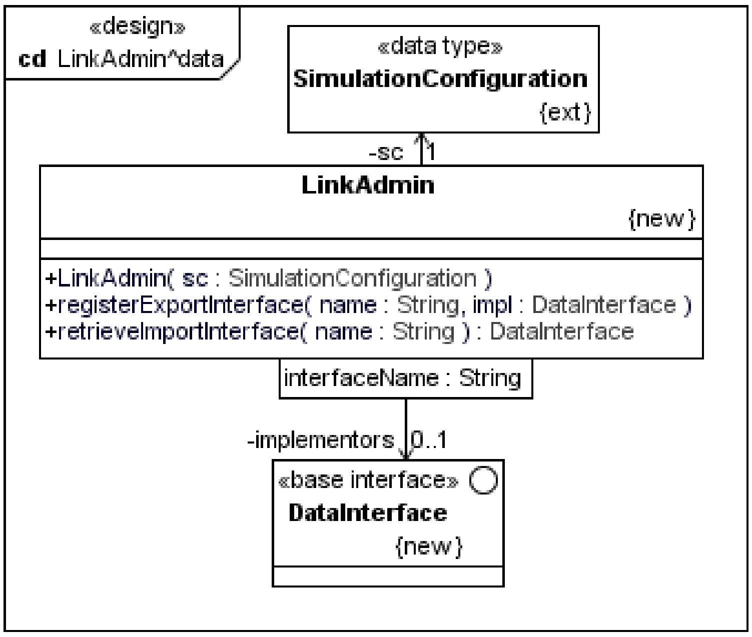

«design»
cd LinkAdmin^data
«data type»
SimulationConfiguration
{ext}
-sc
1
LinkAdmin
{new}
+LinkAdmin(sc : SimulationConfiguration)
+registerExportInterface(name : String, impl : DataInterface)
+retrieveImportInterface(name : String) : DataInterface
interfaceName : String
-implementors
0..1
«base interface»
DataInterface
{new}

Figure 5.5: Details of the class LinkAdmin

The class LinkAdmin is designed for managing the correct linking between simulation models for data exchange. For this purpose it provides the operations registerExportInterface and retrieveImportInterface. The procedure of linking data interfaces is described in detail in Section 5.2.2 where the behaviour of the classes is considered.

[1] A plug point is *optional* if a default implementation is provided by the framework.

ModelMetadata

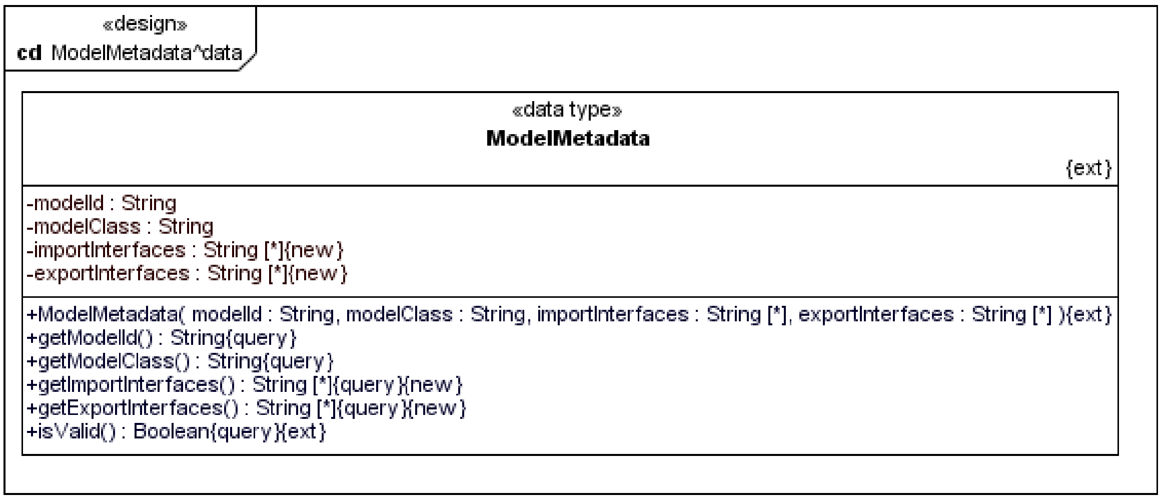

«design»
cd ModelMetadata^data
«data type»
ModelMetadata
{ext}
-modelId : String
-modelClass : String
-importInterfaces : String [*]{new}
-exportInterfaces : String [*]{new}
+ModelMetadata(modelId : String, modelClass : String, importInterfaces : String [*], exportInterfaces : String [*]){ext}
+getModelId() : String{query}
+getModelClass() : String{query}
+getImportInterfaces() : String [*]{query}{new}
+getExportInterfaces() : String [*]{query}{new}
+isValid() : Boolean{query}{ext}

Figure 5.6: Details of the class ModelMetadata

The class ModelMetadata has been extended by the attributes exportInterfaces and importInterfaces, both of type String[*], and their respective getters. Of course, the parameter list of the constructor has been also extended to carry values for these attributes. The values of the attributes provide the names of the export and import interfaces of the simulation model respectively. The attributes play a role for the validity of the SimulationConfiguration which is stated by the following postcondition of the query isValid:

```
context SimulationConfiguration::isValid()
 pre:  true
 post: result = simulationId <> ""
   and self.participatingModels <> null
   and self.participating.Models->forAll(m | m.isValid())
   and self.participatingModels.forAll(m |
       m.importInterfaces->forAll(i |
          self.participatingModels->one(n |
            n.exportInterfaces->includes(i))))

context ModelMetadata::isValid()
 pre: true
 post: result = self.modelId <> ""
        and self.modelClass <> ""
        and self.exportInterfaces <> null
        and self.importInterfaces <> null
```

Note that if the query isValid results to true then obviously the invariant for data exchange stated in Section 5.1 is fulfilled.

5.2.2 Dynamic Design Model

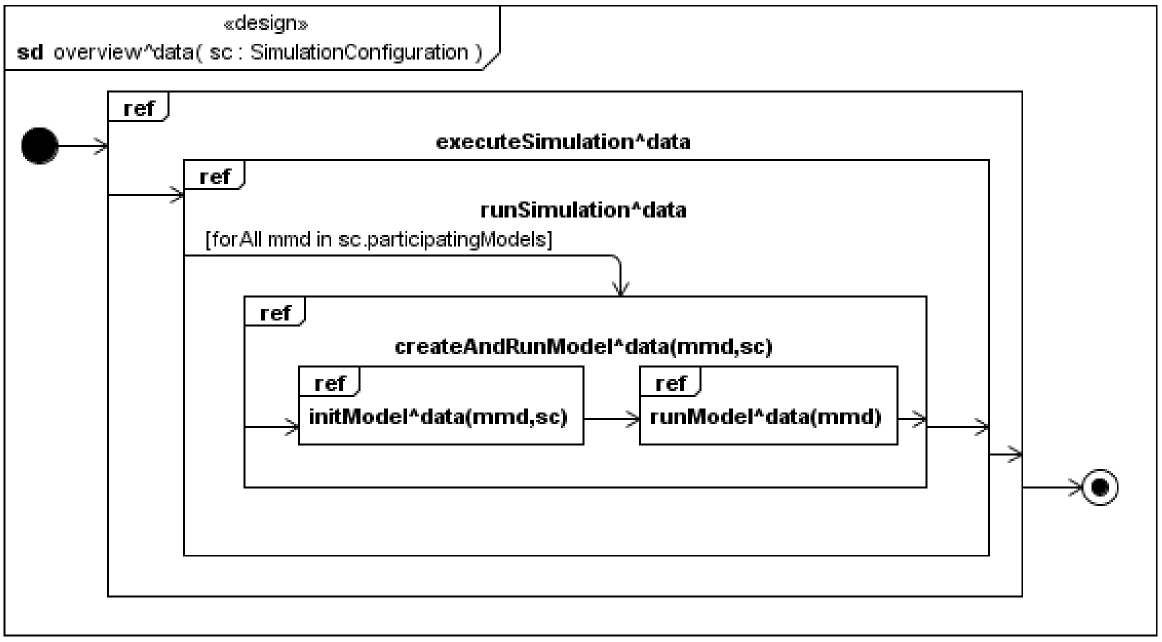

«design»
sd overview^data(sc : SimulationConfiguration)
ref
executeSimulation^data
ref
runSimulation^data
[forAll mmd in sc.participatingModels]
ref
createAndRunModel^data(mmd,sc)
ref
initModel^data(mmd,sc)
ref
runModel^data(mmd)

Figure 5.7: Interaction overview diagram of the view “Data Exchange”

The dynamic design model for the view “data exchange” is given by the interaction overview diagram in Figure 5.7. The most important extension with respect to the base view is the additional sequence diagram initModel^data referenced within the diagram createAndRunModel^data. In the following we will detail about this diagram, whereas the details of the remaining diagrams can be found in Appendix A.2.2.

So let us consider the interactions in the diagram initModel^data (cf. Figure 5.8). The task described by the interactions contained in this diagram is to register an implementing object for each export interface of the respective simulation model to the link administration and then to request an implementing object for each import interface from the link administration. Within the referenced diagrams retrieveExportInterfaces and retrieveImportInterfaces the set of names of export interfaces and import interfaces is requested from the ModelMetadata object mmd respectively. The diagram is quite self-explanatory, but we want to point out the following details.

Firstly consider the break fragment on the lifeline of ModelCore. This fragment is necessary as it might be possible that an interface type cannot be resolved (e.g., when it is misspelled

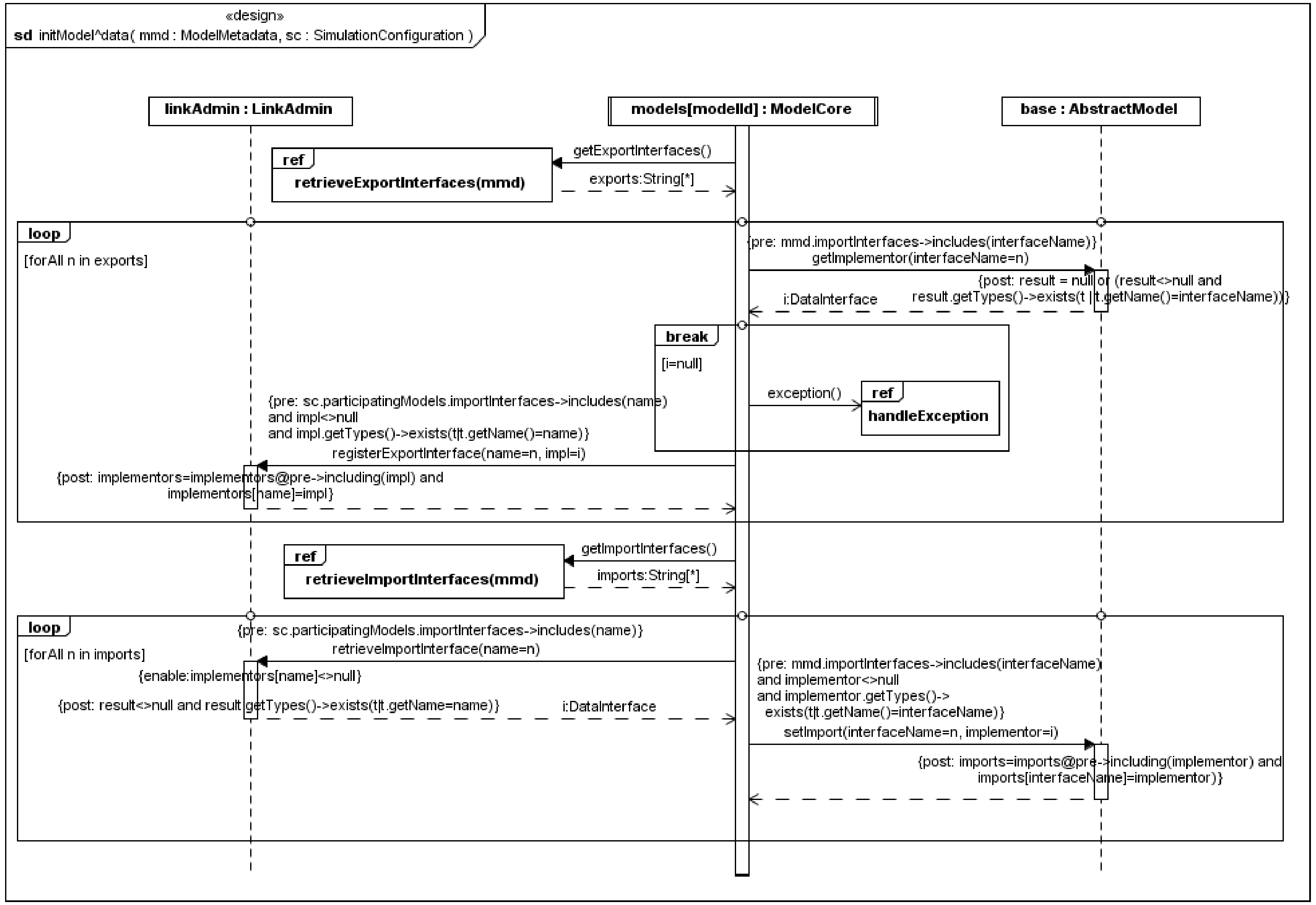

«design»
sd initModel^data(mmd : ModelMetadata, sc : SimulationConfiguration)
linkAdmin : LinkAdmin
models[modelId] : ModelCore
base : AbstractModel
ref
retrieveExportInterfaces(mmd)
getExportInterfaces()
exports:String[*]
loop
[forAll n in exports]
{pre: mmd.importInterfaces->includes(interfaceName)}
getImplementor(interfaceName=n)
{post: result = null or (result<>null and
result.getTypes()->exists(t |t.getName()=interfaceName))}
i:DataInterface
break
[i=null]
exception()
ref
handleException
{pre: sc.participatingModels.importInterfaces->includes(name)
and impl<>null
and impl.getTypes()->exists(t|t.getName()=name)}
registerExportInterface(name=n, impl=i)
{post: implementors=implementors@pre->including(impl) and
implementors[name]=impl}
ref
retrieveImportInterfaces(mmd)
getImportInterfaces()
imports:String[*]
loop
[forAll n in imports]
{pre: sc.participatingModels.importInterfaces->includes(name)}
retrieveImportInterface(name=n)
{enable:implementors[name]<>null}
{post: result<>null and result.getTypes()->exists(t|t.getName=name)}
i:DataInterface
{pre: mmd.importInterfaces->includes(interfaceName)
and implementor<>null
and implementor.getTypes()->
exists(t|t.getName()=interfaceName)}
setImport(interfaceName=n, implementor=i)
{post: imports=imports@pre->including(implementor) and
imports[interfaceName]=implementor)}

Figure 5.8: Sequence diagram initModel^data in the view “Data Exchange”

in the model meta data). In this case the result of the plug-point call getImplementor is null and the condition of the break fragment holds. The course of action continues with calling the formal gate exception on the interaction use handleException which has been introduced in Section 4.3 (cf. Figure 4.8). Note that according to our semantics of a break fragment the system terminates with the execution of the interaction use handleExecption.

Secondly note that the lowermost activation on the lifeline of LinkAdmin is guarded by an enable condition, i.e. the caller of the operation retrieveImportInterface is blocked until the condition implementors[name]<>null results to true. Remember that the interactions of this diagram are performed for each participating simulation model, but the LinkAdmin object is always the same in each diagram. Hence it is guaranteed by the isValid property of SimulationConfiguration which has been checked before creating the SimulationAdmin that the enable condition once results to true.

5.3 Components

The component architecture of the view "data exchange" is depicted in Figure 5.9. With respect to the base view the architecture newly comprises the component ModelLinking which is connected to the component Simulation by the interface ModelLinkingAccess and to the component Model by the interface LinkHandler.[2] The latter interface provides the operations registerExportInterface and retrieveImportInterface by which the data links between simulation models are established as described in Section 5.2.2. The details of the components Simulation, ModelLinking and Model can be found in Appendix A.2.3.

5.4 Conformance with the Development Methodology

After elaborating the architecture of the view "Data Exchange between Simulation Models" (or data view for short), we consider again our development methodology. In the current chapter the following extension and refinement steps have been performed.

$$\begin{tikzcd}
\mathrm{req}^{base} \arrow[r, squiggly] \arrow[d] & \mathrm{des}^{base} \arrow[r, squiggly] \arrow[d] & \mathrm{cmp}^{base} \arrow[d] \\
\mathrm{req}^{data} \arrow[r, squiggly] & \mathrm{des}^{data} \arrow[r, squiggly] & \mathrm{cmp}^{data}
\end{tikzcd}$$

In Section 4.5 we have already considered the refinement steps of the base view (the uppermost line in the above diagram). Hence it remains to prove that,

1. the abstraction levels of the data view are in refinement relation, and

[2]Correctly speaking, the components are connected by assembly connectors which refer to the respective interface.

2. each abstraction level of the data view is an extension of the respective level of the base view,

in order that the above diagram commutes.

We leave the proof of (1) to the reader, as it is similar to the one in Section 4.5, and concentrate on (2) in the following. Of course we could use the mathematical notation introduced in Chapter 3 for the proof, but this would soon lead to an overwhelming effort. So we restrict ourselves to an informal argumentation.

- $\mathrm{req}^{base} \hookrightarrow \mathrm{req}^{data}$.

  For the structural models this is obvious, as omitting the interface DataInterface and its associations from the data view requirements model results in the requirements model of the base view.

  Considering the behavioural models, one can find that the model of the data view (given by the sequence diagram executeSimulation^data in Figure 5.2) covers the same lifelines as the corresponding model of the base view (Figure 4.3) and is extended by exactly one interaction fragment, namely the recursive message call linkModels on the lifeline of type Simulation. Note that, although the loop fragment is split into two parts, the partial order of the interaction fragments of the base view diagram is preserved in the data view diagram, so that we have in fact an extension here.

- $\mathrm{des}^{base} \hookrightarrow \mathrm{des}^{data}$.

  Let us first consider the structural model. Looking at the respective overview diagrams, Figure 4.4 for the base view and Figure 5.3 for the data view, one can see, that the former is extended by the class LinkAdmin and the interface DataInterface and their related associations. We still have to show that the remaining classes in the data view model are extensions of the respective classes of the base view model. The class AbstractModel is extended by the operation setImport, the plug-point getImplementor and the query getImport. In the class ModelCore the parameter list of the constructor has been extended by an entry linkAdmin:LinkAdmin. Finally, the class ModelMetadata is extended by the attributes importInterfaces and exportInterfaces and their respective queries. Likewise the parameter list of the constructor of this class is extended to incorporate values for the new attributes.

  Concerning the behavioural model, we have to look at the nested sequence diagrams executeSimulation in either view. The respective interaction overview diagrams, Figure 4.6 for the base view and Figure 5.7 for the data view, differ only in the diagram initModel^data which is additional to the data view. Hence, in order to verify the extension properties for the whole diagram, it is sufficient to show them for the single corresponding interaction uses in the nested sequence diagram. We do this from inside to outside.

  The innermost diagram is named runModel, and these diagrams coincide in each view. The next diagram in the hierarchy to consider is createAndRunModel. Here the creation of the ModelCore instance is equipped with an additional parameter in the data view. Moreover, the interaction use initModel^data is additional to the data view diagram (as already mentioned before). In the diagram runSimulation one can find the creation of a LinkAdmin instance in the data view as an extension of the base view. The outermost diagram, executeSimulation is again equivalent in both views.

In summary we have seen, that the design model of the data view is indeed an extension of the design model of the base view.

- $\mathrm{cmp}^{base} \hookrightarrow \mathrm{cmp}^{data}$.

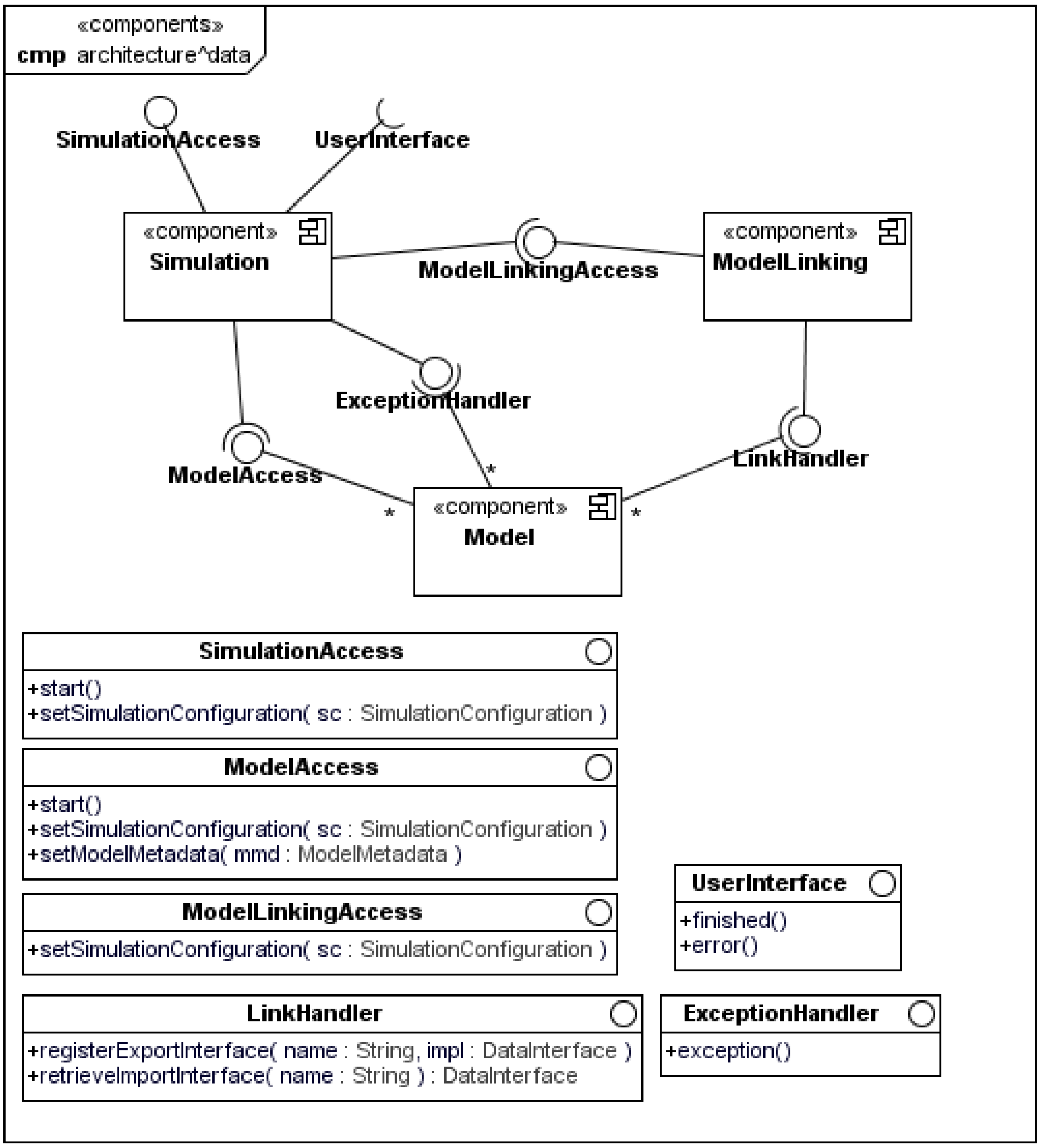
  We consider the architecture diagrams of the components level of each view. Obviously, by omitting the component ModelLinking and its attached assembly connectors from the data view model (cf. Figure 5.9), we obtain the corresponding model of the base view (Figure 4.9). It remains to prove that the components Simulation and Model of the data view are extensions of the corresponding components of the base view, but this can be reduced to the proof on the design level, as the transition from the design to the components level is only a structural refinement step.

This completes the proof.

5.5 Discussion

In principle there exist several approaches to define data exchange between simulation models. Our approach makes use of dedicated interfaces to specify data exchange. Each interface plays two roles; it acts as export interface for the one, and as import interface for the other model. Hence it is easy to check whether a simulation configuration is complete, i.e. for each import interface exists a simulation model offering the same interface as export interface. It is also possible to group coherent exchange parameters together in one interface in order to augment clarity.

Another approach of connecting simulation models is the usage of generic interfaces, like, e.g. within OpenMI (cf. [GGW07]) and ModCom (cf. [HBvEL03]). For instance, within OpenMI an interface IQuantity requires that an exchange item defines properties like ID, description, value type, and dimension. The linking of corresponding output and input exchange items is done – which is also the case within ModCom, OMS (cf. [KKO05]) and TIME (cf. [RSP+03]) – by means of a graphical user interface at run time. For each exchange item a separate link has to be established.



+start()
+setSimulationConfiguration(sc : SimulationConfiguration)

+start()
+setSimulationConfiguration(sc : SimulationConfiguration)
+setModelMetadata(mmd : ModelMetadata)

+setSimulationConfiguration(sc : SimulationConfiguration)

+registerExportInterface(name : String, impl : DataInterface)
+retrieveImportInterface(name : String) : DataInterface

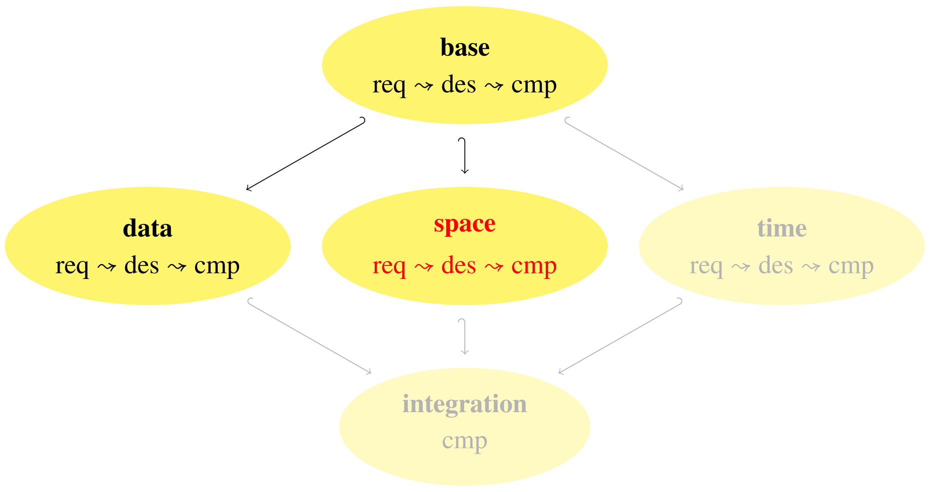
+finished()
+error()

+exception()

Figure 5.9: Architecture in the view “Data Exchange”

6

View “Modelling of the Simulation Space”



In an integrative environmental simulation the consistent treatment of the underlying simulation space is crucial. Hence in this chapter we discuss the view “Modelling of the Simulation Space” again according to our development methodology. We begin with stating the requirements and concepts in Section 6.1, and then move on to develop a class design model in Section 6.2. The component model of this view is presented in Section 6.3, before the chapter is concluded with a brief discussion about related approaches for spatial modelling.

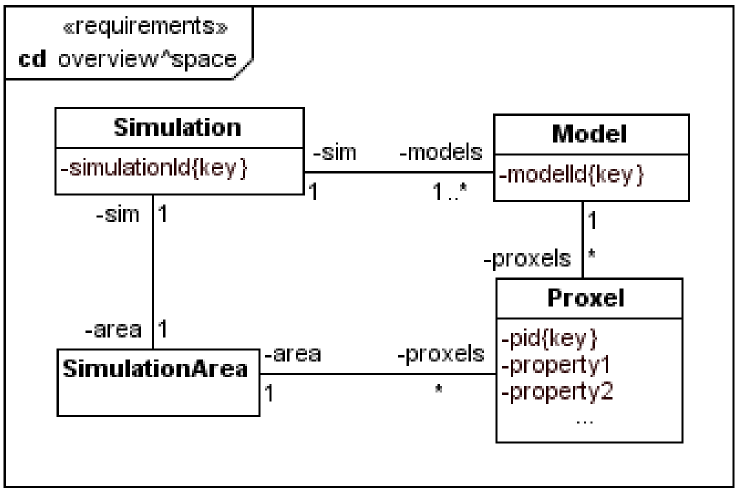


Figure 6.1: Requirements model for the view “simulation space”

6.1 Requirements

It is obvious that in spatially distributed simulations one needs geographical units, which in the following will be called *proxels*. The term proxel (cf. [TK99]) stems from *process pixel* and suggests that a proxel does not only model a structural element of the simulation space, but it shows also dynamic behaviour by simulating the environmental processes on this particular geographical unit. The entire simulation area is then modelled by a set of (non-overlapping) proxels.

The spatial requirements of an integrative simulation are described by the UML class diagram in Figure 6.1. It says that a simulation concerns always exactly one simulation area which, in turn, consists of a set of proxels. The class Proxel requires that each proxel has a unique identifier pid and an operation computeProxel() to compute the next state of a proxel in each time step. Moreover, each proxel can have a number of properties which must be common to all simulation models, like, e.g., geographical coordinates, elevation, land use (cf. [Bra06]), etc.

On the other hand, each simulation model has a set of proxels, on which it operates. The following invariant requires that the models participating in an integrative simulation agree on the set of proxels determined by the area of the simulation.

Invariant for the simulation space

- In an integrative simulation, all participating models operate (only) on proxels which belong to the simulation area of the simulation.

This requirement is expressed in terms of an OCL constraint as follows.

```
context Simulation
  inv area:
    models->forAll(m |
      m.proxels.pid->asSet() = self.area.proxels.pid->asSet()
      and m.proxels->forAll(p1, p2 |
          p1.pid = p2.pid implies p1 = p2))
```

The basic structural properties of the denoted location in the simulation area (like e.g. geographical coordinates, elevation, area) are represented by the attributes property1, property2, etc. The value of each property can be a scalar, a vector or an instance of a complex data structure. To ensure consistency we require that for all participating models, the proxels with the same ID have the same value for each property defined for the respective simulation area. We specify this requirement by an OCL expression too, assuming an appropriate equals relation on each property type which is for simplicity expressed by = in the following OCL invariant.

```
context Simulation
  inv consistency:
    models.forAll(m1, m2 |
      m1.proxels->forAll(p1 |
        m2.proxels->forAll(p2 |
          p1.pid = p2.pid implies
                (p1.property1 = p2.property1)
             and p1.property2 = p2.property2)
             and ...))))
```

let us now consider how the proxel concept is taken into account in the dynamic model. We observe in Figure 6.2 that the sequence diagram executeSimulation^space contains three loop fragments concerning the lifeline of the type Proxel within the model execution loop fragment. Each of the new loop fragments ranges over the set of proxels associated to the simulation model. In the first loop the single proxel objects are created, in the second loop they are initialised and in the third, the computation of the model is delegated to the single proxels.

6.2 Design

In this section we present the class design model of the view “simulation space” which was developed from the concepts model.

6.2.1 Structural Design

Let us first take a look at the structural design model an overview of which is given by the class diagram in Figure 6.3. Several classes have been added in this view which we will describe briefly in the following. The details of these classes – in the case they are not displayed below – can be found in Appendix A.3.1.

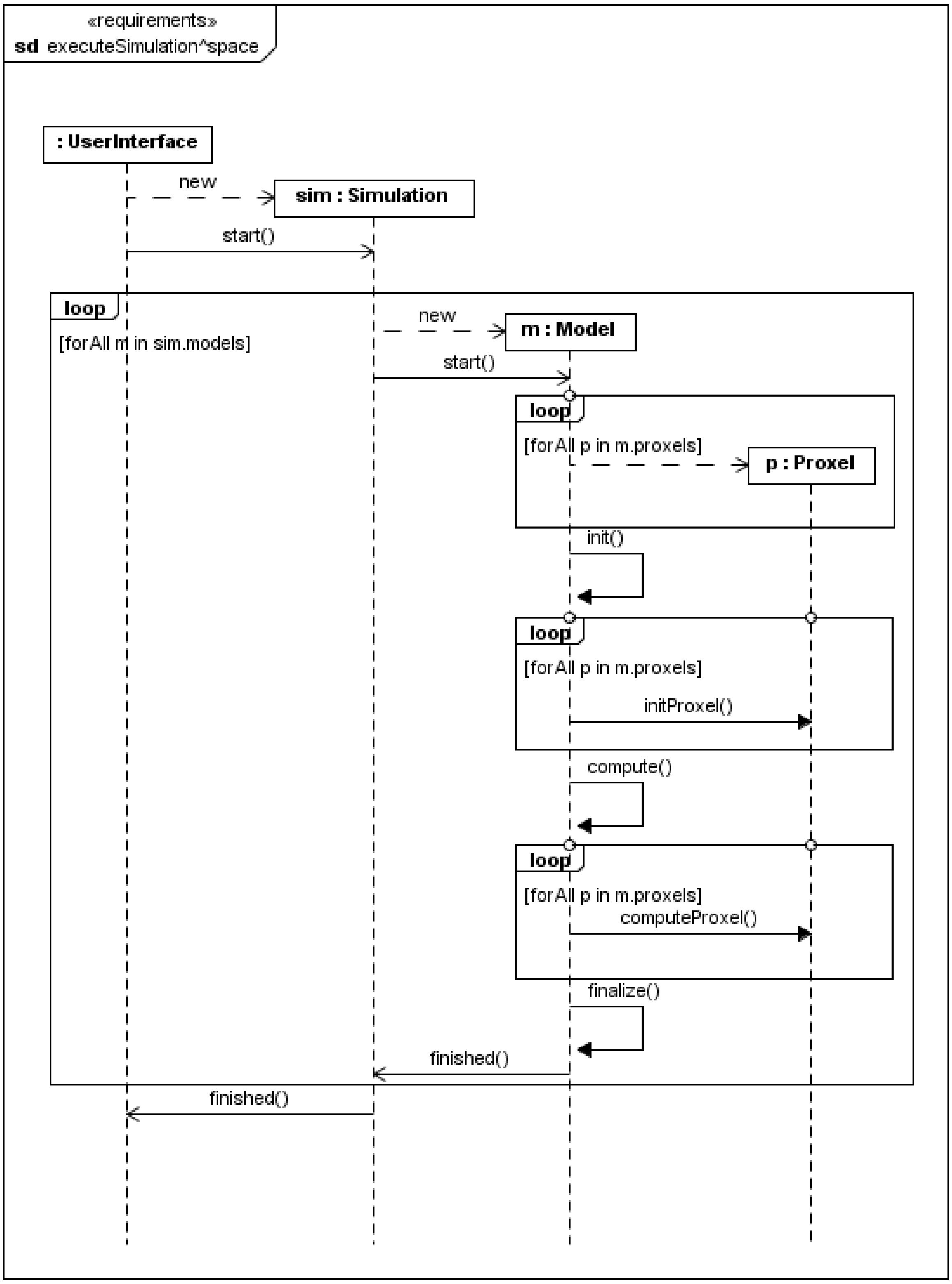


Figure 6.2: Conceptual sequence diagram for the view “simulation space”

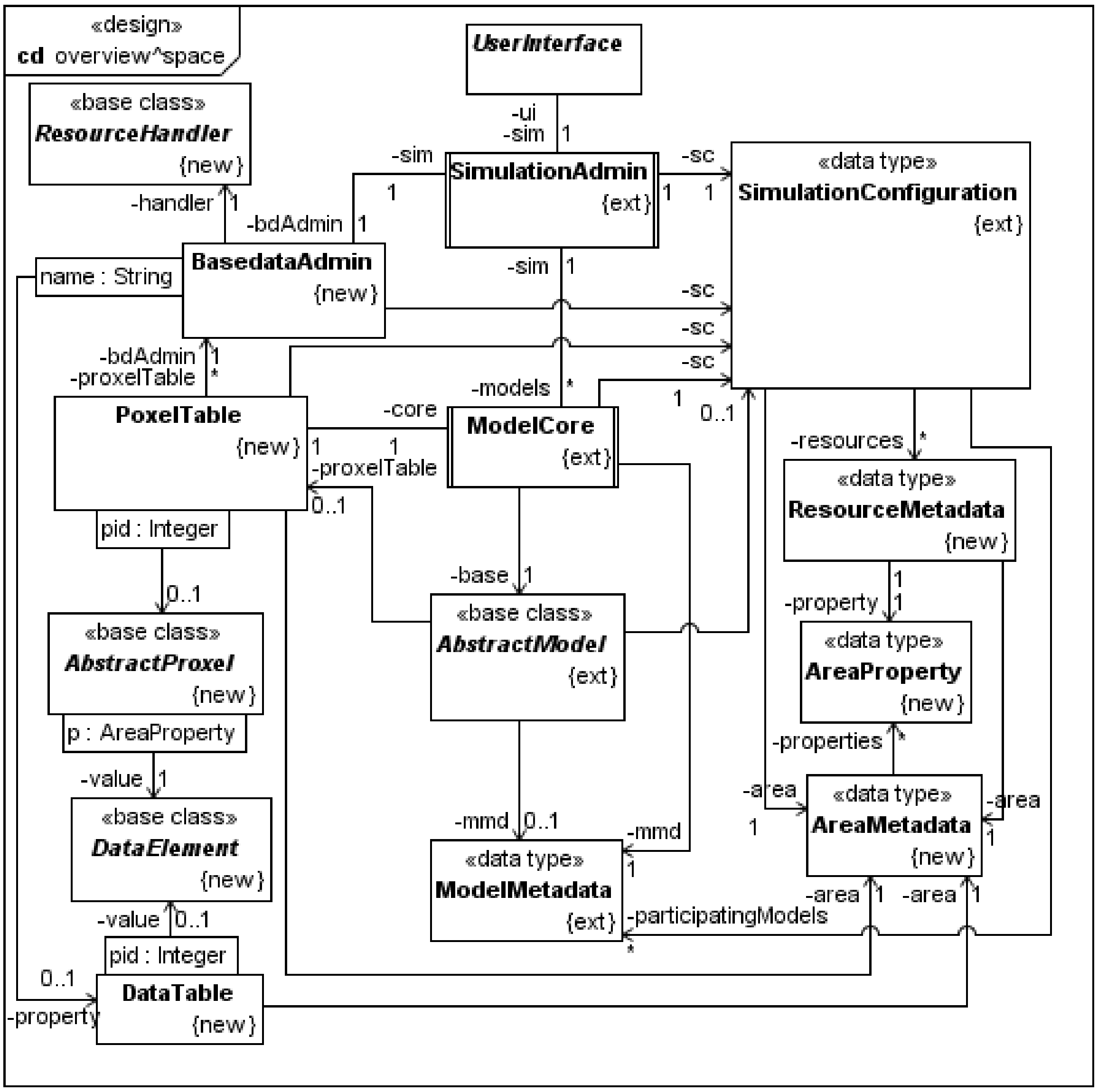

«design»
cd overview^space
UserInterface
«base class» ResourceHandler {new}
SimulationAdmin {ext}
«data type» SimulationConfiguration {ext}
BasedataAdmin {new}
name : String
PoxelTable {new}
pid : Integer
ModelCore {ext}
«data type» ResourceMetadata {new}
«base class» AbstractProxel {new}
p : AreaProperty
«base class» AbstractModel {ext}
«data type» AreaProperty {new}
«base class» DataElement {new}
«data type» ModelMetadata {ext}
«data type» AreaMetadata {new}
DataTable {new}
-ui
-sim
-sc
-handler
-bdAdmin
-proxelTable
-core
-models
-base
-mmd
-value
-property
-resources
-properties
-area
-participatingModels
1
*
0..1

Figure 6.3: Class design model in the view “simulation space” (overview)

We start on the top of the left hand side of the class diagram in Figure 6.3. The abstract base class ResourceHandler is responsible for loading data resources (like e.g. spatial initialisation data) into the system. An implementation of this class has to implement the plug-point loadResource which yields a DataTable object from a ResourceMetadata instance. The usage of the abstract base class ResourceHandler leaves undetermined from which place the data resources are loaded in a concrete system. This place might be a data base or just a file system.

The class BasedataAdmin provides a storage for the data resources used in the system. The simulation models can query this data by calling the operation getBasedata. The set of resources to be stored is determined by the respective SimulationConfiguration.

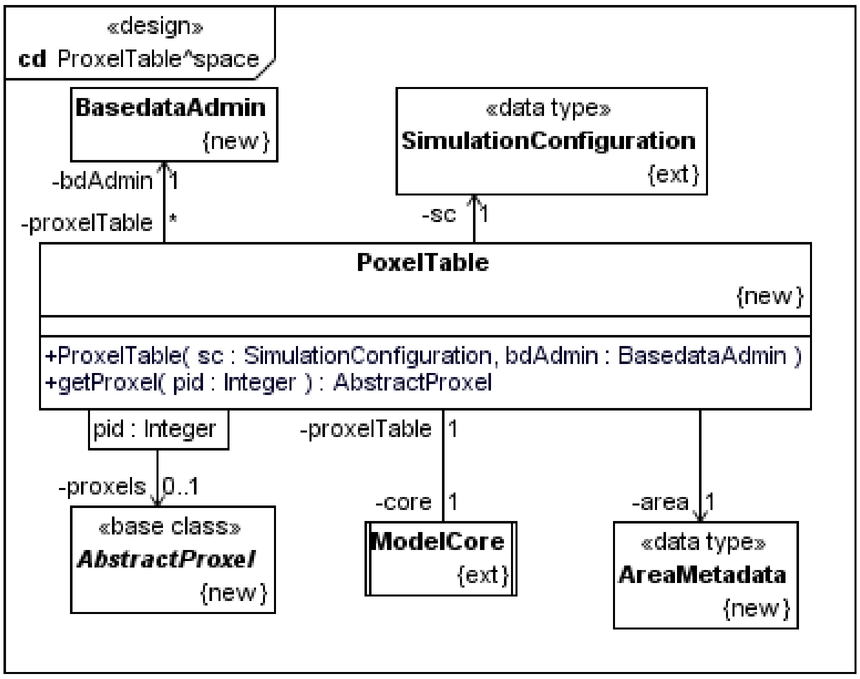


Figure 6.4: Details of the class ProxelTable

The class ProxelTable (cf. Figure 6.4) is one of the central classes of the view “simulation space” as it is responsible for the initialisation of the spatial part of a simulation model. The initialisation process is described in the following section. Each simulation model possesses exactly one ProxelTable instance. Moreover, the class ProxelTable stores a set proxels of AbstractProxel objects which may be queried by the simulation model to execute a computation on the specific proxel or to set or retrieve values of proxel properties. The cardinality of this set is determined by the property nrProxels of the class AreaMetadata which is, in turn, a property of a SimulationConfiguration, and hence has the same value for all participating models of the respective simulation. In particular that means that the operation getProxel yields a proxel object only if the parameter value is between zero and the number of proxels of the respective area. We express this by the following OCL condition:

```
context ProxelTable::getProxel(pid:Integer):AbstractProxel
   pre: pid>0 and pid<=area.nrProxels
   post: result<>null and result.pid=pid
```

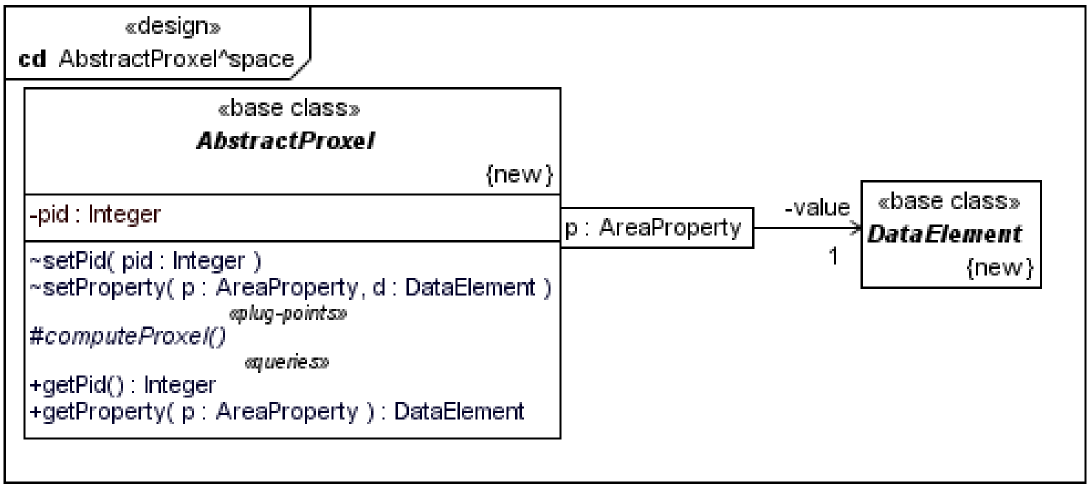


Figure 6.5: Details of the class AbstractProxel

The base class AbstractProxel represents an (abstract) unit of the simulation space (which we call *proxel*; cf. Section 6.1). The spatial unit is abstract in that sense that each simulation model has to define a concrete subclass of AbstractProxel to which it may add the spatial properties relevant for the specific model (see Section 9.3 for an example). In a concrete subclass the plug-point computeProxel has to be implemented appropriately.

There are two more classes shown on the left hand side of Figure 6.7, namely the classes DataElement and DataTable. While the base class DataElement describes an abstract data type (which has to be replaced by a concrete data type; see Section 9.2 for the use of data types) the class DataTable is intended for storing a map of DataElements where each element can be assigned to a certain proxel via the qualified association with the proxel identifier pid as qualifier.

On the right hand side of Figure 6.3 we find some classes marked with the stereotype «new» as well. The class ResourceMetadata describes a data resource (like, e.g. a file or a database table) which is, in turn, contains the initialisation values for a certain AreaProperty. A simulation area is described by the class AreaMetadata (cf. Figure 6.6), which contains besides an identifier areaId and a description of the area the number of proxels (property nrProxels) and a set of properties. Such a property might consist of, e.g., geographical coordinates, elevation, etc. For each property a data resource containing initial values has to provided by the SimulationConfiguration, which is reflected by the following excerpt of the postcondition of the query isValid:

```
context SimulationConfiguration::isValid()
```

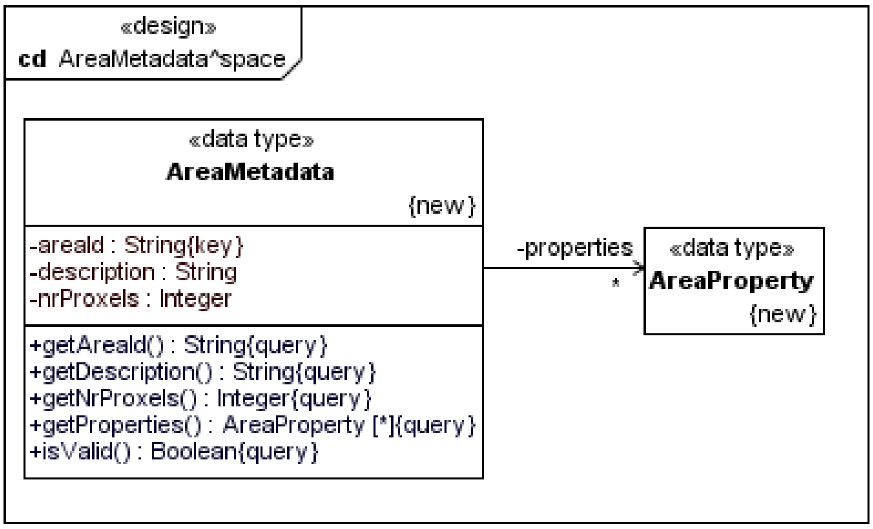

«design»
cd AreaMetadata^space
«data type»
AreaMetadata
{new}
-areaId : String{key}
-description : String
-nrProxels : Integer
+getAreaId() : String{query}
+getDescription() : String{query}
+getNrProxels() : Integer{query}
+getProperties() : AreaProperty [*]{query}
+isValid() : Boolean{query}
-properties
*
«data type»
AreaProperty
{new}

Figure 6.6: Details of the class AreaMetadata

```
  ...
  post:  self.area.properties->forAll(p |
    self.resources->exists(r | r.property = p))
  ...
```

For the sake of completeness we provide the complete postcondition of the query isValid and its dependent queries in the following.

```
context SimulationConfiguration::isValid()
 pre:true
 post:  result = self.simulationId <> ""
  and  self.participatingModels <> null
  and  self.participating.Models->forAll(m | m.isValid())
  and  self.area.isValid()
  and  self.resources->forAll(r | r.isValid())
  and  self.area.properties->forAll(p |
    self.resources->exists(r | r.property = p))

context ModelMetadata::isValid()
 pre:  true
 post:  result = self.modelId <> ""
   and  self.modelClass <> ""
   and  self.proxelClass <> ""

context AreaMetadata::isValid()
 pre:  true
```

```
  post: result = areaId <> ""
    and description <> ""
    and nrProxels > 0
    and properties <> null
    and properties->forAll(p | p.isValid())

context ResourceMetadata::isValid()
  pre:  true
  post: result = resourceId<>""
    and description <> ""
    and resourceType <> ""
    and resourceLocation <> ""
    and property.isValid()

context AreaProperty::isValid()
  pre: true
  post: name <> "" and type <> ""
```

6.2.2 Behavioural Design

An overview of the behaviour in the view "simulation space" is given in Figure 6.7. We notice that with respect to the base view the diagram is extended by the two referenced sequence diagrams initBasedataAdmin^space and initModel^space. We will detail on these two diagrams in the following, while for the details of the other diagrams we refer to Appendix A.3.2.

The interactions of the sequence diagram initBasedataAdmin^space in Figure 6.8 describe how the data resources are loaded into the system. First, the necessary resources are queried from the SimulationConfiguration. Then a new ResourceHandler object is created from which – within a loop over all ResourceMetadata objects obtained from the simulation configuration – the needed resources are requested by calling the operation loadResource. The return value of this operation should be a DataTable object containing the requested data.

Of course it might occur that the requested resource does not exist or cannot be found. For this case the postcondition of loadresource allows the return value to be null. This exceptional situation is caught by the following break fragment which is executed under the condition dt=null where dt denotes the return value of loadResource. In this case the course of action continues (and terminates[1]) with the usual exception handling by means of the interaction use handleException (cf. Figure 4.8).

Let us now consider the sequence diagram initModel^space depicted in Figure 6.9. It describes how the spatial aspects of simulation model are initialised. First, a ProxelTable instance is created by the respective ModelCore. Then, within a loop ranging over the proxel identifiers of the respective simulation area, a Proxel object is created for each proxel id. The postcondition of the constructor ensures that the created object is either null or of the correct type, i.e.

[1]Remember our semantics of a break fragment defined in Section 3.1.2.

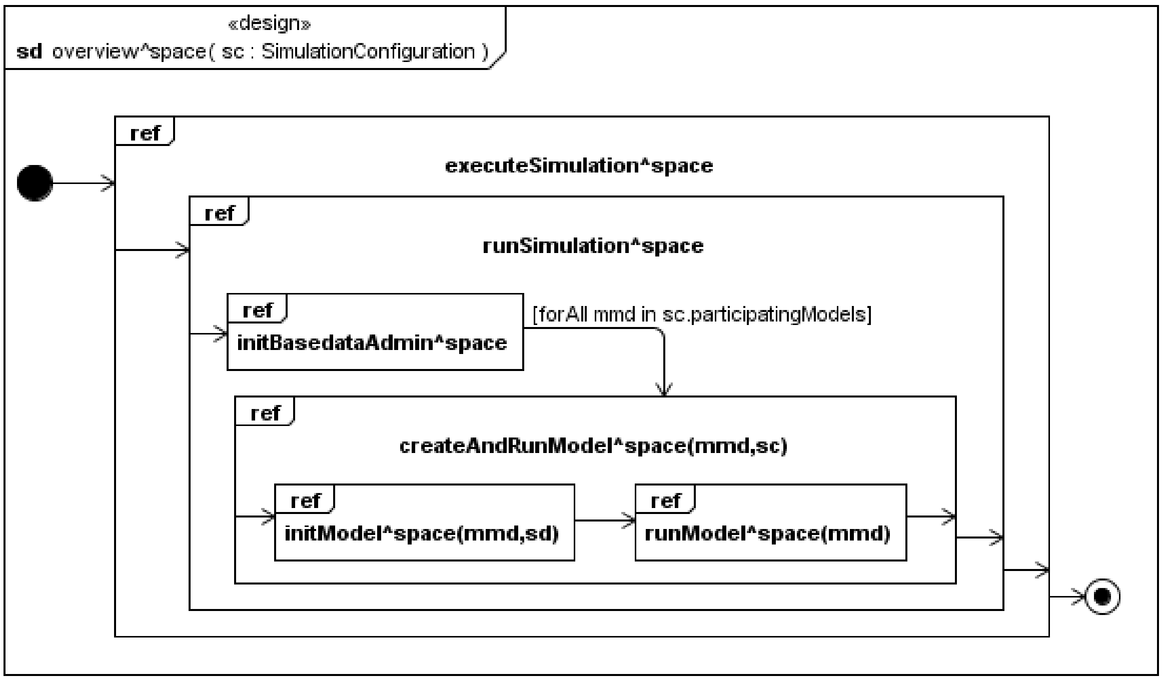


Figure 6.7: Interaction overview diagram on the view “simulation space”

the type specified by the property proxelClass of the ModelMetadata.

If the proxel object is null – this might be the case if the corresponding class definition cannot be found – the course of action continues (and terminates) as defined by the break fragment by calling the formal gate exception on the interaction use handleException (cf. Figure 4.8 in Section 4.3.2).

In the normal case, i.e. when the proxel objects have been created correctly, the course of action continues as follows. Within another loop – this time ranging over the properties of the simulation area – the proxels are initialised with data. For each property p a DataTable object dt is requested from the BasedataAdmin. Note that the object dt cannot be null, as this was checked already when initialising the BasedataAdmin. Within a nested loop, again ranging over the set of proxel identifiers, the values contained in dt are distributed over the proxels by calling the setter setProperty. Note that the spatial structures of ProxelTable and DataTable coincide, that means that a value of a DataTable stored under pid belongs to a proxel object stored under the same pid in the ProxelTable.

Finally, the initialised ProxelTable object is stored as property proxelTable of ModelCore (cf. the respective state invariant on the lifeline of ModelCore) as well as it is set as property of AbstractModel by calling the setter setProxelTable.

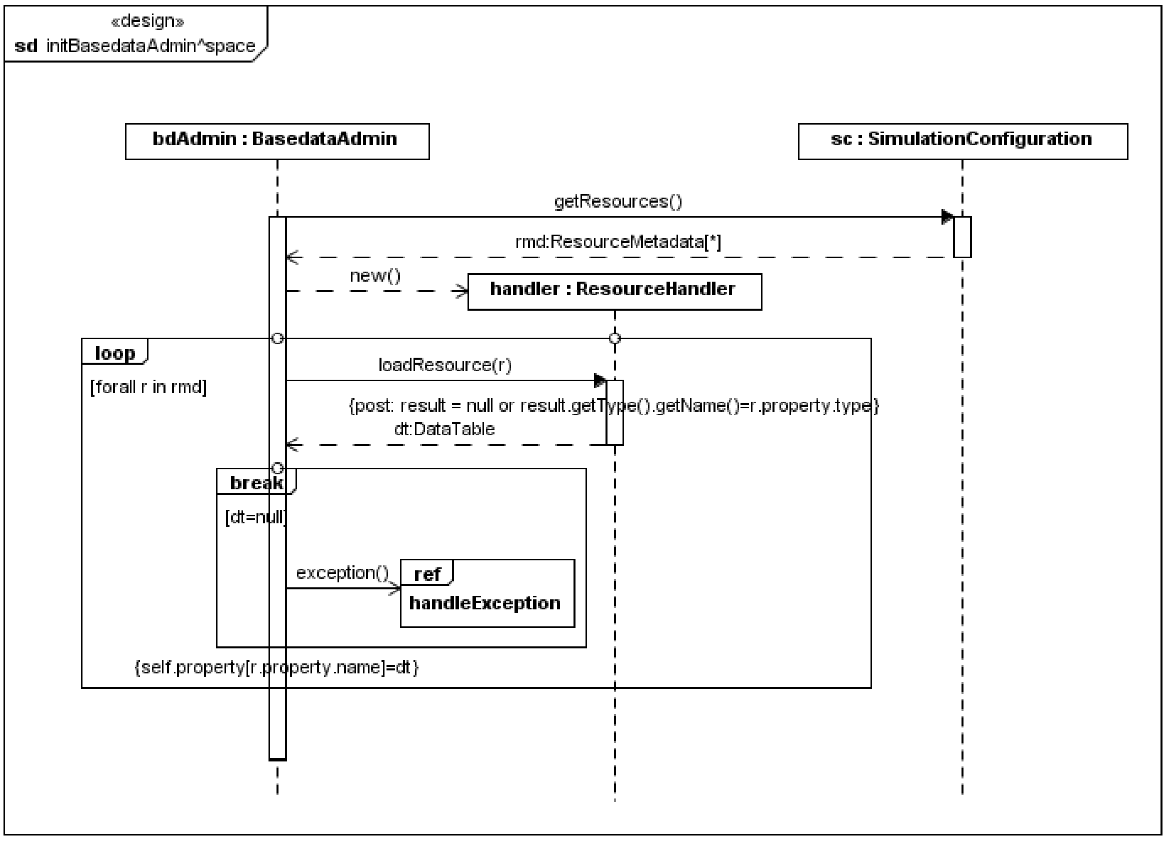

«design»
sd initBasedataAdmin^space
bdAdmin : BasedataAdmin
sc : SimulationConfiguration
getResources()
rmd:ResourceMetadata[*]
new()
handler : ResourceHandler
loop
[forall r in rmd]
loadResource(r)
{post: result = null or result.getType().getName()=r.property.type}
dt:DataTable
break
[dt=null]
exception()
ref
handleException
{self.property[r.property.name]=dt}

Figure 6.8: Sequence diagram initBasedataAdmin^space

6.3 Components

The component architecture of the view "simulation space" can be found in Figure 6.10. The details of the single components are depicted in Appendix A.3.3.

With respect to the base view the components Basedata and Proxel have been added. While Basedata contains the classes regarding the initialisation of the data resources, the component Proxel comprises the spatial aspects of a simulation model. Hence this component is designed as a subcomponent of Model.

6.4 Conformance with the Development Methodology

In this chapter we have elaborated the architecture of the view "Modelling of the Simulation Space" (or space view for short) which corresponds to the following extension and refinement

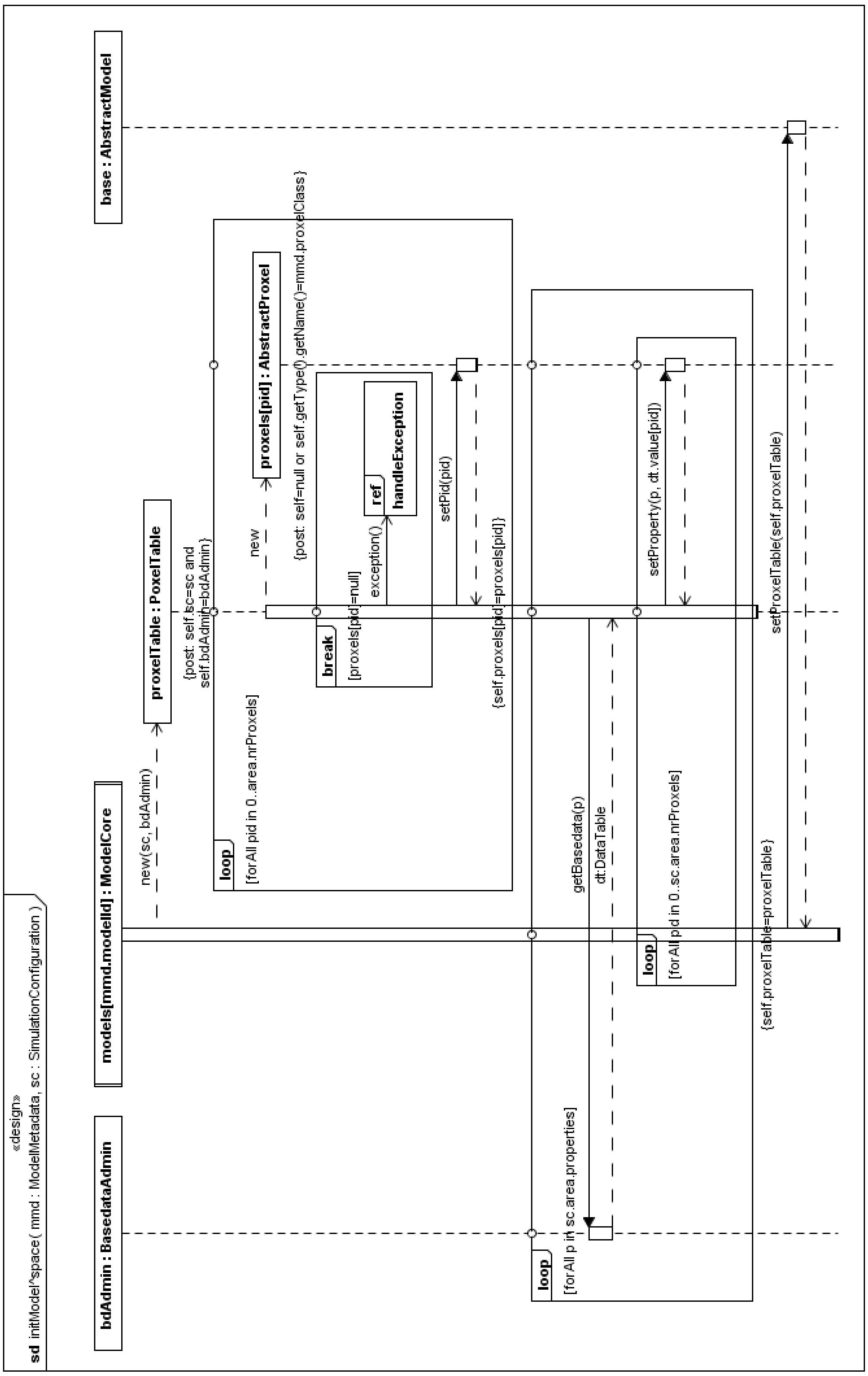

«design»
sd initModel^space(mmd : ModelMetadata, sc : SimulationConfiguration)
bdAdmin : BasedataAdmin
models[mmd.modelId] : ModelCore
proxelTable : PoxelTable
proxels[pid] : AbstractProxel
base : AbstractModel
new(sc, bdAdmin)
{post: self.sc=sc and self.bdAdmin=bdAdmin}
new
{post: self=null or self.getType().getName()=mmd.proxelClass}
loop
[forAll pid in 0..area.nrProxels]
break
[proxels[pid]=null]
exception()
ref
handleException
setPid(pid)
{self.proxels[pid]=proxels[pid]}
loop
[forAll p in sc.area.properties]
getBasedata(p)
dt:DataTable
loop
[forAll pid in 0..sc.area.nrProxels]
setProperty(p, dt.value[pid])
setProxelTable(self.proxelTable)
{self.proxelTable=proxelTable}

Figure 6.9: Sequence diagram initModel^space

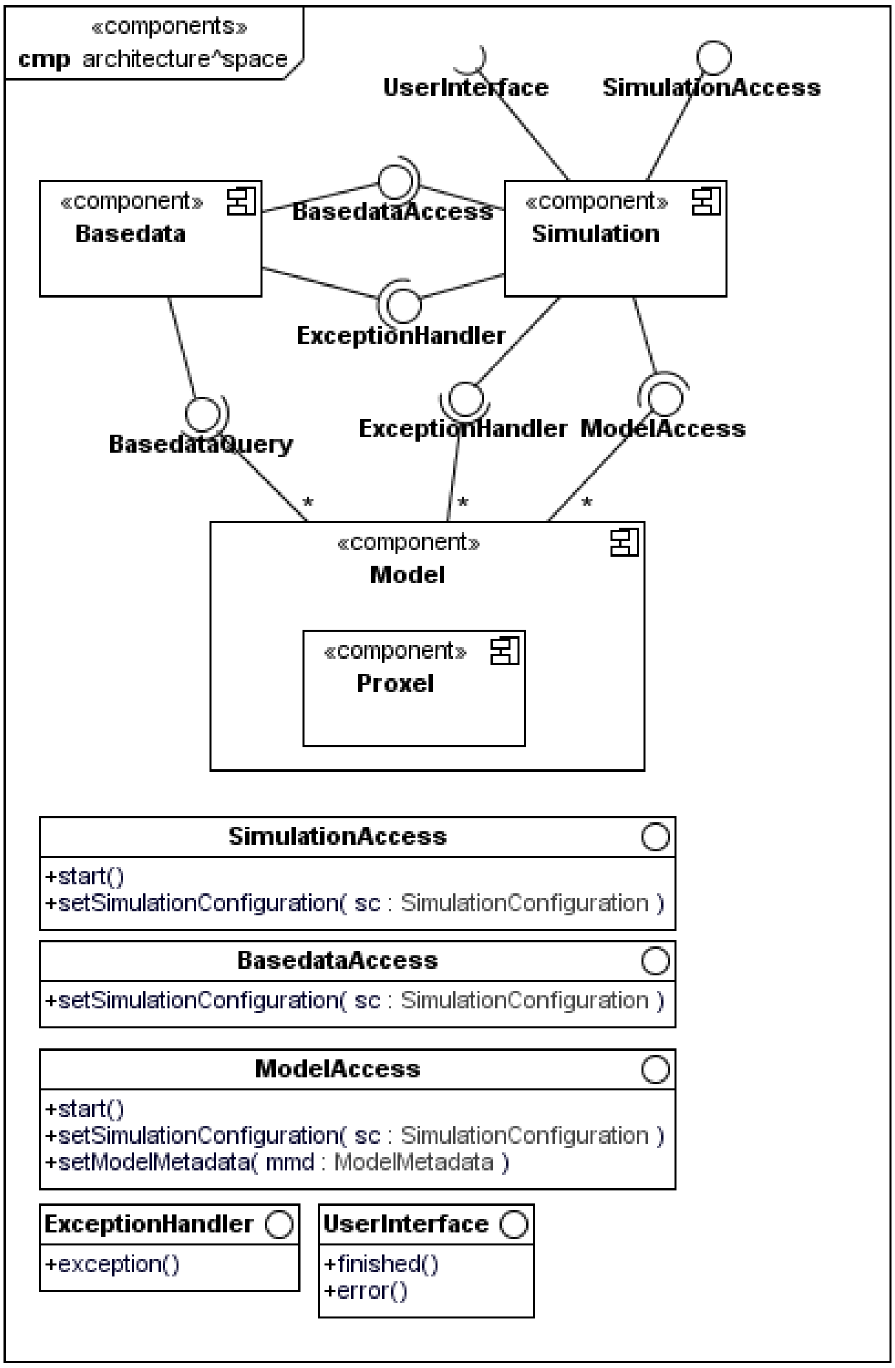


Figure 6.10: Architecture in the view “simulation space”

steps according to our development methodology.

$$\begin{tikzcd}
\text{req}^{base} \arrow[r, rightsquigarrow] \arrow[d] & \text{des}^{base} \arrow[r, rightsquigarrow] \arrow[d] & \text{cmp}^{base} \arrow[d] \\
\text{req}^{space} \arrow[r, rightsquigarrow] & \text{des}^{space} \arrow[r, rightsquigarrow] & \text{cmp}^{space}
\end{tikzcd}$$

Again, as the refinement of the base view has already been shown in Section 4.5, one has to prove that

1. the abstraction levels of the space view are in refinement relation, and
2. each abstraction level of the space view is an extension of the respective level of the base view.

In both cases we leave the proof to the reader as they can be produced analogously to those provided in Sections 4.5 and 5.4 respectively.

6.5 Discussion

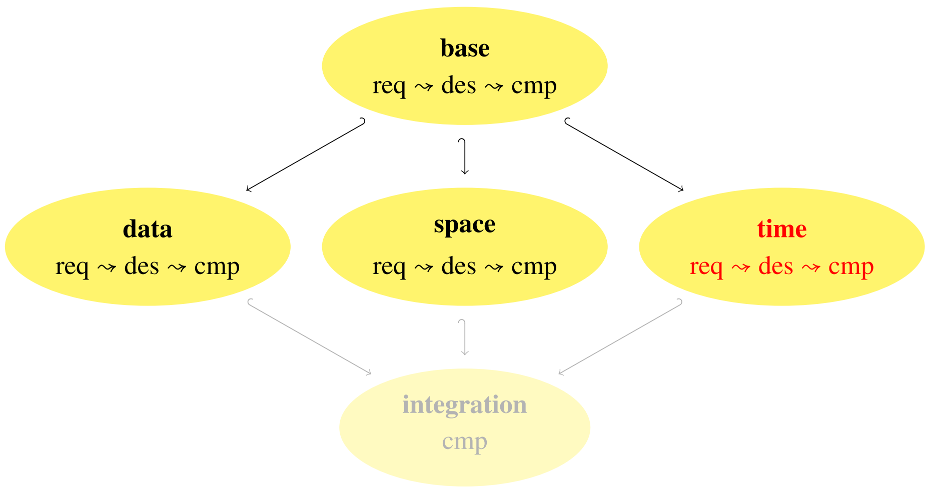
The spatial resolution is – beside the time step – one of the most important properties of a simulation model. When coupling models with different spatial resolution the consistency of exchanged data has to be taken into account.

There are several possibilities to ensure consistency. First, each model operates on its own spatial resolution and either the export model, the import model, or a dedicated conversion component is responsible for converting the spatial data to the appropriate resolution. In this case one has to bear in mind that converting spatial data is not a trivial task, in particular when converting from a coarse-grained resolution to a more fine-grained one Second, each model operates – at least with respect to the data to be exchanged – on a common spatial model.

Our approach covers – on the demand of the underlying research project – a common spatial modelling for the exchanged data with consistent initialisation of common base data handled on framework level. This implies of course more effort for model developers in converting spatial data to their appropriate resolution, but yields more reliable simulation results. The approach of OMS (cf. [KKO05]) which uses so called Spatial Compound Components as spatial units is the most similar approach to ours. In contrast, ModCom (cf. [HBvEL03]) does not support spatial modelling on the framework level at all, delegating this to the single models. TIME (cf. [RSP+03]) mainly supports raster based spatial models, while OpenMI (cf. [GGW07]) again uses a generic approach by defining an interface ElementSet which covers spatial properties.

7

View "Time Coordination of Integrative Simulations"



An important characteristics of our problem domain is the concurrent execution of different simulation models which iteratively exchange information at run time via their interfaces. In order to guarantee the consistency of data exchange during a simulation run, the single simulation models must be appropriately coordinated with respect to the progressing simulation time. The *correct* coordination is a non-trivial task since, in general, simulation models have different, individual time steps determining the model time between two consecutive computations. Model time steps depend, of course, on the simulated processes which typically range from minutes or hours, like in natural sciences, to months, like in social sciences. Hence a precise, unambiguous specification of the coordination problem is mandatory.

In this chapter we provide such a specification by means of the process algebra *Finite State Processes* (FSP) [MK06] which has been presented by the author in [HL05] and [HL06].

The usage of FSP has several advantages: it permits a clear distinction between the (non-constructive) specification of requirements and the specification of a design model; the formal requirements can be validated with respect to the intuitive requirements by visualising them as labelled transition systems; FSP comes along with a model checking tool *Labelled Transition System Analyser* (LTSA) [LTS11] which allows for checking the correctness of the design model with respect to the requirements. An example of a formalisation of the coordination problem on a meta level using purely mathematical notations is given in [BK04].

This chapter is again organised according to our methodology provided in Chapter 3, hence we have a section stating the requirements and concepts of the view (Section 7.1), one where the class design is developed (Section 7.2) and finally there is a section presenting the component design (Section 7.3). The chapter closes with a brief discussion of our and related approaches for time coordination (Section 7.5).

7.1 Requirements

We first extend the static concepts model of the base given in Figure 4.2. The result of the extension is depicted in Figure 7.1 and explained in the following. A simulation model simulates a physical or social process for a certain period of time which we call *simulation time*. The simulation time is finite which means that there is always a begin and an end time, modelled by the respective attributes of the class Simulation. As in our approach only time-discrete simulation models are considered, we can represent the whole simulation period by a strictly ordered, discrete set of points in time, at which data is provided by a simulation model. Each model has an individual *time step* (cf. attribute timeStep of the class Model) which determines the distance between two subsequent simulation points. For instance, a meteorological model provides the air temperature every hour, while a groundwater model provides the amount of groundwater withdrawal only once a day. We assume that the time step of a model remains fixed during the whole simulation.

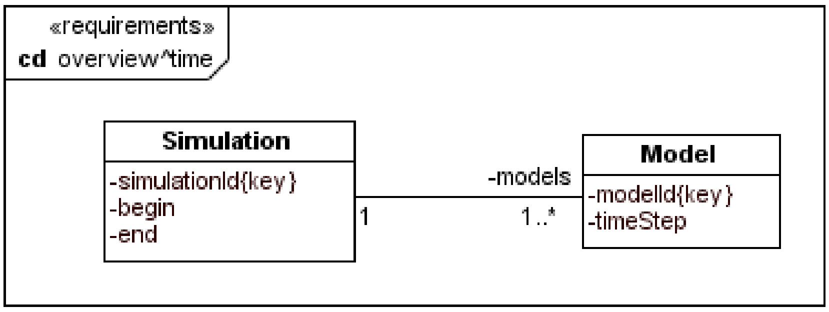


Figure 7.1: Concept model of the view "time coordination"

In contrast to a stand-alone simulation model, a coupled simulation model not only computes data, but rather has to perform activities concerning data exchange. The general life cycle that a coupled simulation model must follow can be described as follows.

- *provide* initial data at the model's provided interfaces
- while not at simulation end
  - *get data* from the model's required interfaces
  - *compute* new data for the next time step
  - *provide* newly computed data at the model's provided interfaces

In the sequence diagram in Figure 7.2 the interaction executeSimulation of the base (cf. Figure 4.3) is extended by this life cycle, but without taking into account any coordination effort. Before we address the problems arising from this naive approach of coupling models we develop a formal description of a simulation model.

7.1.1 Formalisation of a Simulation Model

For the formalisation of a model's life cycle we use the process algebra *Finite State Processes* FSP [MK06] which is based on the process algebra *Communicating Sequential Processes* (CCS) [Hoa85]. The (discrete) simulation time is modelled by natural numbers. The following FSP process MODEL specifies the general behaviour of a simulation model. In order to be generally applicable the process is parametrised with respect to the model's time step. Note that in the process definition we have to provide a default time step (e.g. Step = 1) which is necessary according to the finite states assumption of FSP. For the same reason it is necessary to model the simulation start and the simulation end by some predefined constants. The sequence of actions in line 8, getData[t] -> compute[t+Step] -> provide[t+Step], is iteratively performed with increasing time t and thus formalises the iteration in the informal description of a model's life cycle given above. Note that the computation of new data for time t+Step relies on data obtained for time t. This time difference avoids deadlocks of concurrently running models (in the case of feedback loops) but it may also lead to imprecisions whose relevance must be analysed in concrete cases and, if necessary, can be solved by using smaller time steps.

```
 1 const SimStart = 0
 2 const SimEnd = 6
 3 range SimTime = SimStart..SimEnd
 4
 5 MODEL(Step = 1) = (start -> provide[SimStart] -> M[SimStart]),
 6 M[t:SimTime] =
 7    if (t+Step <= SimEnd)
 8    then (getData[t] -> compute[t+Step] -> provide[t+Step] ->
 9       M[t+Step])
10    else (finish -> STOP).
```

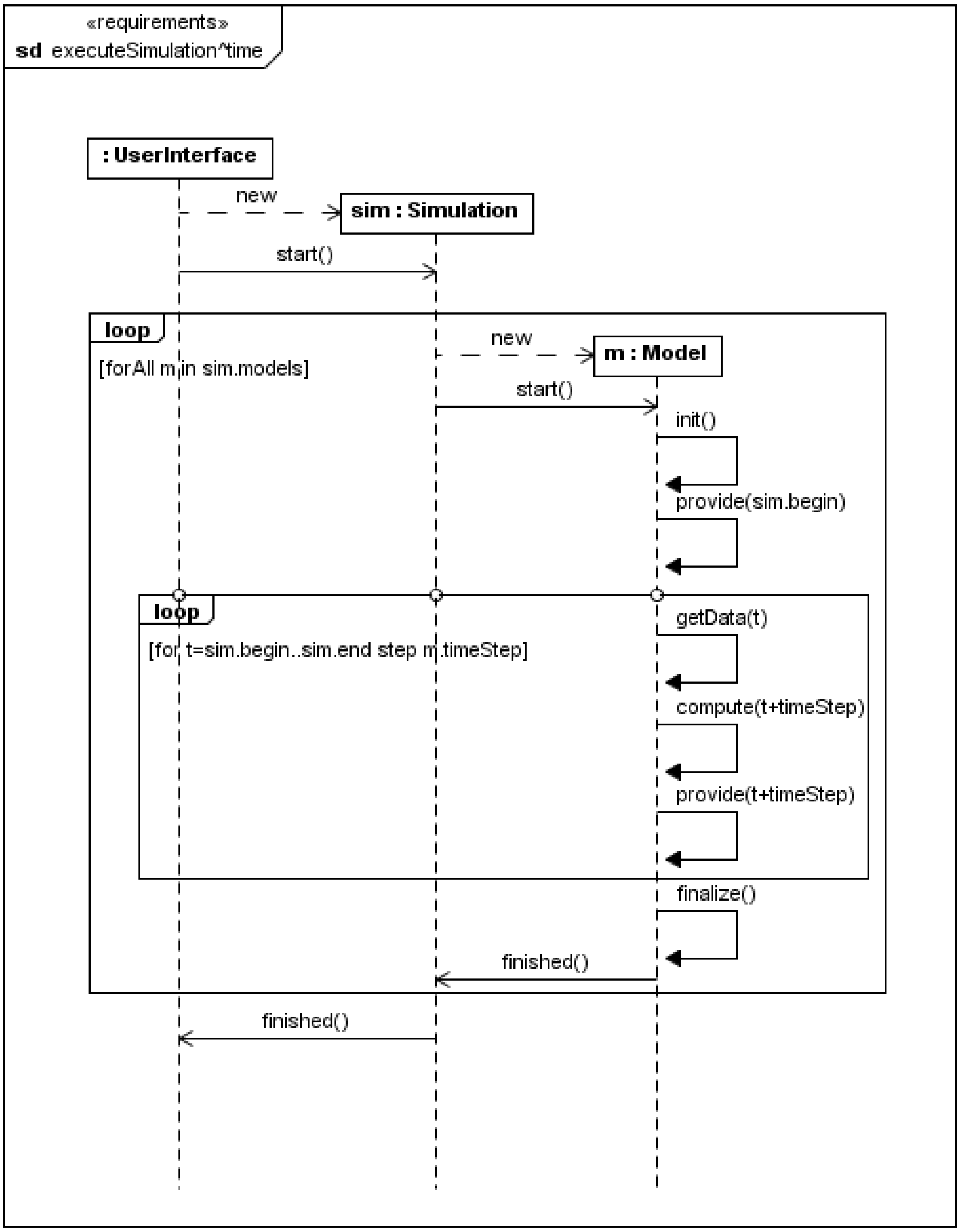


Figure 7.2: Conceptual sequence diagram of the view “time coordination”

In the above process description the (indexed) actions prov[x] represent providing of export data which are valid at time x, the actions get[x] represent getting of import data which are valid at time x and the actions compute[x] represent the computation of new data based on import data which are valid at time x. Indeed the choice of the time dependent indices of the actions is crucial for the behaviour of the whole system to be developed. To explain our choice let us assume for the moment that the simulation time is a multiple of the model's time step. Then, according to the above process description, the last data that a model gets is valid at time $SimEnd - Step$ and the last data a model provides is valid at time $SimEnd$. For the whole simulation, this means that imported data is considered to be *last recently valid* for the computation of new export values to be valid at time t if the imported data is valid at time $t - Step$.

Of course, there are other choices for the definition of last recently valid data. For instance, the intuitively best choice would be to require that the imported values used for the computation of exported values to be valid at time t are also valid at time t (instead of being valid at time $t - Step$). But then the analysis of any attempt to construct a design model for the coordination problem will show that there is no deadlock-free solution (whenever there are, as usual, mutually dependent export and import data).

To represent a particular instance of a simulation model we have to provide a model name (model identifier) and the particular time step of the model under consideration. For specifying model identifiers we use process labels and the time step of a model is determined by an actual parameter. For instance, the FSP processes $[1]:\text{MODEL}(2)$ and $[2]:\text{MODEL}(3)$ represent two simulation models, one with number 1 and time step 2 and the other one with number 2 and time step 3, resp. The behaviour of model 2 is illustrated by the LTS in Figure 7.3.

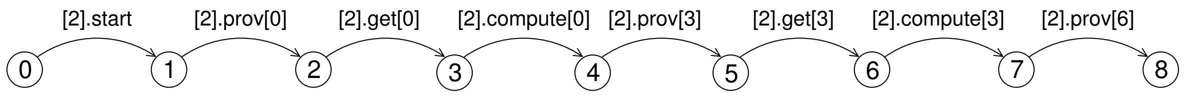

$$\begin{tikzcd}
0 \arrow[r, bend left, "{[2].start}"] & 1 \arrow[r, bend left, "{[2].prov[0]}"] & 2 \arrow[r, bend left, "{[2].get[0]}"] & 3 \arrow[r, bend left, "{[2].compute[0]}"] & 4 \arrow[r, bend left, "{[2].prov[3]}"] & 5 \arrow[r, bend left, "{[2].get[3]}"] & 6 \arrow[r, bend left, "{[2].compute[3]}"] & 7 \arrow[r, bend left, "{[2].prov[6]}"] & 8
\end{tikzcd}$$

Figure 7.3: LTS of a simulation model

7.1.2 The Coordination Problem

In an integrative simulation various simulation models work together by mutually exchanging data via their import and export ports. Each of the participating models performs a local simulation for the same overall time period (the global simulation time) but has usually a different (local) time step. It is crucial for integrative simulations that each model gets, whenever needed, the last recently valid data from partner models. A first attempt to model an integrative simulation could be to simply combine the processes which represent the single simulation models by parallel composition. For instance, for the two simulation models from above we would obtain the following composite process:

```
const NrModels = 2
```

```
range Models = 1..NrModels

||SYS = ([1]:MODEL(2)||[2]:MODEL(3))/{start/[Models].start}
```

The relabelling clause { start /[Models]. start } ensures that the processes synchronise on the start action. Let us now consider some possible execution traces of the composite process which illustrate three characteristic problems that we have to take into account when we want to specify the desired safety properties for the system.

1. *Missing import data*

$$\mathit{start} \to [1].\mathit{prov}[0] \to [1].\mathit{get}[0] \to \ldots$$

   Model 1 gets data while model 2 has not yet provided data.

2. *Obsolete import data*

$$\begin{aligned} \mathit{start} &\to [2].\mathit{prov}[0] \to [1].\mathit{prov}[0] \to [1].\mathit{get}[0] \to [1].\mathit{compute}[2] \\ &\to [1].\mathit{prov}[2] \to [1].\mathit{get}[2] \to [1].\mathit{compute}[4] \to [1].\mathit{prov}[4] \\ &\to [1].\mathit{get}[4] \to \ldots \end{aligned}$$

   Model 1 gets data expected to be valid at time 4 while the last data provided by model 2 was valid at time 0 and model 2 has not yet provided data valid at time 3 (which would be the last recently valid data according to the time step of model 2).

3. *Overwritten import data*

$$\begin{aligned} \mathit{start} &\to [2].\mathit{prov}[0] \to [1].\mathit{prov}[0] \to [2].\mathit{get}[0] \to [2].\mathit{compute}[3] \\ &\to [2].\mathit{prov}[3] \to [1].\mathit{get}[0] \to \ldots \end{aligned}$$

   Model 1 gets data expected to be valid at time 0 while model 2 has already provided data that is valid at time 3.

In the following we provide a formalisation of the coordination problem in terms of safety and liveness conditions.

Safety Properties

We start by formalising the corresponding synchronisation conditions by means of FSP property processes. The crucial idea is that the problem can be simplified if we consider only two simulation models at a time and, moreover, if we consider each of the two models only under one particular aspect, either as a provider or as a user of information. In the following let U denote a user model and let P denote a provider model. From the user’s point of view we obtain the following condition (1), from the provider’s point of view we obtain condition (2).

(1) U gets data expected to be valid at time t_U only if the following holds:
P has last provided data valid at time $last_P$ with $last_P \leq t_U$ and the next data that P provides is valid at time t_P with $t_U < t_P$.

(2) P provides data valid at time t_P only if the following holds:
The next data that U gets is expected to be valid at time t_U with $t_U \geq t_P$.

An execution trace w of an integrative simulation with an arbitrary number of simulation models $[1]:\text{MODEL}(Step_1),\ldots,[n]:\text{MODEL}(Step_n)$ is called *legal* with respect to a user U and a provider P, if w meets the above requirements (1) and (2). We model the legal execution traces by a generic FSP property process which is parameterised with respect to the model number and the time step of the user and the provider model respectively.

```
property VALIDDATA(User=1,StepUser=1,Prov=1,StepProv=1) =
  VD[SimStart][SimStart],

VD[nextGet:Time][nextProv:Time] =
  (when (nextProv-StepProv<=nextGet \& nextGet<nextProv)
      [User].get[nextGet] -> VD[nextGet+StepUser][nextProv]
  |when (nextGet>=nextProv)
      [Prov].prov[nextProv] -> VD[nextGet][nextProv+StepProv]).
```

The first alternative of the property process formalises condition (1) from above where the index variable nextUser corresponds to t_U, nextProv corresponds to t_P and nextProv−StepProv corresponds to $last_P$. The second alternative formalises condition (2) from above. For the sake of simplicity we did not take into account the end of a simulation in the above process definition. For this purpose the process can be appropriately extended in order to avoid index overflow when the simulation end is reached and to ensure that the user and the provider have a clean termination.

All system requirements concerning the validity of data are now obtained by pairwise instantiations of the generic property process VALIDDATA. As an example let us consider model 1 with time step 2 as a user and model 2 with time step 3 as a provider. The corresponding safety property is then given by the property process VALIDDATA(1,2,2,3). The labelled transition system of this process is shown in Figure 7.4.

Labelled transition systems assigned to property processes have an error state, pictorially represented by −1, and are complete in the sense that for any action and any state (apart from the error state) there is always an outgoing transition. This transition leads to the error state if it is not properly defined in the property process definition. Thus the legal and illegal execution traces determined by a property process are revealed. For instance, the three example traces considered in Section 7.1.2 are illegal w.r.t. the property process VALIDDATA(1,2,2,3), because their restrictions to the alphabet of VALIDDATA(1,2,2,3) lead to the error state.

Besides the requirements concerning the validity of exchanged data we have to cope also with data access. Since, in reality, getting and providing data are non-atomic actions we have to ensure that a model gets data only if no other model provides data at the same time and

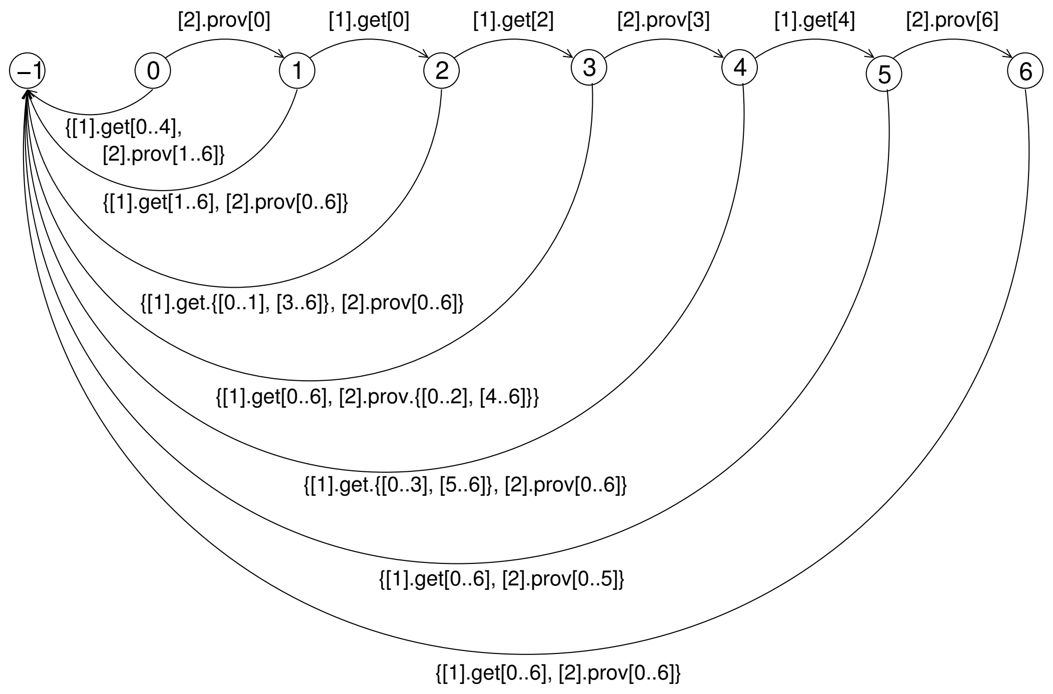

[2].prov[0]
[1].get[0]
[1].get[2]
[2].prov[3]
[1].get[4]
[2].prov[6]
−1
0
1
2
3
4
5
6
{[1].get[0..4], [2].prov[1..6]}
{[1].get[1..6], [2].prov[0..6]}
{[1].get.{[0..1], [3..6]}, [2].prov[0..6]}
{[1].get[0..6], [2].prov.{[0..2], [4..6]}}
{[1].get.{[0..3], [5..6]}, [2].prov[0..6]}
{[1].get[0..6], [2].prov[0..5]}
{[1].get[0..6], [2].prov[0..6]}

Figure 7.4: LTS of the property process VALIDDATA(1,2,2,3)

vice versa. To formalise mutual exclusion we first enclose the critical regions, which in our case are represented by the get and prov actions, by corresponding enter and exit actions. For this purpose the process definition for simulation models given above is slightly adapted in the following way.

```
MODEL(Step = 1) = (start -> INIT),
INIT = (enterProv[SimStart] -> prov[SimStart] ->
        exitProv[SimStart] -> M[SimStart]),
M[t:Time] =
   if (t+Step <= SimEnd)
   then (enterGet[t] -> get[t] -> exitGet[t] ->
         compute[t+Step] ->
         enterProv[t+Step] -> prov[t+Step] ->
         exitProv[t+Step] -> M[t+Step])
   else  STOP + {Labels}.
```

where

```
set GetProvs = {{get, prov}[Time]}
set EnterExits = {{enterGet, exitGet, enterProv, exitProv}[Time]}
set Labels = {GetProvs, EnterExits}
```

Note that the alphabet extension by Labels is necessary for technical reasons because the alphabet of property processes must be included in the alphabet of processes to be checked. By means of the enter and exit actions the desired exclusion conditions can now be expressed by a further property process, called EXCLUSION, which follows a standard scheme; cf. [MK06].

```
const NrModels = 2
range Models = 1..NrModels
range CountModels = 0..NrModels

property EXCLUSION =
  ([Models].enterGet[Time] -> GET[1]
  |[Models].enterProv[Time] -> PROV[1]),
GET[i:CountModels] =
  ([Models].enterGet[Time] -> GET[i+1]
  |when (i>1) [Models].exitGet[Time] -> GET[i-1]
  |when (i==1) [Models].exitGet[Time] -> EXCLUSION),
PROV[i:CountModels] =
  ([Models].enterProv[Time] -> PROV[i+1]
  |when (i>1) [Models].exitProv[Time] -> PROV[i-1]
  |when (i==1) [Models].exitProv[Time] -> EXCLUSION).
```

Liveness Properties

In contrast to the safety properties it is easy to identify the required liveness properties for integrative simulations. Obviously, we want that each simulation model provides data during the whole simulation period at any time that fits to its local time step. More formally, this means that for all execution traces w of an integrative simulation, for all models $m \in Models$ and for each time $t \in Time$ with $t\%Step_m = 0$ we have $[m].\mathrm{prov}[t] \in w$.

7.2 Design

The specification of the system requirements of the last section is highly non-constructive. In this section we firstly focus on a formal solution of the coordination problem in terms of an FSP process for which we can verify that the resulting system fulfils the requirements, and then show how the formal design model can systematically be transformed to an UML class design model.

7.2.1 Formal Design Model

The basic idea of the formal design model is to introduce a global time-controller that coordinates appropriately all simulation models participating in an integrative simulation. More precisely, we want to design an FSP process, called TIMECONTROLLER, such that for n simulation models the composite process

$$\begin{aligned} \|\text{SYS} &= ([1]:\text{MODEL}(Step_1)\|\ldots\|[n]:\text{MODEL}(Step_n)\| \\ &\quad \text{TIMECONTROLLER}(Step_1,\ldots,Step_n))/\{\text{start}/[\text{Models}].\text{start}\} \end{aligned}$$

restricts the execution traces of the uncontrolled simulation models to the legal ones. The composite process SYS is then considered as the design model for the system. The (static) structure of SYS is represented by the diagram in Figure 7.5 which indicates the required communication links.

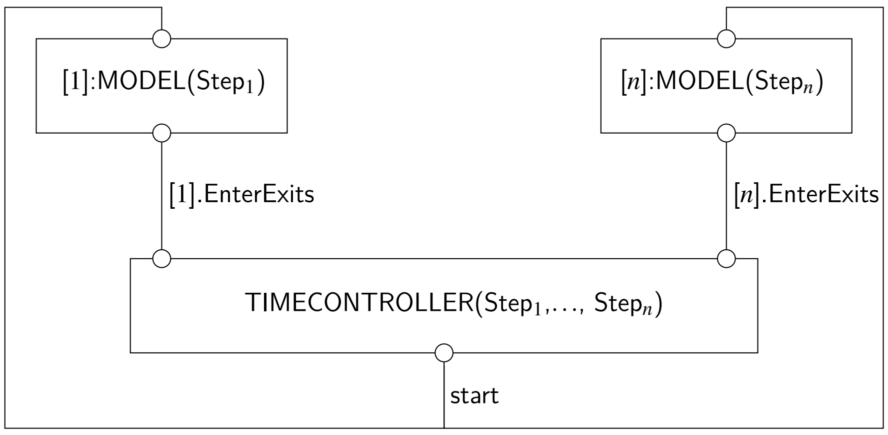


Figure 7.5: Structure diagram of the design model

The communication links show that each simulation model m communicates with the timecontroller via the shared enter and exit actions in the (labelled) set [m].EnterExits (see Section 7.1.2 for the definition of EnterExits). This means that the simulation models synchronise with the timecontroller on actions of the form [m].enterGet[t] etc., where $m \in$ Models and $t \in$ Time. It is then the task of the timecontroller to guarantee that synchronisation can only occur if the constraints determined by *all* property processes (given in Section 7.1) are satisfied. For this purpose the enter actions of the timecontroller are guarded by appropriate conditions which monitor the validity of the safety properties. To express the necessary conditions the timecontroller is equipped with a local state (modelled by index variables) which records the execution status of all simulation models to be coordinated. More precisely, the timecontroller stores for each model the time for which it gets the next import data (represented by the index nextGet) and the time for which the model will provide the next export data (represented by the index nextProv).

The following time controller definition is formulated for the case of two simulation models where the time steps of the two models are given by parameters. It is obvious that this description provides a general pattern which can be easily applied to an arbitrary number of simulation models. For a timecontroller definition which is generic w.r.t. the number of simulation models one would need array types which are not available in FSP (but would be available in SPIN [Hol04]). Let us still remark that the guards of the enter actions are inferred from the requirements specification by building the conjunction of the guards occurring in the property processes for the validity of data. Moreover, note that model checking shows that the exclusion property for get and prov is already guaranteed by these conditions and therefore does not need a special treatment.

```
TIMECONTROLLER(Step1=1,Step2=1) =
  (start -> TC[SimStart][SimStart][SimStart][SimStart]),

TC[nextGet1:Time][nextProv1:Time]
    [nextGet2:Time][nextProv2:Time] =
  (dummy[t:Time] ->
    //enterGet
    (when (nextProv1-Step1<=t \& t<nextProv1 \&
           nextProv2-Step2<=t \& t<nextProv2)
           [Models].enterGet[t] ->
           TC[nextGet1][nextProv1][nextGet2][nextProv2]
    //exitGet
    |[1].exitGet[t] -> TC[t+Step1][nextProv1]
                          [nextGet2][nextProv2]
    |[2].exitGet[t] -> TC[nextGet1][nextProv1]
                          [t+Step2][nextProv2]
    //enterProv
    |when (nextGet1>=t \& nextGet2>=t)
           [Models].enterProv[t] ->
           TC[nextGet1][nextProv1][nextGet2][nextProv2]
    //exitProv
    |[1].exitProv[t] ->
           if (t+Step1<=SimEnd)
           then TC[nextGet1][t+Step1][nextGet2][nextProv2]
           else TC[SimStart][SimStart][SimStart][SimStart]
    |[2].exitProv[t] ->
           if (t+Step2<=SimEnd)
           then TC[nextGet1][nextProv1][nextGet2][t+Step2]
           else TC[SimStart][SimStart][SimStart][SimStart]
    |dummy[t] -> TC[nextGet1][nextProv1][nextGet2][nextProv2])
  )\{dummy[Time]}.
```

Let us still comment the role of the actions dummy[t:Time] in the above process description. In fact, we would not need these actions if we could write

```
TC[nextGet1:Time][nextProv1:Time]
    [nextGet2:Time][nextProv2:Time] =
    //enterGet
    (when (nextProv1-Step1<=t \& t<nextProv1 \&
           nextProv2-Step2<=t \& t<nextProv2)
           [Models].enterGet[t:Time] ->
           TC[nextGet1][nextProv1][nextGet2][nextProv2]
```

...

This would make perfect sense expressing that for any $m \in$ Models *and* for any $t \in$ Time the action [m].enterGet[t] can only happen if the guard is satisfied for t. Unfortunately FSP does not support this possibility since the index variable t is considered to be undefined in the guard. However, if we first introduce the (non-sense) actions dummy[t:Time] then the index variable t is known where necessary. The dummy actions are made invisible by applying the hiding operator.

As an example, the design model of a distributed simulation with two simulation models having time steps 2 and 3 resp. is given by the following composite process.

```
const StepModel1 = 2
const StepModel2 = 3
||SYS =
    ([1]:MODEL(StepModel1)||[2]:MODEL(StepModel2)||
     TIMECONTROLLER(StepModel1, StepModel2))
     /{start/[Models].start}.
```

We cannot visualise the labelled transition system of the process SYS because it has too many states and transitions. However, for an analysis of the behaviour of the design model we can consider different views on the system which can be formally defined by means of the interface operator. For instance, if we want to focus only on the get and prov actions executed by the system we can build the process SYS@{[Models].GetProvs} where the set GetProvs has been defined in Section 7.1.2. The corresponding LTS, after minimisation w.r.t. invisible actions, is shown in Figure 7.6.

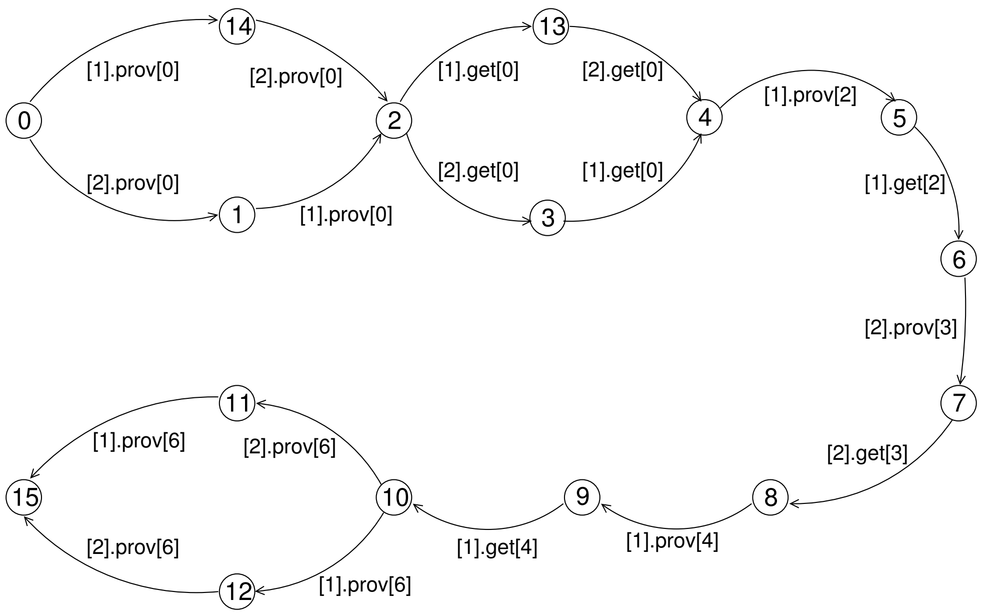


Figure 7.6: View on the get/prov actions

Similarly one could focus on the mutual exclusion behaviour of the system by exhibiting only the enter and exit actions, i.e. by considering the process SYS@{[Models].EnterExits}. The (minimised) LTS of this view is depicted in Figure 7.7. As the LTS grows rapidly with the number of actions contained, only the actions nterProv[0], exitProv[0][Models].e are considered.

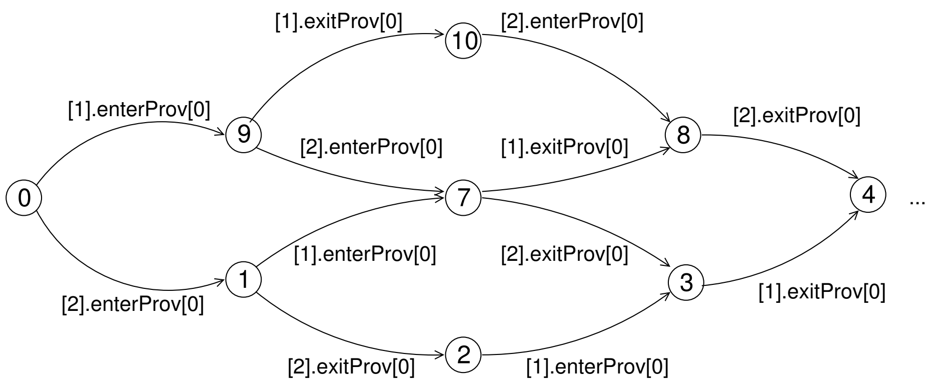


Figure 7.7: View on the enter/exit actions

Checking the Safety Properties

In order to check that the design model indeed satisfies the required safety properties we apply standard model checking techniques. For this purpose we construct for each property process the parallel composition with the design model. If in the resulting LTS the error state is not reachable then the safety property is fulfilled, otherwise it is violated. For instance, if the two simulation models from above are involved in an integrative simulation we construct the following processes.

```
||CHECK_VALIDDATA_USER1_PROV2 =
        (SYS||VALIDDATA(1,StepModel1,2,StepModel2)).
||CHECK_VALIDDATA_USER2_PROV1 =
        (SYS||VALIDDATA(2,StepModel2,1,StepModel1)).
||CHECK_EXCLUSION = (SYS||EXCLUSION).
```

The analysis with the LTSA tool shows that no errors occur, i.e. the design model satisfies the coordination requirements for the validity of data and for get/provide exclusion. For more complex configurations more efficient model checkers like SPIN [Hol04] should be used. Several runs with SPIN have shown that the efficiency of model checking the design of the timecontroller depends strongly on the distribution of the individual model steps whereby it is beneficial if their greatest common divisor is as small as possible. Otherwise one may run out of memory and therefore appropriate abstraction techniques have still to be investigated.

Checking the Liveness Properties

In section 7.1.2 we have stated a liveness property which requires that each simulation model provides data during the whole simulation period at any time that fits to its local time step. To check this condition with LTSA we can define a collection of progress properties of the form

```
progress PROV_Model_m_t = {[m].prov[t]}
```

for each $m \in Models$ and $t \in Time$ with $t\%Step_m = 0$. With this approach, however, two difficulties arise. First, we obtain quite a lot of progress properties to be considered and, more seriously, none of the properties will be fulfilled because simulations are finite but progress properties assume infinite execution traces.

The first difficulty can be easily solved by using indexed progress properties. In our case we define for each model a family of progress properties indexed by the time for which the model should provide data. This means that for each $m \in Models$ we obtain an (indexed) progress property of the following form:

```
progress PROV_Modelm[i:0..(SimEnd-SimStart)/StepModelm] =
  {[m].prov[SimStart + i * StepModelm]}
```

To overcome the second problem the idea is to introduce artificial cycles such that after a simulation is finished it is automatically restarted. We will not further detail here the necessary, straightforward modifications of the processes occurring in the design model. It should be obvious that for checking the required liveness property for integrative simulations it is now (necessary and) sufficient to check that the modified design model satisfies all progress properties from above. Indeed a progress analysis with LTSA shows that no progress property is violated. Thus, in summary, we have shown that the timecontroller-based design model is a correct solution of the coordination problem. The corresponding LTS is shown in Figure 7.8. Also the timecontroller must be adapted accordingly which is not detailed here.

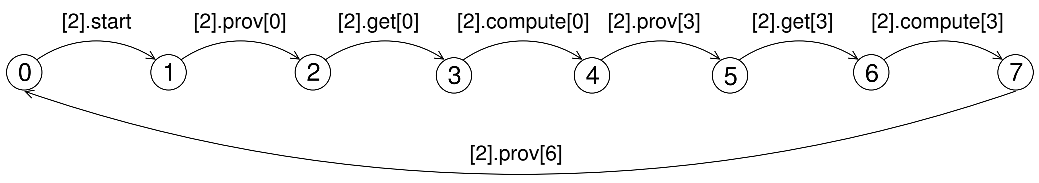


Figure 7.8: Model with cycles

Again we can visualise certain views on the behaviour of the modified system. For instance, if we focus on the actions get and prov we obtain the (minimised) LTS shown in figure 7.9.

For checking the required liveness property for integrative simulations it is now (necessary and) sufficient to check that the modified design model satisfies all progress properties from above. Indeed a progress analysis with LTSA shows that no progress property is violated. Thus, in summary, we have shown that the timecontroller-based design model is a correct solution of the coordination problem.

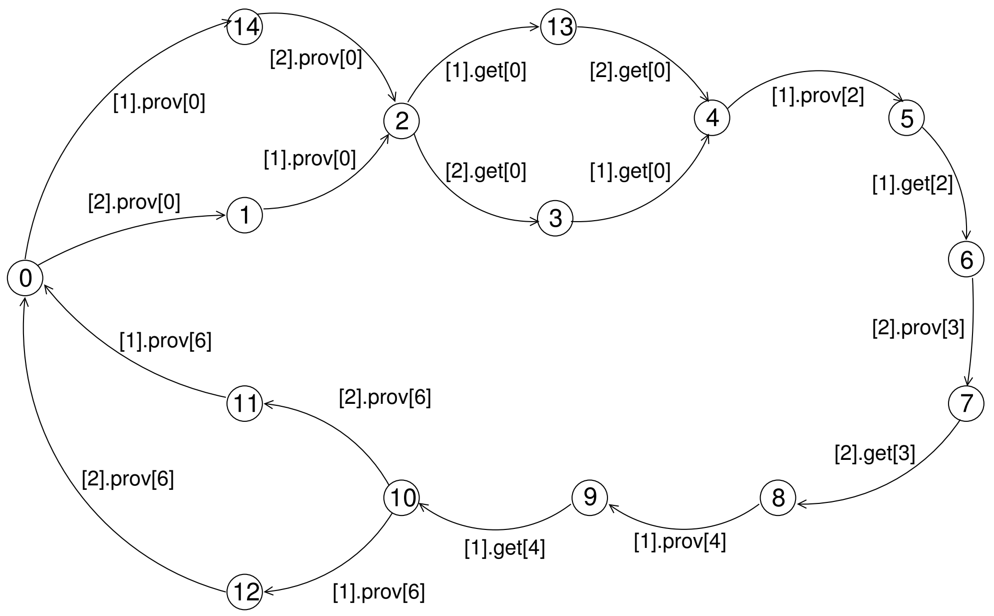


Figure 7.9: The minimised LTS focussing on get/prov actions with cycles

7.2.2 Structural UML Design Model

In this section we sketch how we derive in a systematic way a UML design model from the timecontroller-based FSP model. In principle, many steps of the translation procedure, which follows the pragmatic ideas of Magee and Kramer provided in [MK06][1], could be automated if the FSP model would be enhanced by additional information, saying, for instance, which processes are considered as active or passive objects and which actions are considered as input or output actions.

An overview of the structural UML design model is given in the class diagram in Figure 7.10. The details of the classes can be found in Appendix A.4.1. In the following we will briefly describe the classes and their relation to the formal FSP design model.

The class Timecontroller (cf. Figure 7.11) which is newly introduced in the view "time coordination" arises from the FSP process TIMECONTROLLER. It provides the operations enterGet, exitGet, enterProv and exitProv which correspond to the respective actions of the TIMECONTROLLER process. Furthermore there are qualified associations to the class Date with role names nextProv and nextGet respectively which store the information about the current progress of the models involved in the simulation. The class Date represents a time point which has been modelled by a natural number in the FSP model. The associations correspond to the indices of the local processes TC. Note that one Timecontroller instance controls arbitrarily many ModelCore instances which is denoted by the multiplicities 1 and * at the respective ends of the (directed) association.

[1]Magee and Kramer actually provide a translation scheme from FSP to Java which we adapt

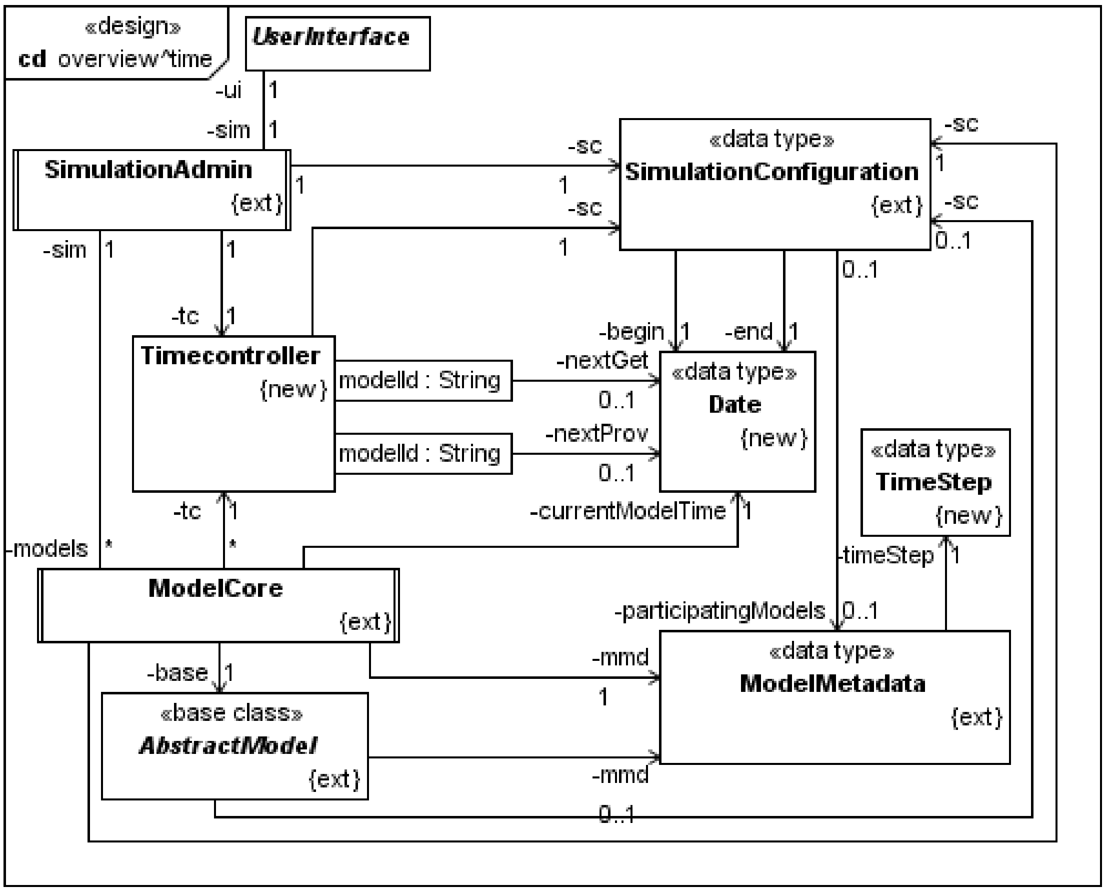


Figure 7.10: Structural UML design model in the view “time coordination” (overview)

As the process MODEL runs through the model’s life cycle, it is represented by the active class ModelCore (cf. Figure 7.12). All actions of the MODEL process are considered as output actions which correspond to operation calls on a Timecontroller object on the one hand (enterGet, exitGet, enterProv, exitProv) and on a concrete simulation model object on the other hand (getData, compute, provide) which is represented on the framework level by the base class AbstractModel (cf. Figure 7.13). The aforementioned operations are designed as mandatory plug-points, i.e. as abstract operations which have to be implemented in a concrete simulation model. The course of action is implemented in the start operation of the class ModelCore. We will detail on this operation in the following section.

All relevant information which has been coded as global constants, process parameters or process labels in the FSP model (like, e.g. simulation start and end, model identifiers, etc.) we find in the UML design model as properties of the classes SimulationConfiguration and ModelMetadata respectively (cf. Figure 7.14 for details of SimulationConfiguration; for details of the remaining classes please refer to Appendix A.4.1). In contrast to the FSP model, where

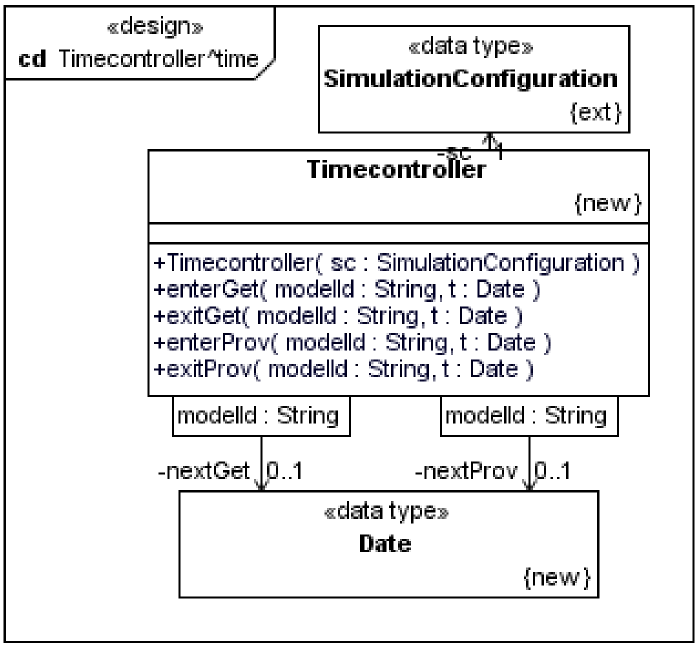

«design»
cd Timecontroller^time
«data type»
SimulationConfiguration
{ext}
-sc
1
Timecontroller
{new}
+Timecontroller(sc : SimulationConfiguration)
+enterGet(modelId : String, t : Date)
+exitGet(modelId : String, t : Date)
+enterProv(modelId : String, t : Date)
+exitProv(modelId : String, t : Date)
modelId : String
modelId : String
-nextGet
0..1
-nextProv
0..1
«data type»
Date
{new}

Figure 7.11: Class Timecontroller in the view "time coordination"

data only may be coded by natural numbers, we use more sophisticated types here. Therefore, simulation begin and end (like time points in general) are modelled by the class Date which provides query operations to decide whether a Date instance is before or after another, and additionally, a query to obtain the consecutive time point with respect to a certain time step (getNextDate).

The time step of a simulation model is represented by the class TimeStep which comprises an integer attribute value as well as an enumeration property unit of type TimeStepUnit. With this construct it is possible to express periodic time steps (like, e.g., one minute, six hours, one day), as well as calendar-based non-periodic time steps (like, e.g., one month, one year) which are mainly important in social sciences. Finally note, that the composite process SYS may be viewed as represented by the class SimulationAdmin.

Finally we have to consider the query isValid of the class SimulationConfiguration the result of which decides whether a simulation configuration is valid (and hence the corresponding simulation is executable) or not. Concerning the temporal aspects of a simulation configuration one has to ensure that

- simulation begin and simulation end are defined and begin is before end;
- each participating model comprises a valid time step, i.e. a time step value greater than zero and a well-defined time step unit.

The conditions stated above are expressed by the following OCL conditions (note that the

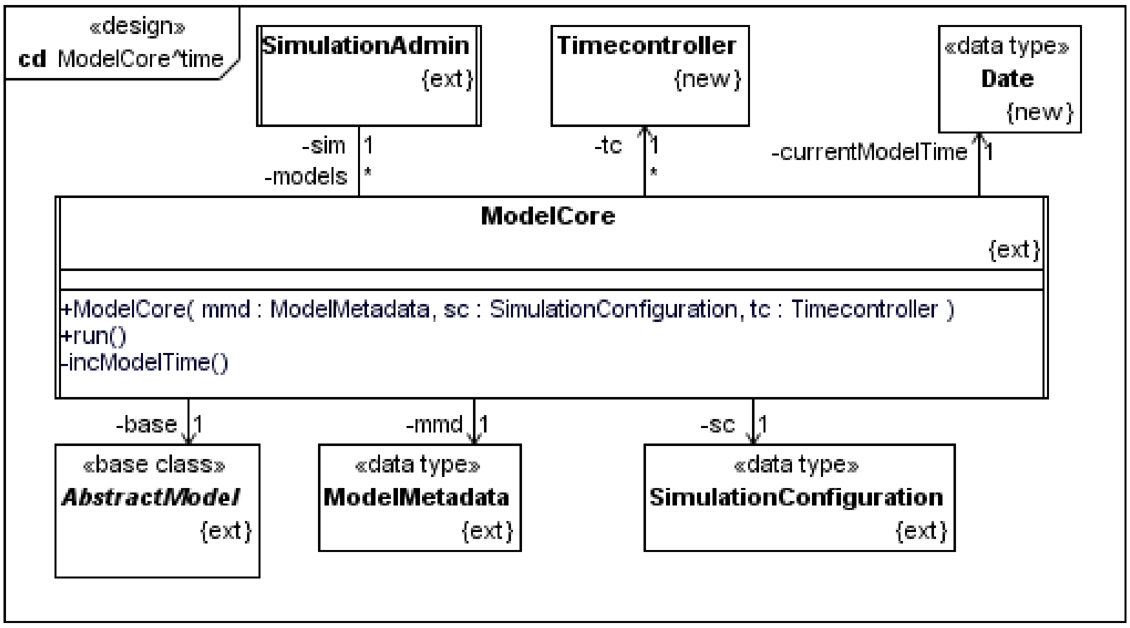


Figure 7.12: Class ModelCore in the view "time coordination"

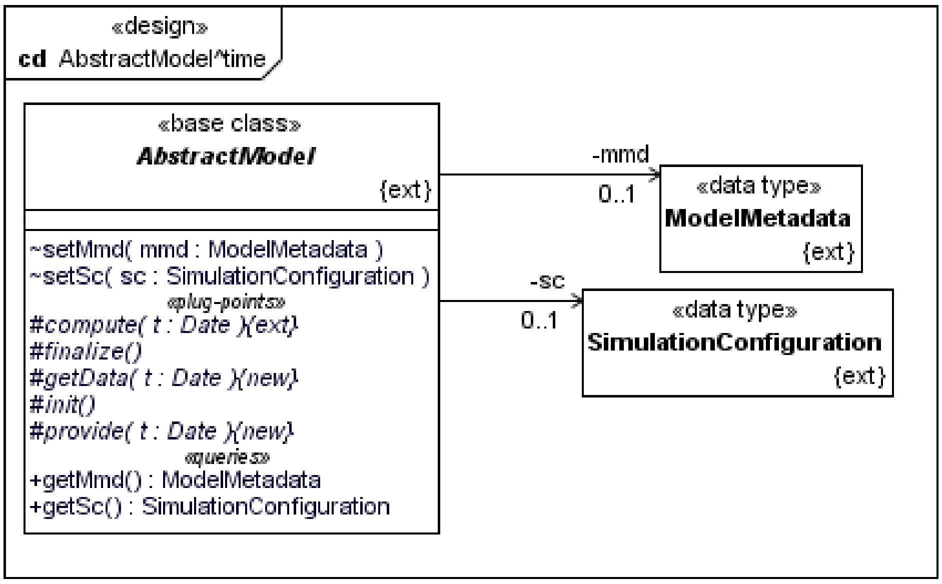

«design»
cd AbstractModel^time
«base class»
AbstractModel
{ext}
~setMmd(mmd : ModelMetadata)
~setSc(sc : SimulationConfiguration)
«plug-points»
#compute(t : Date){ext}
#finalize()
#getData(t : Date){new}
#init()
#provide(t : Date){new}
«queries»
+getMmd() : ModelMetadata
+getSc() : SimulationConfiguration
-mmd
0..1
«data type»
ModelMetadata
{ext}
-sc
0..1
«data type»
SimulationConfiguration
{ext}

Figure 7.13: Class AbstractModel in the view "time coordination"

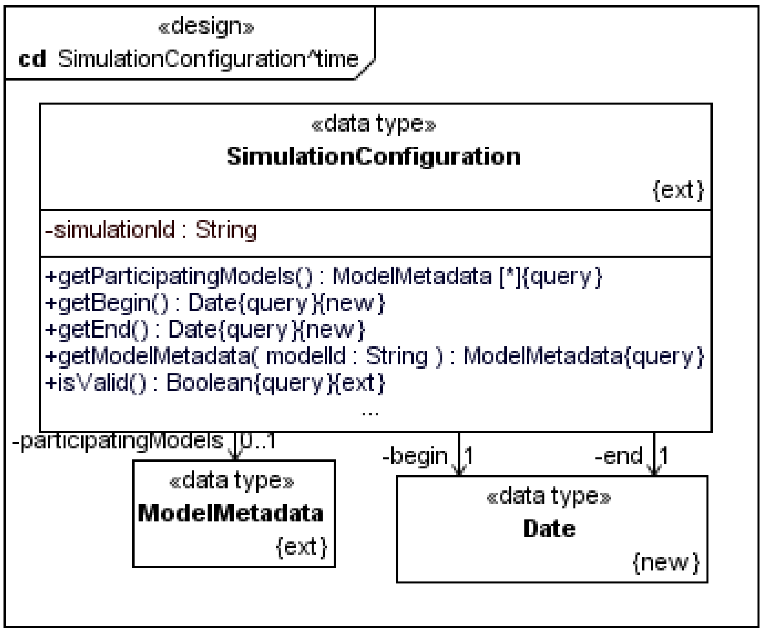

«design»
cd SimulationConfiguration^time
«data type»
SimulationConfiguration
{ext}
-simulationId : String
+getParticipatingModels() : ModelMetadata [*]{query}
+getBegin() : Date{query}{new}
+getEnd() : Date{query}{new}
+getModelMetadata(modelId : String) : ModelMetadata{query}
+isValid() : Boolean{query}{ext}
...
-participatingModels 0..1
-begin 1
-end 1
«data type»
ModelMetadata
{ext}
«data type»
Date
{new}

Figure 7.14: Class SimulationConfiguration in the view “time coordination”

existence of a begin and an end date is implicitly ensured by the multiplicity 1 at the respective association ends).

```
context SimulationConfiguration::isValid()
 pre:  true
 post: result = self.simulationId <> ""
   and self.participatingModels <> null
   and self.participating.Models->forAll(m | m.isValid())
   and self.begin.isBefore(self.end)

context ModelMetadata::isValid()
 pre:  true
 post: result = self.modelId <> "" and self.modelClass <> ""
   and self.timeStep.isValid()

context TimeStep::isValid()
 pre:  true
 post: result = self.value > 0
```

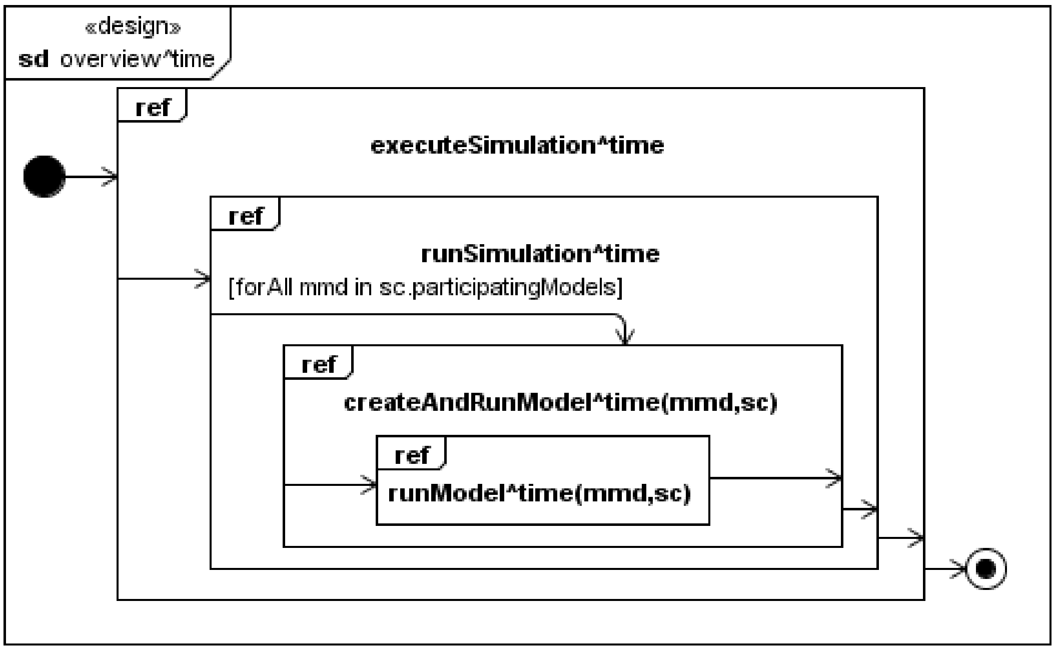


Figure 7.15: Interaction overview diagram of the view “time coordination”

7.2.3 Behavioural UML Design Model

Let us now consider the behavioural part of the UML design model an overview of which is given by the interaction overview diagram in Figure 7.15. At first sight this diagram shows no extension with respect to the interaction overview diagram of the base view, so we have to take a closer look at the single interaction uses contained in the overview. The details of each diagram can be found in Appendix A.4.2. In the following we will concentrate on the diagram runModel^time which comprises the most interesting extension with respect to the base view.

So let us consider the sequence diagram runModel^time (cf. Figure 7.16). All interactions between the operation calls init and finalize (which are already part of the base diagram) are derived from the (extended) FSP process MODEL.

In the following we go through the diagram along the lifeline of the ModelCore object. After the init operation call has returned the current model time has to be set to the actual simulation begin time which is specified by the respective state invariant. The state invariant is followed by the message call enterProv on the Timecontroller instance. As this message call corresponds to the action enterProv which is shared by the processes TIMECONTROLLER and MODEL the synchronisation of these processes has to be taken into account here. Remember that the action enterProv in the process TIMECONTROLLER is guarded by the condition nextGet1>=t & nextGet2>=t which stands exemplary for the condition nextGet1>=t & nextGet2>=t & ... & nextGetn>=t when considering *n* participating models. This condition can be translated into the following OCL statement by using the

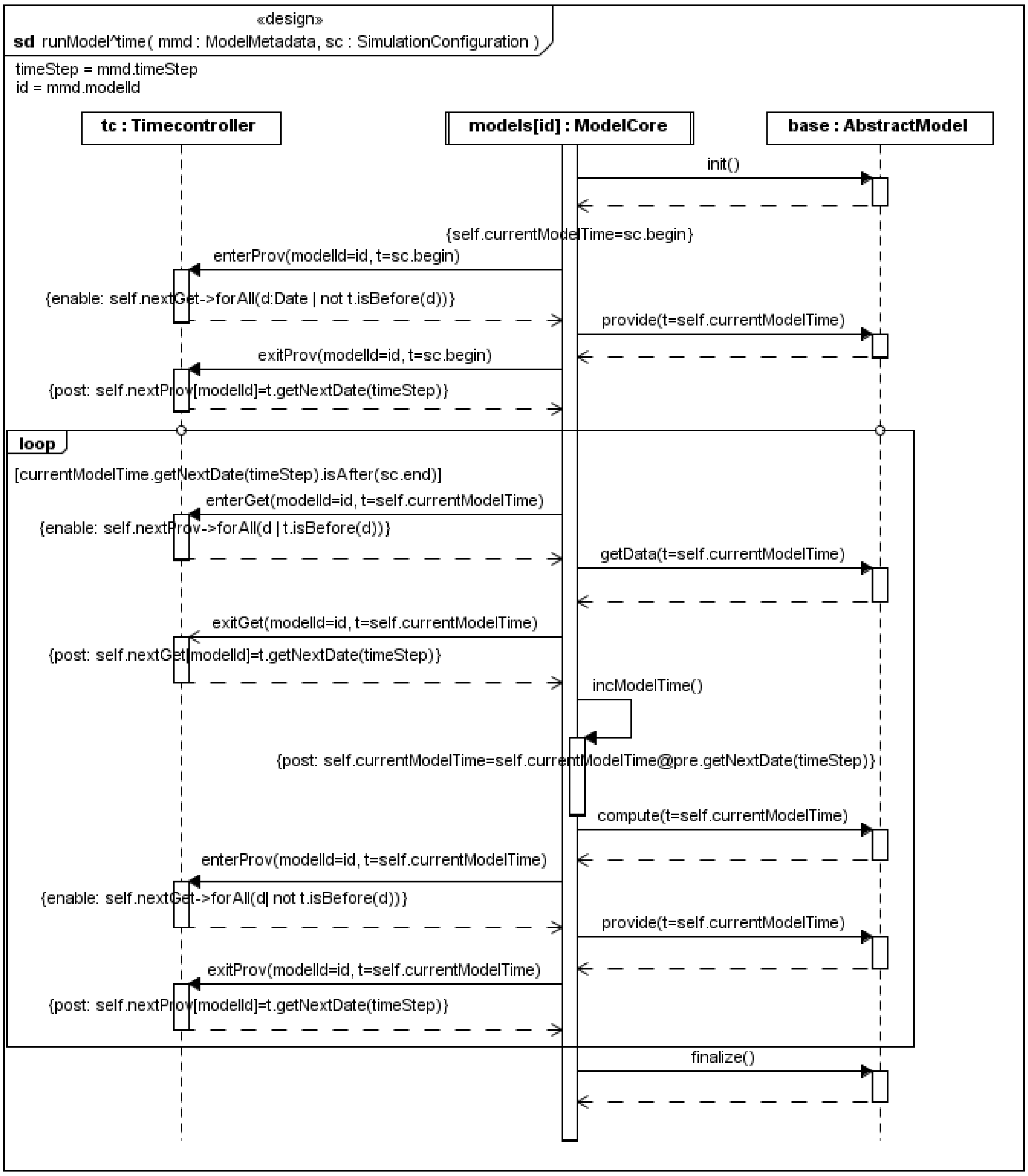

«design»
sd runModel^time(mmd : ModelMetadata, sc : SimulationConfiguration)
timeStep = mmd.timeStep
id = mmd.modelId
tc : Timecontroller
models[id] : ModelCore
base : AbstractModel
init()
{self.currentModelTime=sc.begin}
enterProv(modelId=id, t=sc.begin)
{enable: self.nextGet->forAll(d:Date | not t.isBefore(d))}
provide(t=self.currentModelTime)
exitProv(modelId=id, t=sc.begin)
{post: self.nextProv[modelId]=t.getNextDate(timeStep)}
loop
[currentModelTime.getNextDate(timeStep).isAfter(sc.end)]
enterGet(modelId=id, t=self.currentModelTime)
{enable: self.nextProv->forAll(d | t.isBefore(d))}
getData(t=self.currentModelTime)
exitGet(modelId=id, t=self.currentModelTime)
{post: self.nextGet[modelId]=t.getNextDate(timeStep)}
incModelTime()
{post: self.currentModelTime=self.currentModelTime@pre.getNextDate(timeStep)}
compute(t=self.currentModelTime)
enterProv(modelId=id, t=self.currentModelTime)
{enable: self.nextGet->forAll(d| not t.isBefore(d))}
provide(t=self.currentModelTime)
exitProv(modelId=id, t=self.currentModelTime)
{post: self.nextProv[modelId]=t.getNextDate(timeStep)}
finalize()

Figure 7.16: Sequence diagram runModel^time

types of the UML class design model:

```
context Timecontroller::enterProv(modelId:String, t:Date)
  enable: self.nextGet.forAll(d : Date | not t.isBefore(d))
```

Note that we use the keyword to express a condition that blocks the caller of the respective operation until the conditions becomes true (i.e. the condition has to be checked every time the state of the affected objects changes). In Section 10.2.2 we provide an implementation pattern for the enable condition in Java. After the enterProvide message returned, the provide message is called on AbstractModel, i.e. on a concrete simulation model instance at run time. When the provide operation has finished, it is signalled to the Timecontroller by calling exitProv.

After the initial providing of data the execution loop is entered which corresponds to the conditional recursive call of the local FSP process M. The interactions within the loop are performed until the exit condition denoted in square brackets remains false, i.e. the simulation end is not reached. Within the loop at first the interaction pattern from above is repeated with the enterGet, getData and exitGet message respectively. After that the current model time is increased by the actual time step (cf. the message call inc ime to self and the corresponding postcondition), before the computation of new data is activated by calling compute on AbstractModel. At the end of the loop we find again the interaction pattern for the provide operation and its corresponding enter and exit operations.

7.3 Component Design

The component diagram in Figure 7.17 shows the architecture of the view “time coordination”. The architecture of this view extends the base architecture by the component TimeCoordination and its provided interfaces TimecoordinationAccess and TimecontrollerMonitor. While the first one is intended to configure the component by setting the appropriate simulation configuration (via the operation setSimulationConfiguration), the latter interface is relevant for the coordination of the simulation models. It comprises the operations enterGet, exitGet, enterProv, and exitProv, which have been described in the previous section.

The details of the components Simulation, Model and TimeCoordination in the current view can be found in Appendix A.4.3.

7.4 Conformance with the Development Methodology

In this chapter the architecture of the view “Time Coordination of Integrative Simulations“ (or time view for short) has been constructed which corresponds to the following extension and refinement steps according to our development methodology.

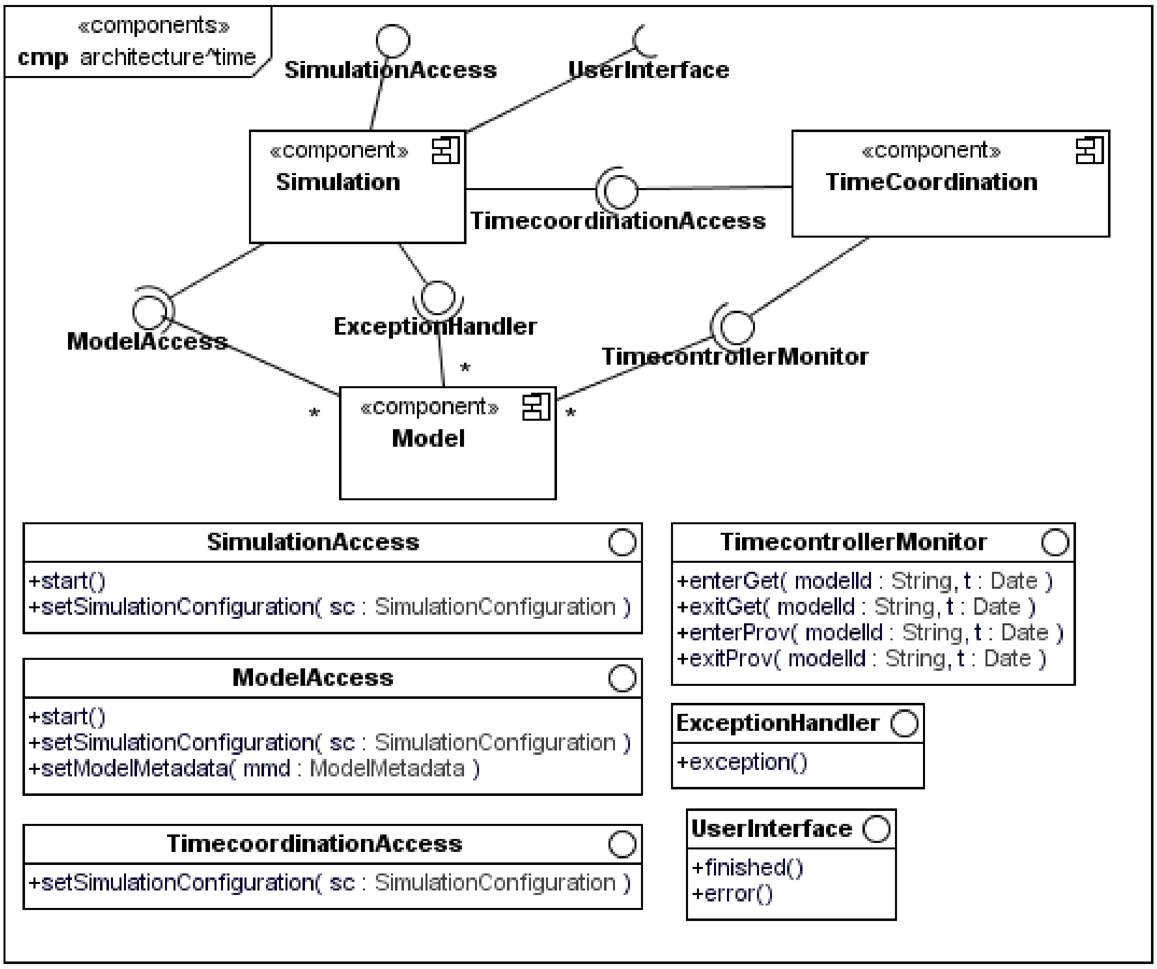

«components»
cmp architecture^time
SimulationAccess
UserInterface
«component»
Simulation
TimecoordinationAccess
«component»
TimeCoordination
ModelAccess
ExceptionHandler
TimecontrollerMonitor
«component»
Model
SimulationAccess
+start()
+setSimulationConfiguration(sc : SimulationConfiguration)
ModelAccess
+start()
+setSimulationConfiguration(sc : SimulationConfiguration)
+setModelMetadata(mmd : ModelMetadata)
TimecoordinationAccess
+setSimulationConfiguration(sc : SimulationConfiguration)
TimecontrollerMonitor
+enterGet(modelId : String, t : Date)
+exitGet(modelId : String, t : Date)
+enterProv(modelId : String, t : Date)
+exitProv(modelId : String, t : Date)
ExceptionHandler
+exception()
UserInterface
+finished()
+error()

Figure 7.17: Architecture in the view “time coordination”

$$\begin{tikzcd}
\mathrm{req}^{base} \arrow[r, rightsquigarrow] \arrow[d, hookrightarrow] & \mathrm{des}^{base} \arrow[r, rightsquigarrow] \arrow[d, hookrightarrow] & \mathrm{cmp}^{base} \arrow[d, hookrightarrow] \\
\mathrm{req}^{time} \arrow[r, rightsquigarrow] & \mathrm{des}^{time} \arrow[r, rightsquigarrow] & \mathrm{cmp}^{time}
\end{tikzcd}$$

As the refinement of the base view has already been shown in Section 4.5, one has to prove that

1. the abstraction levels of the time view are in refinement relation, and
2. each abstraction level of the time view is an extension of the respective level of the base view.

Again we leave the proof of both cases to the reader as they can be produced analogously to those provided in Sections 4.5 and 5.4 respectively.

7.5 Discussion

Our approach for time coordination of an integrative simulation allows for the parallel execution of simulation models and guarantees that each model gains the most recent data when importing from another model during the simulation. Of course, when comparing our approach with those inherent to other frameworks we have to face the following disadvantages.

- There might be a time lag of at most one time step between the imported data and the current model time of the importing model (although the data is most recent). This can be attenuated by increasing the frequency of the affected models, i.e. by reducing the respective time steps.

- When a model with a smaller time step is connected to a model with a greater one, then redundant data is fetched by the model with the smaller time step until the other model changes its state again. This is, however, no essential problem but rather an issue of performance and could be eliminated with appropriate mechanisms. For example, an observer mechanism could be used here: the model with the smaller time step (e.g. 1 hour) registers itself as an observer at the model with the greater time step (e.g. 1 day); each time the latter model provides new data it notifies its observers. For the observing models it is then sufficient to fetch the data immediately after notification (i.e. once per day when considering the above mentioned time steps).

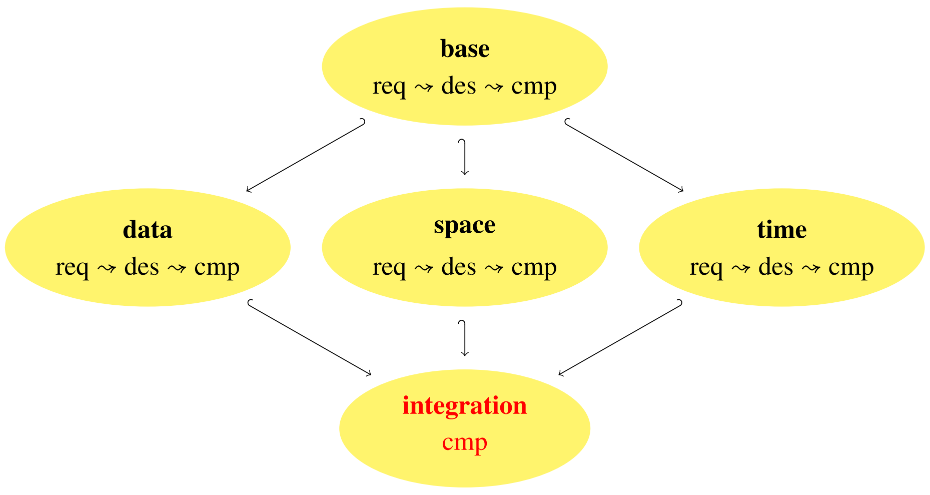
However, our approach is – to our knowledge – the only one which enables parallel execution and hence, in combination with network distribution, the creation of a parallel and distributed simulation system.

The OpenMI approach (cf. [GGW07]) of time coordination is based on a request and reply mechanism where a chain of models is created and the first model in the chain, the so called *trigger*, is activated and then calls an operation getValues on the next model of the chain, and so on. This operation is sequentially called until the simulation end is reached. Note that the getValues call causes the enquired model to perform a computation for the respective time step. In the case that a model contains a cycle this would lead to a deadlock. To avoid this deadlock each model remembers if its getValues method has been called within the current time step (or if it initiated the getValues call chain itself) and in this case returns a so called ”best guess“ of values (e.g. an extrapolation of computed values of previous time steps) instead of computing new values.

The OMS (cf. [KKO05]) introduces the notion of so called Time Compound Components (TCC), which define a certain time step. Within a TCC all models having the respective time step are sequentially called to run over a time step.

8

Integration of the Views: The Framework Architecture



In the preceding chapters we developed an architecture for different views of the Generic Simulation Framework under consideration. Now we are able to complete the figure above and construct the complete framework architecture by the integration of the models of the different views according to the methodology provided in Chapter 3. As we have seen there, the integration process could be performed automatically, and leads to a unique result except for renaming.

In the remainder of this chapter we first present the result of the structural integration (Section 8.1), i.e. the integrated component architecture, and then exhibit the result of the behavioural integration (Section 8.2), i.e. the integrated sequence diagrams.

8.1 Structural Integration

The result of the integration of the component architectures from the different views is shown in Figure 8.1. The complete architecture of the framework consists of five components: BaseData, Simulation, TimeCoordination, ModelLinking and Model, the latter comprising a subcomponent Proxel. Note that Model is the only component exhibiting multiplicity *, which means that multiple instances of this component exist at run time. This makes sense as the component Model represents a generic simulation model – at run time a concrete simulation model has to be bound to that component in order to integrate this model into the simulation system – and we want to integrate an arbitrary number of simulation models within the system. All other components are singletons as they act as controlling instances for one integrative simulation.

After presenting the architecture we have to detail on the single components. Fortunately most of the components have already been fully specified within the single views:

- the component ModelLinking within the view “data exchange”,
- the components BaseData and Proxel within the view “simulation space”, and
- the component TimeCoordination within the view “time coordination”.

So we still have to consider the details of the components Simulation and Model with its subcomponent Proxel. For lack of space we restrict ourselves to showing the component Model here in Figure 8.2 and refer for component Simulation to Appendix A.5.1.

Exemplarily we also want to show the integrated package metadata (cf. Figure 8.3) which contains the class SimulationConfiguration and the types of its properties, like, e.g. ModelMetadata, Date, AreaMetadata, etc.

The integrated postcondition of the query isValid of the class SimulationConfiguration results from the conjunction over the respective postconditions in the single views and reads as follows:

```
context SimulationConfiguration::isValid()
 pre:  true
 post: result = self.simulationId <> ""
   and self.participatingModels <> null
   and self.participating.Models->forAll(m | m.isValid())
   and self.participatingModels.forAll(m |
     m.importInterfaces->forAll(i |
      self.participatingModels->one(n |
        n.exportInterfaces->includes(i))))
   and self.area.isValid()
   and self.resources->forAll(r | r.isValid())
   and self.area.properties->forAll(p |
      self.resources->exists(r | r.property = p))
   and self.begin.isBefore(self.end)
```

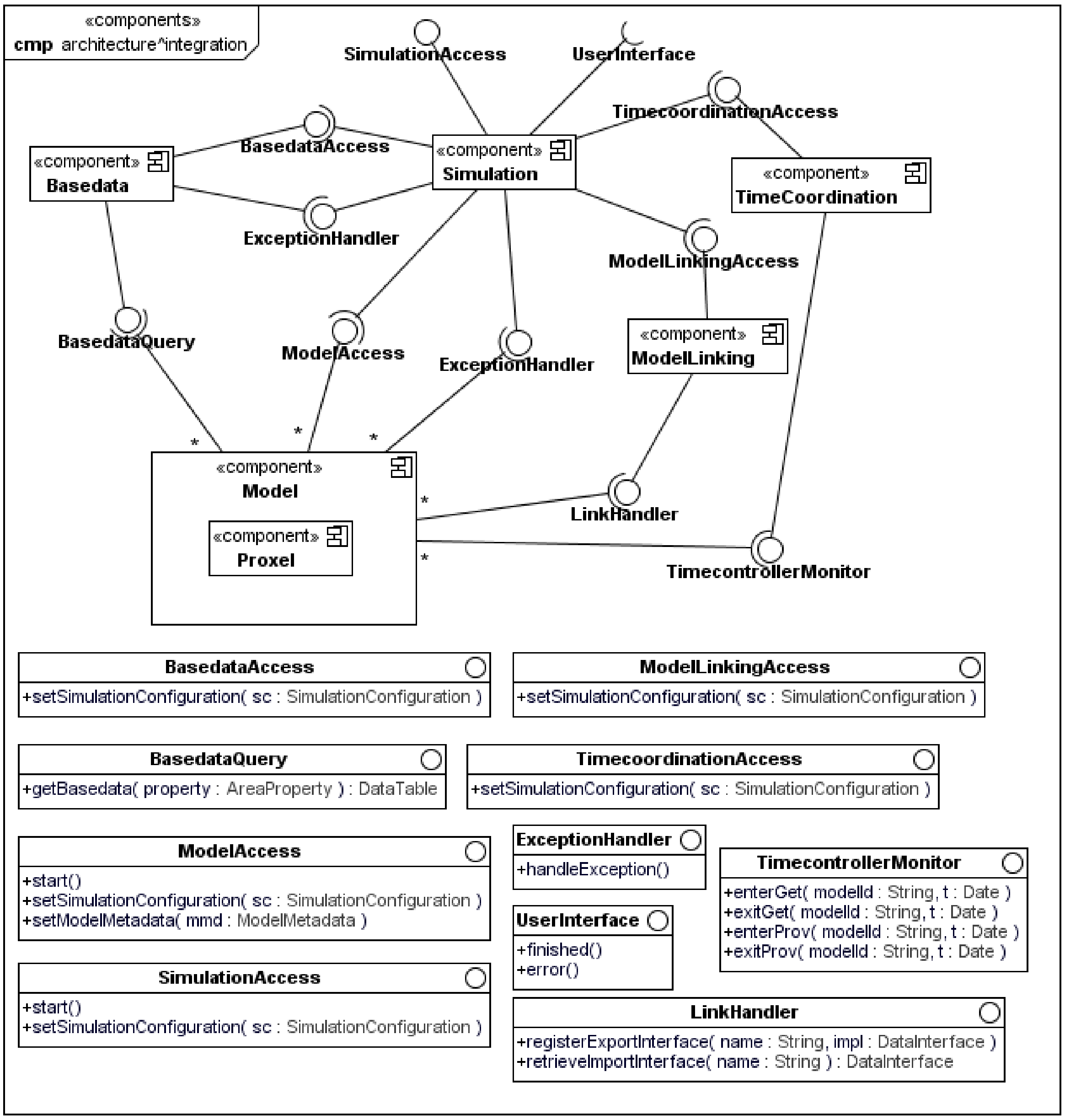

«components»
cmp architecture^integration
SimulationAccess
UserInterface
TimecoordinationAccess
BasedataAccess
«component» Basedata
«component» Simulation
«component» TimeCoordination
ExceptionHandler
ModelLinkingAccess
BasedataQuery
ModelAccess
ExceptionHandler
«component» ModelLinking
*
*
*
«component» Model
«component» Proxel
*
LinkHandler
*
TimecontrollerMonitor
BasedataAccess
+setSimulationConfiguration(sc : SimulationConfiguration)
ModelLinkingAccess
+setSimulationConfiguration(sc : SimulationConfiguration)
BasedataQuery
+getBasedata(property : AreaProperty) : DataTable
TimecoordinationAccess
+setSimulationConfiguration(sc : SimulationConfiguration)
ModelAccess
+start()
+setSimulationConfiguration(sc : SimulationConfiguration)
+setModelMetadata(mmd : ModelMetadata)
ExceptionHandler
+handleException()
TimecontrollerMonitor
+enterGet(modelId : String, t : Date)
+exitGet(modelId : String, t : Date)
+enterProv(modelId : String, t : Date)
+exitProv(modelId : String, t : Date)
UserInterface
+finished()
+error()
SimulationAccess
+start()
+setSimulationConfiguration(sc : SimulationConfiguration)
LinkHandler
+registerExportInterface(name : String, impl : DataInterface)
+retrieveImportInterface(name : String) : DataInterface

Figure 8.1: Architecture of the simulation framework

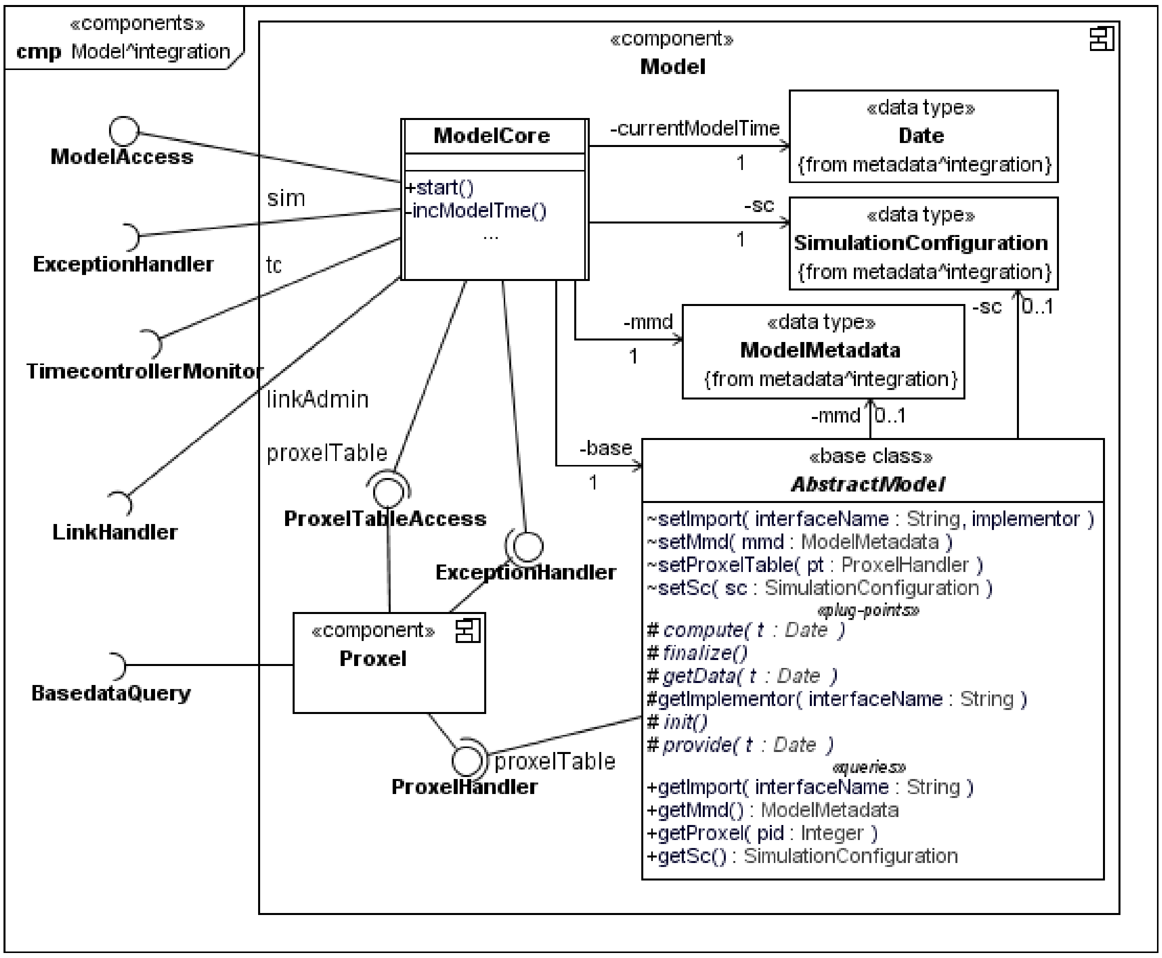


Figure 8.2: Integrated component Model

The postconditions of the isValid operations which are used within the above postcondition are listed below.

```
context ModelMetadata::isValid()
 pre:  true
 post: result = self.modelId <> ""
   and self.modelClass <> ""
   and self.exportInterfaces <> null
   and self.importInterfaces <> null
   and self.proxelClass <> ""
   and self.timeStep.isValid()

context AreaMetadata::isValid()
```

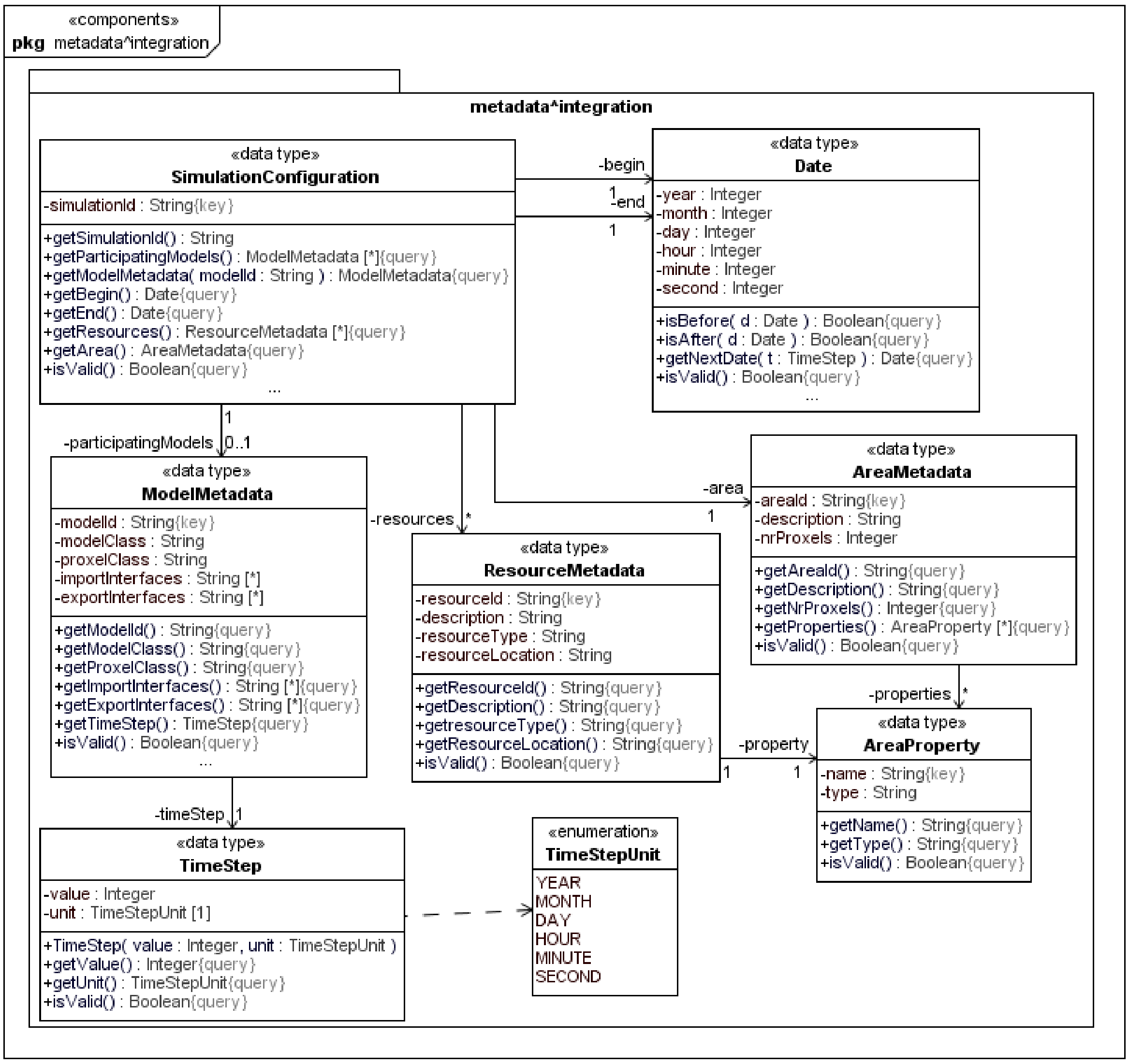

«components»
pkg metadata^integration
metadata^integration
«data type»
SimulationConfiguration
-simulationId : String{key}
+getSimulationId() : String
+getParticipatingModels() : ModelMetadata [*]{query}
+getModelMetadata(modelId : String) : ModelMetadata{query}
+getBegin() : Date{query}
+getEnd() : Date{query}
+getResources() : ResourceMetadata [*]{query}
+getArea() : AreaMetadata{query}
+isValid() : Boolean{query}
...
-begin
1
-end
1
«data type»
Date
-year : Integer
-month : Integer
-day : Integer
-hour : Integer
-minute : Integer
-second : Integer
+isBefore(d : Date) : Boolean{query}
+isAfter(d : Date) : Boolean{query}
+getNextDate(t : TimeStep) : Date{query}
+isValid() : Boolean{query}
...
1
-participatingModels
0..1
«data type»
ModelMetadata
-modelId : String{key}
-modelClass : String
-proxelClass : String
-importInterfaces : String [*]
-exportInterfaces : String [*]
+getModelId() : String{query}
+getModelClass() : String{query}
+getProxelClass() : String{query}
+getImportInterfaces() : String [*]{query}
+getExportInterfaces() : String [*]{query}
+getTimeStep() : TimeStep{query}
+isValid() : Boolean{query}
...
-resources
*
-area
1
«data type»
AreaMetadata
-areaId : String{key}
-description : String
-nrProxels : Integer
+getAreaId() : String{query}
+getDescription() : String{query}
+getNrProxels() : Integer{query}
+getProperties() : AreaProperty [*]{query}
+isValid() : Boolean{query}
«data type»
ResourceMetadata
-resourceId : String{key}
-description : String
-resourceType : String
-resourceLocation : String
+getResourceId() : String{query}
+getDescription() : String{query}
+getresourceType() : String{query}
+getResourceLocation() : String{query}
+isValid() : Boolean{query}
-properties
*
-property
1
1
«data type»
AreaProperty
-name : String{key}
-type : String
+getName() : String{query}
+getType() : String{query}
+isValid() : Boolean{query}
-timeStep
1
«data type»
TimeStep
-value : Integer
-unit : TimeStepUnit [1]
+TimeStep(value : Integer, unit : TimeStepUnit)
+getValue() : Integer{query}
+getUnit() : TimeStepUnit{query}
+isValid() : Boolean{query}
«enumeration»
TimeStepUnit
YEAR
MONTH
DAY
HOUR
MINUTE
SECOND

Figure 8.3: Integrated package metadata

```
   pre:  true
   post: result = areaId <> ""
         and description <> ""
         and nrProxels > 0
         and properties <> null
         and properties->forAll(p | p.isValid())

  context ResourceMetadata::isValid()
   pre:  true
   post: result = resourceId<>""
     and description <> ""
     and resourceType <> ""
     and resourceLocation <> ""
     and property.isValid()

  context AreaProperty::isValid()
   pre:  true
   post: name <> "" and type <> ""

  context TimeStep::isValid()
   pre: true
   post: result = self.value > 0
```

8.2 Behavioural Integration

The interaction overview diagram in Figure 8.4 shows the complete course of action by means of integrated sequence diagrams. Note that the referenced sequence diagrams have been integrated from those of the different views (denoted by the postfix integration) if they have already been present in the base view. This holds for the diagrams executeSimulation, runSimulation, createAndRunModel, and runModel. Otherwise the diagrams from different views are referenced directly and need not be integrated. Note further that – according to the methodology – interaction fragments from different views are independent and hence may be executed in parallel, which is denoted by the fork and join bars before and after the diagrams initModel^data and initModel^space respectively. Although the diagram runModel^integration is not shown here (but of course in the Appendix), we want to remark that the integration of the diagrams from the different views in this case is equal to the diagram runModel^time, as in the other views the extension with respect to the base view is trivial (i.e. the diagrams are equal).

The integrated sequence diagram createAndRunModel^integration is depicted in Figure 8.5. The remaining sequence diagrams referenced in Figure 8.4 can be found in Appendix A.5.2. Note that the interaction uses initModel^data and initModel^space are put in different operands of a par fragment to reveal the independent execution of the contained interactions.

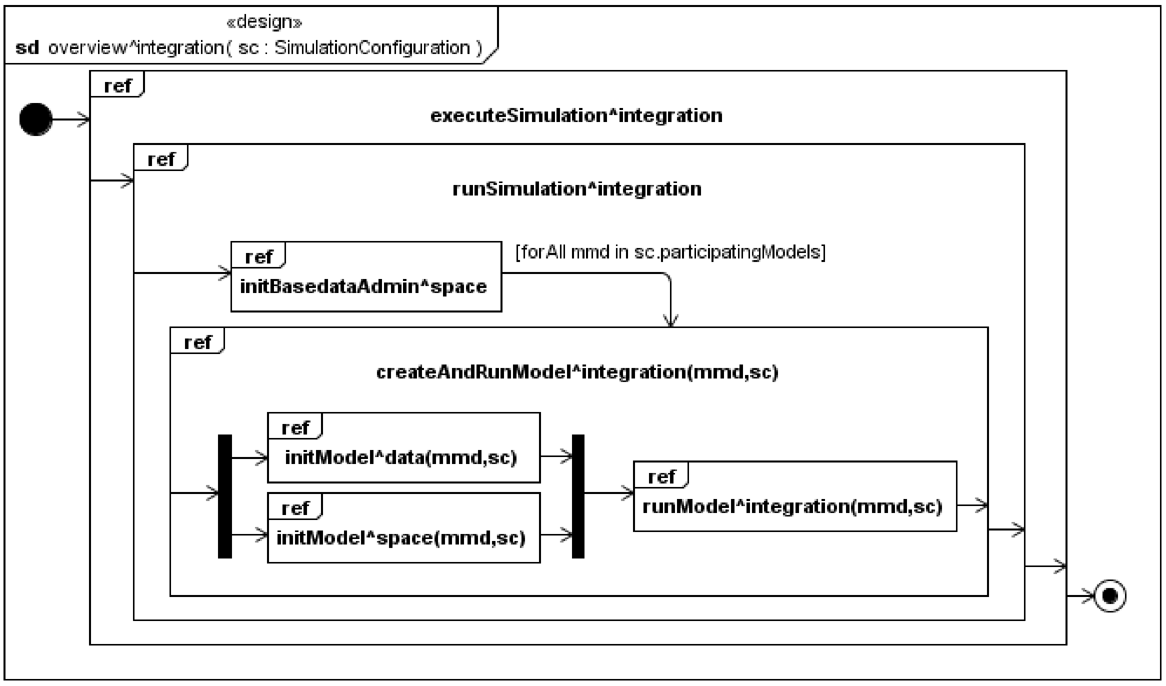

«design»
sd overview^integration(sc : SimulationConfiguration)
ref
executeSimulation^integration
ref
runSimulation^integration
ref
initBasedataAdmin^space
[forAll mmd in sc.participatingModels]
ref
createAndRunModel^integration(mmd,sc)
ref
initModel^data(mmd,sc)
ref
initModel^space(mmd,sc)
ref
runModel^integration(mmd,sc)

Figure 8.4: Interaction overview diagram of the integrated views

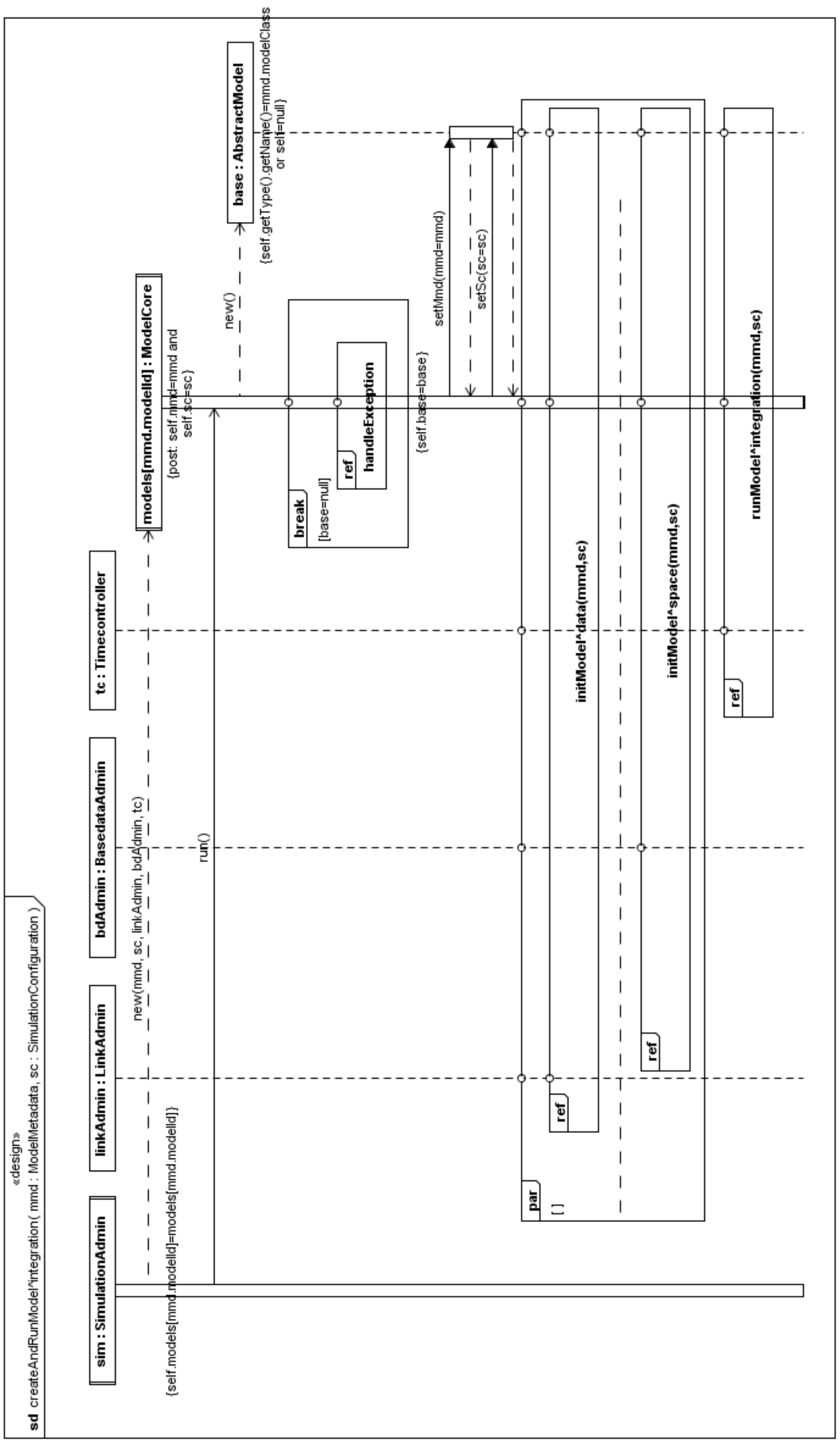

«design»
sd createAndRunModel^integration(mmd : ModelMetadata, sc : SimulationConfiguration)
sim : SimulationAdmin
linkAdmin : LinkAdmin
bdAdmin : BasedataAdmin
tc : Timecontroller
models[mmd.modelId] : ModelCore
base : AbstractModel
new(mmd, sc, linkAdmin, bdAdmin, tc)
{self.models[mmd.modelId]=models[mmd.modelId]}
{post: self.mmd=mmd and self.sc=sc}
run()
new()
{self.getType().getName()=mmd.modelClass or self=null}
break
[base=null]
ref
handleException
{self.base=base}
setMmd(mmd=mmd)
setSc(sc=sc)
par
[ ]
ref
initModel^data(mmd,sc)
ref
initModel^space(mmd,sc)
ref
runModel^integration(mmd,sc)

Figure 8.5: Sequence diagram createAndRunModel^integration

9

Some Guidelines for the Application of the Simulation Framework

This chapter provides some guidelines how the Generic Simulation Framework can be applied within a multidisciplinary environmental research project in order to obtain and apply an integrative simulation system. For this purpose a characteristic workflow for the application of the framework is presented in Section 9.1. Then, in the subsequent sections the most important activities within this workflow are detailed.

While the current chapter provides an abstract view on the application of the framework, in particular without taking into account a specific programming language, in Chapter 11, a concrete application of the framework within the multidisciplinary environmental research project GLOWA-Danube to set up the simulation and decision support system DANUBIA is discussed in detail.

9.1 Workflow of a Simulation Project

A *simulation project* is intended to examine a certain environmental problem or question, like e.g. "how will the expected frequency of the occurrence of extreme discharge at a gage P change within the next 100 years?" by means of simulation. A characteristic workflow for the application of the Generic Simulation Framework on a simulation project is given by the activity diagram in Figure 9.1.

Typically, in an integrative project a number of people from different scientific disciplines and from different locations work together and with fill different roles within the project (cf. [End03]). The most important roles with respect to the creation of a simulation system are *project management*, *model developer*, and *simulation administrator*. These roles can are assigned to responsibilities within the workflow which are denoted by the vertical partitions (or "swim lanes") in the activity diagram.

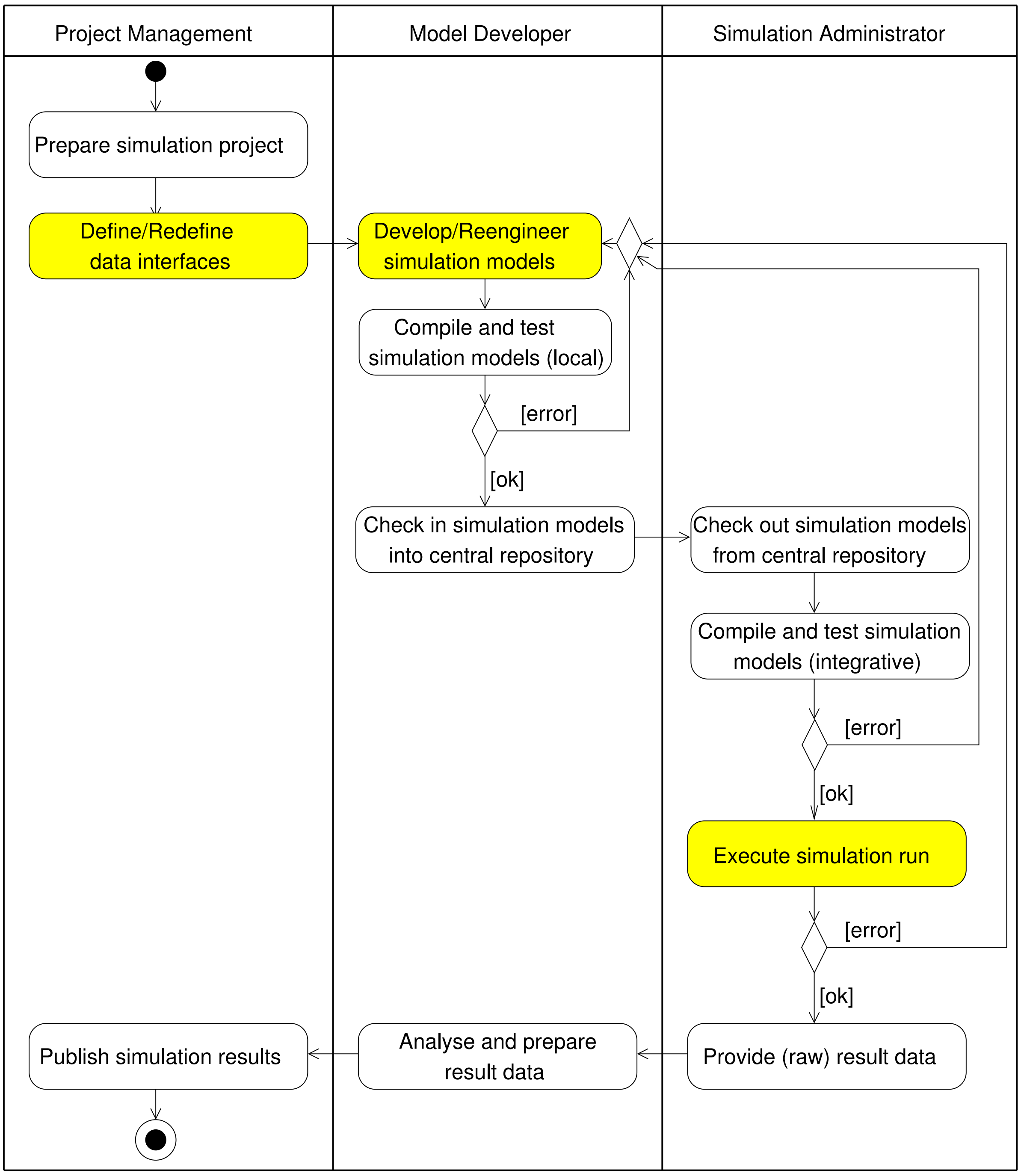

Project Management
Model Developer
Simulation Administrator
Prepare simulation project
Define/Redefine data interfaces
Develop/Reengineer simulation models
Compile and test simulation models (local)
[error]
[ok]
Check in simulation models into central repository
Check out simulation models from central repository
Compile and test simulation models (integrative)
[error]
[ok]
Execute simulation run
[error]
[ok]
Publish simulation results
Analyse and prepare result data
Provide (raw) result data

Figure 9.1: Workflow for the framework application (highlighted activities are detailed in subsequent sections)

The project management normally consists of a group of leaders of the participating disciplines or subprojects. In each subproject at least one person should act as model developer, and at least one person is necessary to fill the role of the central simulation administrator. Note that the activities in the middle partition of the diagram are supposed to be performed in each subproject separately.

In the following we describe briefly the single activities of the workflow; those activities for which the Generic Simulation Framework is essential (highlighted by a yellow background) are detailed in subsequent sections.

Prepare simulation project. This activity includes the definition of aims and scope of the simulation project and the selection of an appropriate set of simulation models to obtain the aimed result parameters.

Define data interfaces. After the participating simulation models have been selected, interfaces have to be defined which specify the data to be exchanged between the models. This task is detailed in Section 9.2.

Develop/Reengineer simulation models. With this task the responsibility within the workflow is passed over to the single model developers which have to implement new models or reengineer existing models so that they meet the requirements of the framework. See Section 9.3 for details.

Compile and test simulation models (local). Each model developer is obliged to compile and test his/her simulation model locally before passing it over to the central simulation administrator. For the purpose of a run time test the model developer can utilise the framework core as a test environment; possibly he or she needs some dummy models acting as data providers for the required interfaces.

Check in simulation models into central repository. If compile and run time tests in the previous activity have been passed error-free, the model developer is allowed to check in his/her simulation model into a central model repository[1]. The term simulation model includes here

- code and/or executables of the model,
- required data (e.g. initialisation data), that is not obtained via interfaces from other models during the simulation run,
- meta data of the simulation model.

[1]We assume that such a repository has been set up before.

Check out simulation models from central repository. As soon as all necessary simulation models are available in the repository the simulation administrator can check out these models (including data and meta data) and set up a simulation environment. A simulation environment is typically distributed on different computers in a local area network or on different nodes within a computer cluster.

Compile and test simulation models (integrative). Although the simulation models have been locally tested it is possible that they contain errors which emerge only in the interaction with other models. Hence the simulation models are again compiled (in order to avoid errors caused by different compiler versions) and tested under run time conditions.

Execute simulation run. If all tests have been passed error-free, the integrative simulation run can be executed. This task is detailed in Section 9.4. If the execution ends with an error message – which can occur despite intensive testing – the source of the error has to be investigated and the responsible developer(s) of the model(s) which caused the error have to be notified about the error.

Provide (raw) result data. The result data which have been produced during the simulation run must be provided to the model developers, for example by transferring them onto a file server.

Analyse and prepare result data. It is now the task of the model developer to analyse the result data and prepare the data for later publication.

Publish simulation results. The final step in a simulation project is to publish the results in an appropriate manner, for example within a scientific publication.

Finally note the following remarks on the workflow described in Figure 9.1.

1. The importance of the single activities typically varies during the environmental research project. While the definition of data interfaces and the development of simulation models demands a lot of time at the beginning of a project, the execution of simulation runs gets more and more important towards the end of the project. In the latter case, in particular when an adequate number of simulation models already resides in the repository, it is even possible to skip the aforementioned activities.

2. Within a simulation project often several simulation runs with the same set of participating models, but different configurations, i.e. variations of the model parameters, are performed. This is sometimes called a *simulation ensemble*.

3. In the case of a long computing time of the simulation run, it is advisable to provide some result data already during the simulation run (e.g. after one month simulation time

has been computed). This gives model developers the opportunity to check whether their model works correctly even in the combination with other models, and in case of need to request a rerun of the simulation.

9.2 Design of Data Interfaces

In an integrative simulation project where models shall be linked via interfaces it is important that the interfaces to be linked fit together.

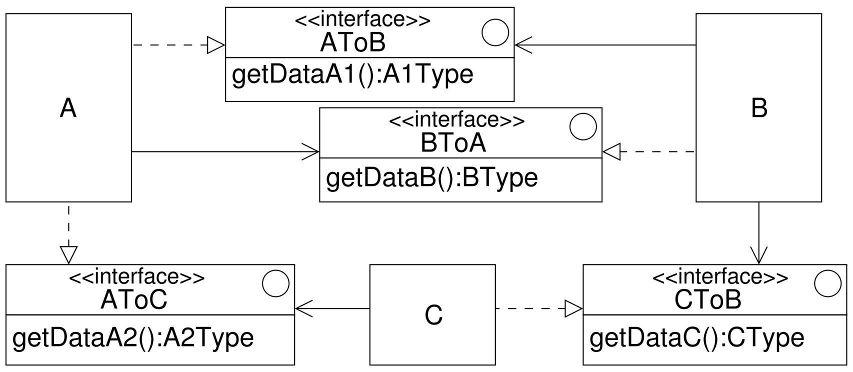


Figure 9.2: Models and Interfaces

In our approach we give the following recommendations for the definition of data interfaces (cf. Figure 9.2).

- A data interface specifies the data exchange between two simulation models, one model acting as provider and the other one acting as user of the data.
- In case of a bidirectional data exchange two interfaces should be specified.
- The name of the interface should reflect provider and user model (e.g. the interface AToB has provider A and user B).
- The operations for data exchange are required to comply to a predefined signature pattern of the form get$X()$:T, where X denotes a parameter name and T a predefined data type.

The use of project-wide common data types is strongly recommended to avoid errors caused by the usage of different physical units in the various disciplines. For example, water flux is typically measured by hectolitre per year in social sciences and by cubic metre per second in natural sciences. So an agreement has to be achieved which probably might consist in the usage of standard (SI) units. Possibly in some models data have to be converted in order to match the standard units, but this effort is by far compensated by avoiding mistakes and misunderstandings.

Furthermore, the interfaces can be enhanced by specifying the expected and assured ranges of exchanged data such that interface compatibility can be checked upon set up of a simulation configuration. A syntax for specification and a tool for checking the specifications was developed in [Wag07].

9.3 Development of a Simulation Model

The development of a simulation model is considerably facilitated by the developer interface of the Generic Simulation Framework. In particular, the model developer can concentrate on implementing the scientific code for the model which reflects the simulated process and is usually contained in the compute and the computeProxel operation respectively. He or she does not need to care about administrative issues, e.g. linking the models for data exchange or coordinating different time steps during the simulation, as these issues are already handled by the framework core.

In order to comply with the Generic Simulation Framework a simulation model implementation has to comprise

- a subclass of each abstract base class of the components Model and Proxel, i.e. subclasses of the classes AbstractModel and AbstractProxel, providing implementations of the respective mandatory plug-points; these subclasses are called *model class* and *proxel class* respectively;
- an instance of ModelMetadata containing the meta data of the model in an appropriate form.

The model class must also implement the provided interfaces of the model. The form of the meta data depends on how meta data are handled in the actual framework implementation; for example this could be a text file containing key-value pairs defining values for the attributes of ModelMetadata. Beyond that a model developer is free in the implementation of his/her simulation model.

In the field of computer-based environmental simulation many well-approved simulation models exist, often implemented in imperative programming languages like C or FORTRAN. In order to reuse such so called *legacy models* within the object oriented simulation framework, wrapping techniques can be applied. The model class then acts as a wrapper for the legacy model. Within the plug-points appropriate routines of the FORTRAN model can be called (in the case of Java for example by using the Java Native Interface).

Figure 9.3 provides an example for the application of the developer interface on the development of a sample Groundwater model. While the upper part of the figure (above the dashed line) shows the general framework classes, the lower part contains classes and objects which have (at least) to be implemented or instantiated by the model developer. While the base classes AbstractModel and AbstractProxel have to be specialised by concrete subclasses (in Figure 9.3 by Groundwater and GroundwaterProxel respectively), the object of type ModelMetadata represents the meta data of the model.

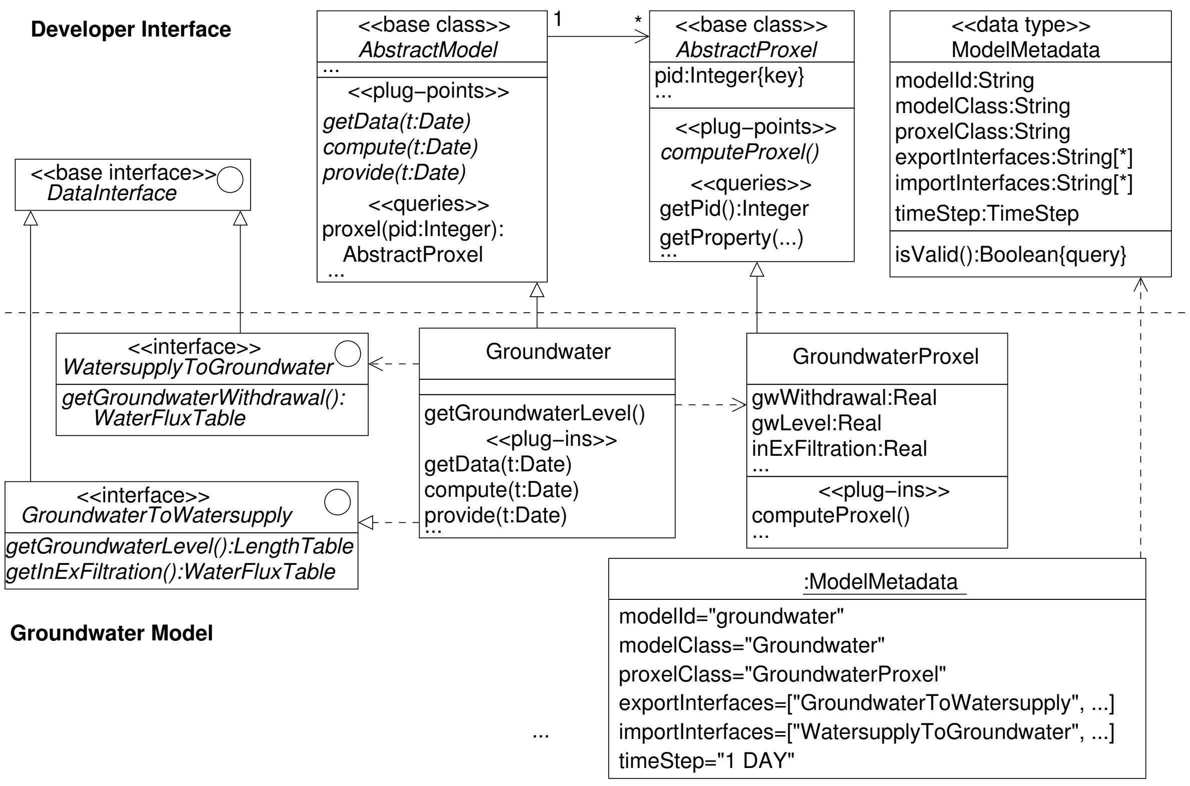


Figure 9.3: Application of the Developer Interface for the creation of a simulation model

9.4 Execution of an Integrative Simulation Run

For the execution of an integrative simulation run a user interface for accessing the simulation framework is indispensable. As the requested functionality of a user interface strongly depends on the requirements of the actual research project, the user interface is not part of the framework. However, in Chapter 4 a couple of interfaces (SimulationAccess and UserInterface) have been developed which provide access to the component Simulation for a user interface on the one hand, and define the minimum requirements for a user interface on the other hand.

Figure 9.4 shows these interfaces once more in the context of a possible user interface implementation UserInterfaceImpl and the framework core class SimulationAdmin.

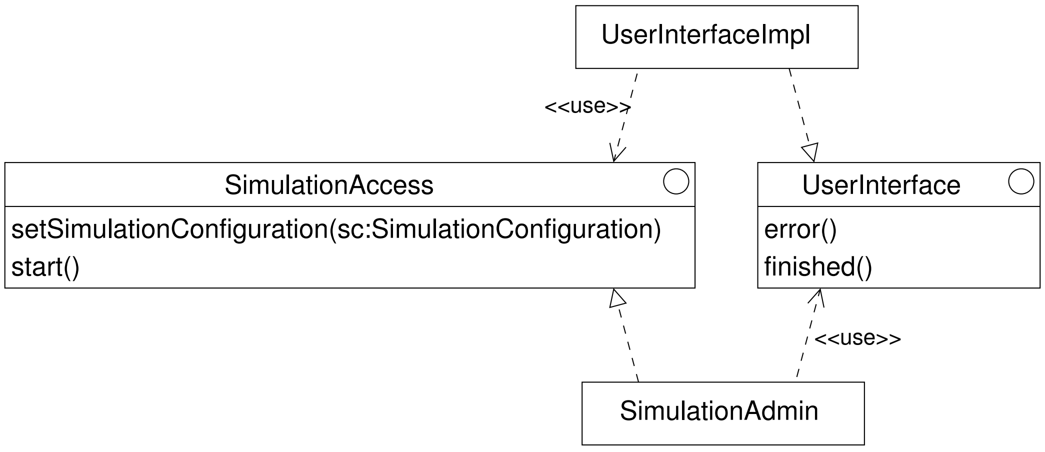


Figure 9.4: Interfaces between user interface and simulation framework

The interface SimulationAccess which is implemented by SimulationAdmin and used by a user interface implementation comprises the operations setSimulationConfiguration and start. On the other hand, the interface UserInterface contains the operations finished and error, by which the SimulationAdmin notifies the user interface about the normal or abnormal termination of a simulation run.

Hence a user interface for the Generic Simulation Framework must be able to

- create a SimulationConfiguration object and pass it as parameter to the setSimulationConfiguration operation,
- send the start signal,
- receive the signals finished and error and notify the user about the reception.

Moreover, the user interface implementation must be able to check, whether a SimulationConfiguration is valid or not.

To set up a simulation configuration is typically a non-trivial task as it requires intrinsic knowledge about the available simulation models, their interfaces and the goal of the simulation, i.e. the kind of result that shall be obtained. The activity diagram in Figure 9.5 refines the activity Execute simulation run which is part of the overall workflow introduced in Section 9.1. It comprises two sub-activities, Configure simulation and Start simulation. The activity Configure simulation contains several sub-activities which can be executed in parallel and with which the properties of a SimulationConfiguration (cf. Figure8.3 on page 137) can be defined. These properties include the simulation identifier, the set of participating models, simulation begin and end, the simulation area and the set of resources describing the simulation space (cf. Chapter 6). Note that the activity Start simulation can only be executed if the simulation configuration is valid.

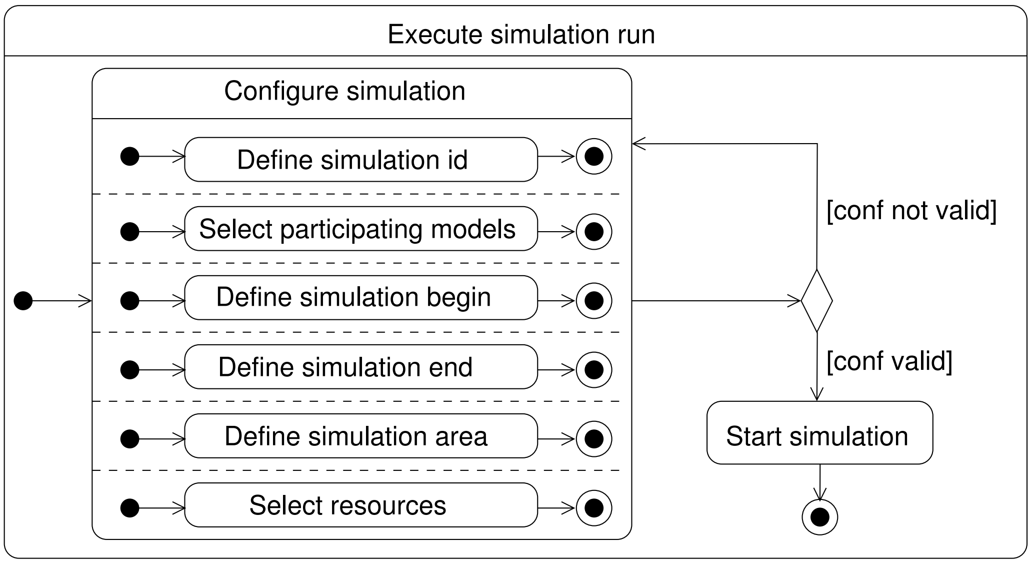


Figure 9.5: Sub-activities of executing a simulation run

Figure 9.6 shows an object diagram containing a sample SimulationConfiguration object with its properties. Note that the query isValid must result to true – which is specified by an OCL condition in Section 8.1 – in order to perform an integrative simulation with the respective configuration. Obviously this is the case for the object depicted in Figure 9.6.

An example for a graphical user interface fulfilling the aforementioned requirements is given in Section 11.3.2 in the context of the DANUBIA system.

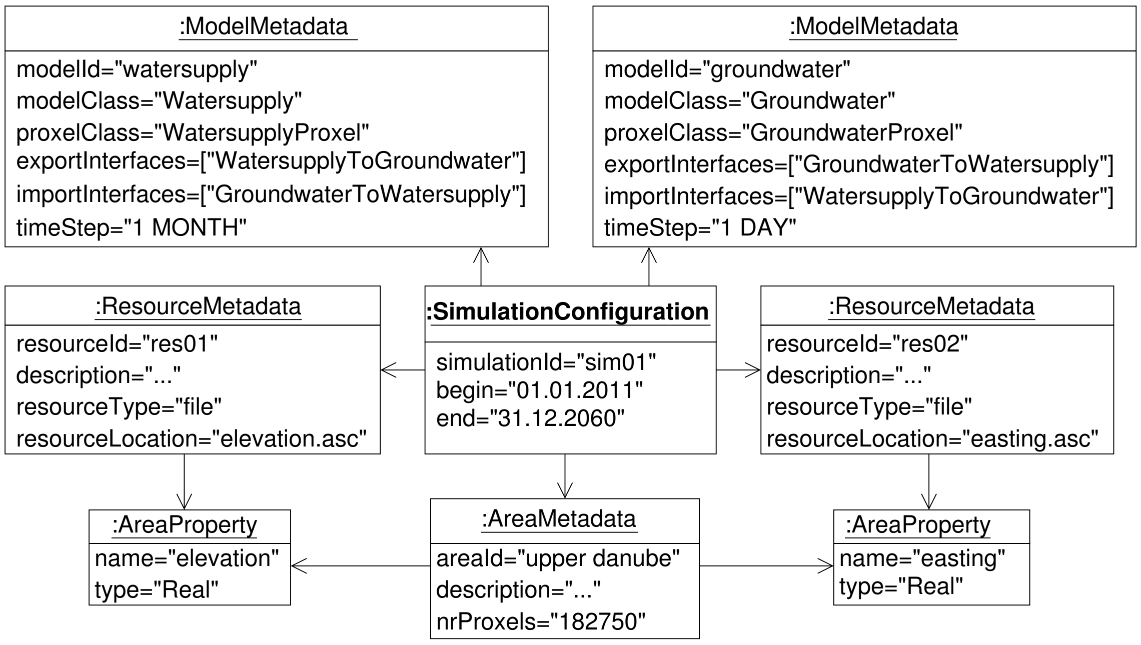


Figure 9.6: Providing a Simulation Configuration

10

Implementation of the Simulation Framework in Java

In this chapter we describe a reference implementation of the Generic Simulation Framework in the object-oriented programming language Java.

10.1 Implementation Architecture

The implementation architecture of the Generic Simulation Framework is based on a *distributed client/server architecture*. The network connections between the distributed components are established by so called *dynamic proxies*. We detail on these issues in the remainder of this section.

10.1.1 Distribution

Usually an integrative simulation system is quite resource intensive. In order to obtain an optimal usage of the available computation resources the Generic Simulation Framework is implemented as a distributed system based on a client/sever architecture. The idea of this architecture is sketched in Figure 10.1. A centralised SimulationServer covers the central components of the framework (Simulation, BaseData, ModelLinking and TimeCoordination) whereas the single simulation models (i.e. instances of the framework component Model) reside in several SimulationClients distributed over a network. While the solid lines in Figure 10.1 denote the network connections between the centralised server and the clients for controlling the simulation, the dashed lines between the clients denote the data connections between the simulation models.

The network communication between the server and the clients is established by the Java Remote Method Invocation (RMI, cf. [Gro01]) architecture. The same holds for the connec-

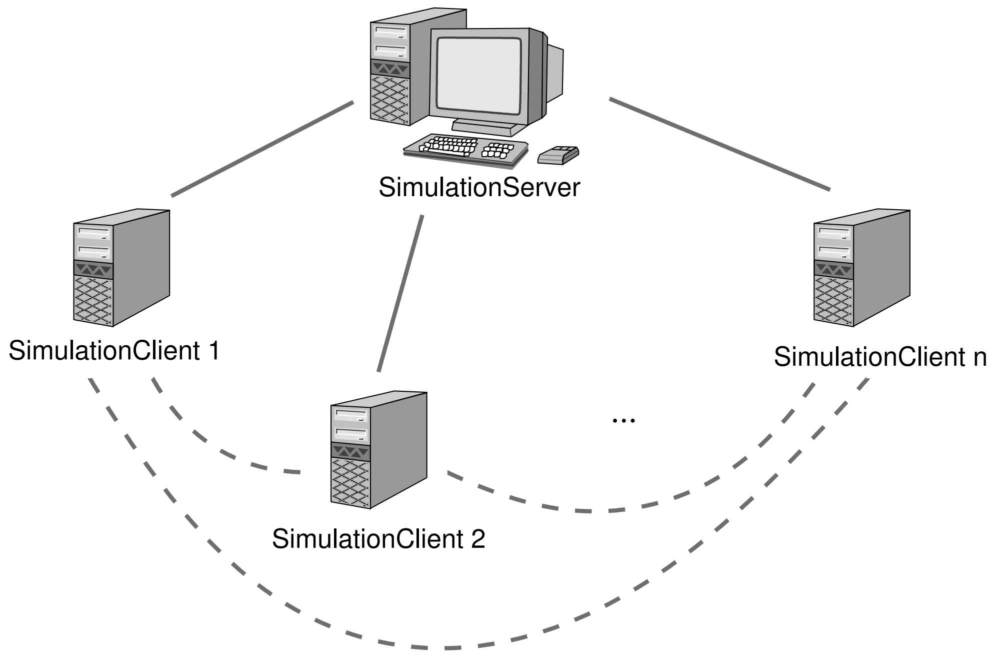


Figure 10.1: Client/Server architecture of the Generic Simulation Framework

tions between the clients for data exchange. To enable remote access to a class, this class must implement a remote interface, i.e. an interface that inherits from the predefined interface `java.rmi.Remote` (in our diagrams we denote such interfaces by the stereotype «remote»). Figure 10.2 shows the classes SimulationServer and SimulationClient which provide access to the system on the server and the client side respectively (through their main methods).

The course of action for starting the system is the following. First, a SimulationServer has to be started and registered to the RMI naming service (RMIRegistry). From this moment the server is accessible via the remote interface SimulationAccess for the user interface on the one hand, and via the remote interface ClientRegistration for the clients on the other hand. After the server has been started, the SimulationClients may be instantiated on their different nodes, provided with the network address of the SimulationServer. Via the network address and the RMI naming service the server instance can be obtained, and the client registers itself by calling the method register of the interface ClientRegistration. The server stores the registered clients. When a simulation run is started, the server calls createSimulationClientManager on each client, which will initiate the instantiation process of the single simulation models.

10.1.2 Dynamic Proxies

When using RMI for network communication it is usually necessary to add RMI specific code to the involved classes and interfaces. In particular, an interface designated for remote communication has to inherit from `java.rmi.Remote` and each operation has to declare to

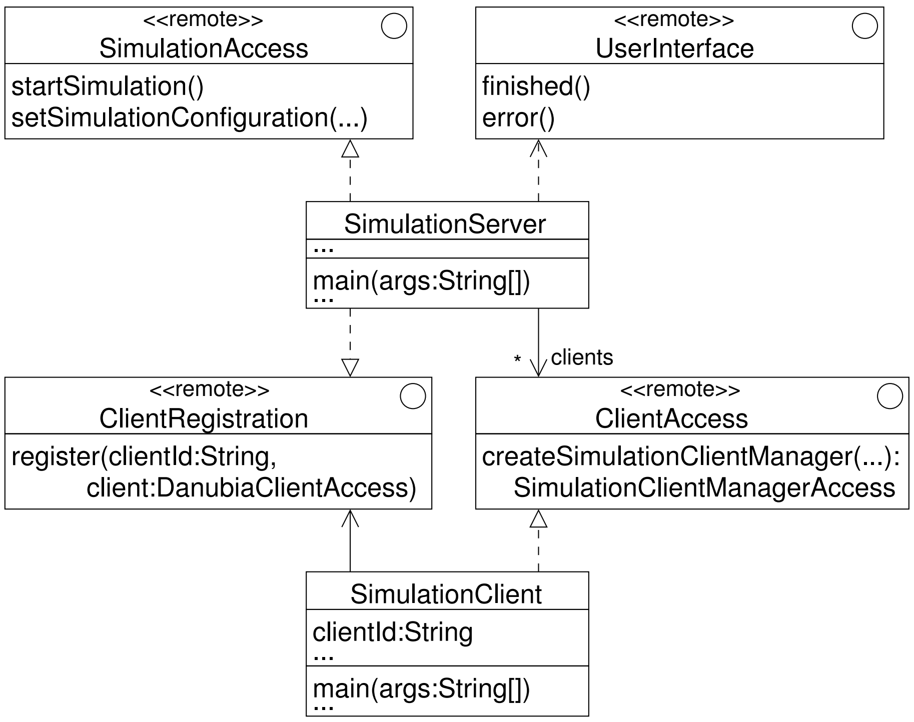


Figure 10.2: Simulation Server and Simulation Client

throw a remote exception. In order to avoid this we introduced the concept of *dynamic proxies*. The principle of dynamic proxies is depicted in Figure 10.3.

The figure is divided into three layers. The uppermost layer contains classes and interfaces provided by the Java programming interface. The middle layer shows the actual dynamic proxy implementation and the lowermost layer exemplifies the application of dynamic proxies on a network connection between two classes A and B via an interface I. We do not want to go into detail, but just want to point out, that the dynamic proxies work on basis of reflection and are used for all kinds of network communication in the Generic Simulation Framework (except for the initial communication described above).

10.2 Implementation Patterns

In this section we provide some implementation patterns which have been used within the reference implementation. With the first pattern design components are transformed to Java, the second pattern shows how the enable construct used in sequence diagrams can be implemented with synchronised Java methods, and finally we say some words about the implementation of parallel combined fragments.

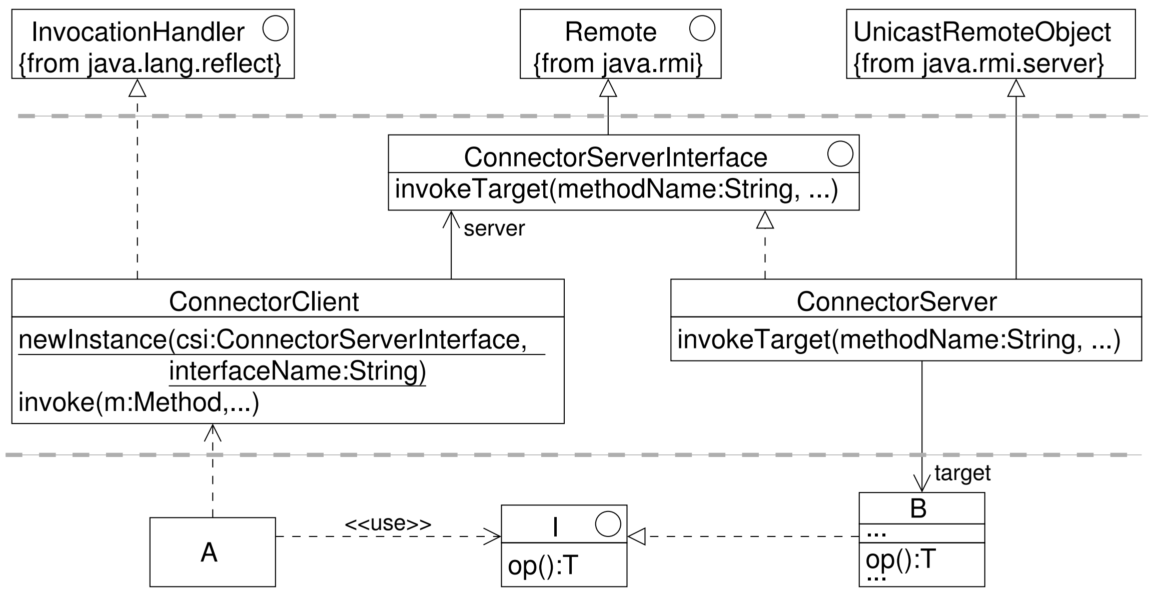


Figure 10.3: Dynamic Proxy architecture

10.2.1 Components

As Java does not provide a means for implementing components directly we have to substitute a UML design component with an appropriate construct. Of course a couple of approaches dealing with components and their implementation exist, like e.g., Java/A [BHH+06], ArchJava [ACN02], SOFA [BHP06], Fractal [BCL+06], or the Catalysis approach [DW99], but none of them turned out to be fully suitable for our purposes. Hence we developed a simple implementation pattern for the components used in our methodology based on the principle of *indirect implementation* (cf. [HKKR05]). In the following we describe this pattern and illustrate it by an example.

The namespace of a component is represented by a package which is named with the component's name in lowercase letters. Subcomponents are represented by nested packages. A package representing a component is denoted by the stereotype «comp impl» and contains

- implementation classes (interfaces) for all design classes (interfaces) which belong to the internal structure of the component;
- the provided interfaces of the component;
- a public class named with the component's name and the suffix `Manager`.

The implementation pattern is exemplified in Figure 10.4. The package h realises the hierarchical component H. It comprises the nested packages c1 and c2 which represent the components C1 and C2 respectively. Each (nested) package comprises a public manager class which is used for accessing the component from outside, i.e. from the next higher level. Implementing objects of interfaces which are required by the component are passed to the constructor

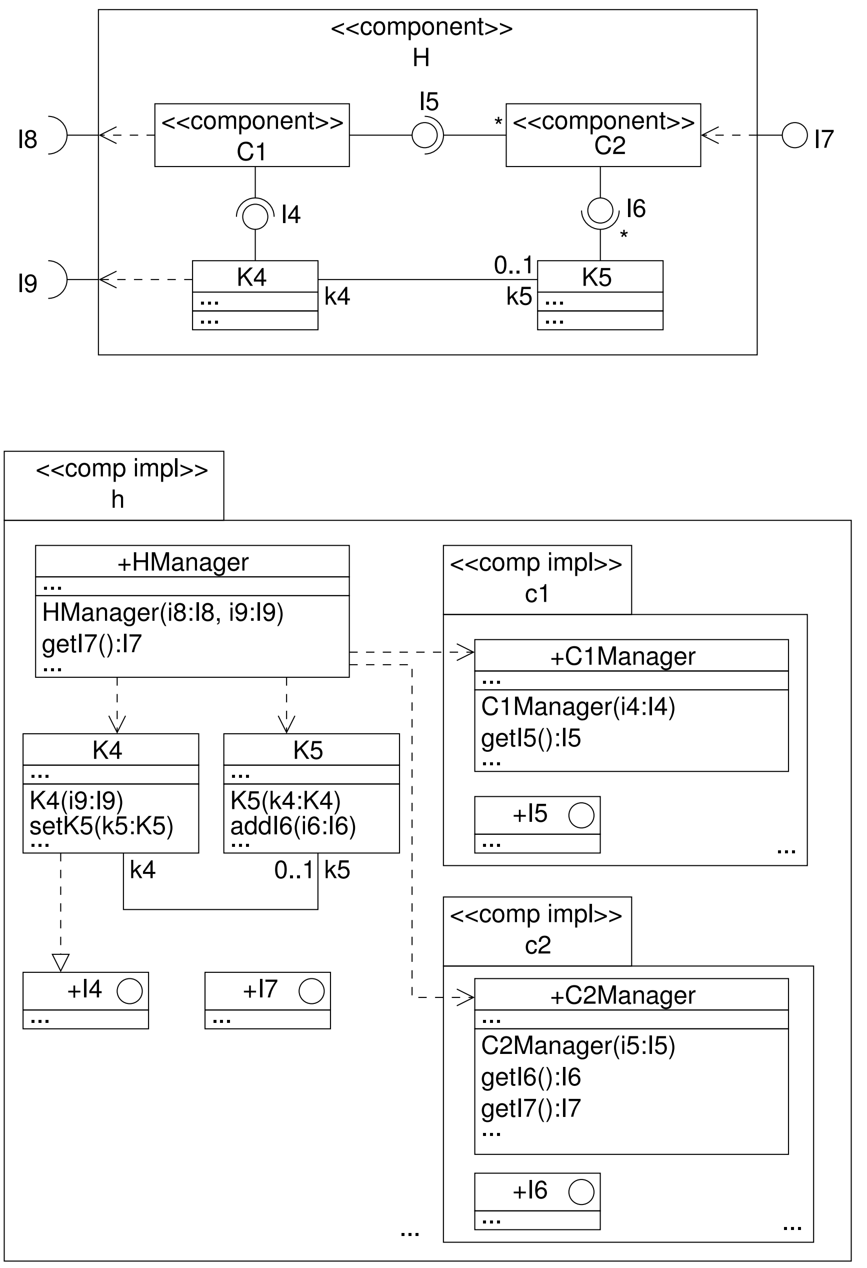


Figure 10.4: Implementation pattern applied to the component H

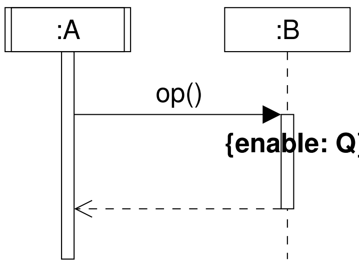
of the manager class (e.g. objects implementing the interfaces I8 and I9 are passed to the constructor of HManager). As the association between the classes K4 and K5 has multiplicity 0..1 at the association end near K5, an implementing object of K5 has not to be created within or passed to the constructor of K4 but rather may be set by calling the operation setK5 later. Similarly, the operation addI6 of the class K5 adds an implementing object of type I6. Note that this interface type has multiplicity *. Provided interfaces are contained in the package of the outermost component which provides the interface. Hence the interface I5 is contained in the package c1, but the interface I7 in the package h.

The following code excerpt of the class HManager shows the implementation of the constructor of HManager which illustrates the instantiation of the single elements of the package h and the binding of the interfaces by means of the operations described above.

```
public class HManager {
  I7 i7;
  public HManager(I8 i8, I9 i9) {
    K4 k4 = new K4(i9);
    K5 k5 = new K5(k4);
    C1Manager c1Manager = new C1Manager(k4, i8);
    C2Manager c2Manager = new C2Manager(c1Manager.getI5());
    k5.addI6(c2Manager.getI6());
    i7 = c2Manager.getI7();
  }
  public getI7() { return i7; }
...
}
```

10.2.2 Enable Conditions

The implementation of an operation with an enable condition is shown beneath.



```
public class B {
  ...
  public synchronized void op()
    throws InterruptedException {
    while (!Q) wait();
    ... // do something
    notifyAll();
  } ...
}
```

10.2.3 Parallel Combined Fragments

During the integration process a number of parallel combined fragments appeared in the design level sequence diagrams as this was the appropriate way to combine interactions from different views. For the implementation, however, parallelisation would cause quite more effort, so we restrict ourselves here to implement the operands of each parallel fragment in an (arbitrary) sequential order. This is not contradictory to the design as any sequential order of the operands of a parallel fragment results in a correct refinement of the fragment.

11

DANUBIA – an Integrative Environmental Simulation System

In this chapter we show how the generic simulation framework has been successfully applied within the integrative environmental project GLOWA-Danube to construct the integrative simulation system DANUBIA[1]. For this purpose we first provide a brief introduction to the GLOWA-Danube project in Section 11.1, before in Section 11.2 the DANUBIA system is described. Finally, Section 11.3 presents how DANUBIA has been applied to calculate so called "GLOWA-Danube scenarios" by performing integrative simulation runs.

11.1 GLOWA-Danube: Overview

GLOWA-Danube[2] is a research and development project focusing on the comprehensive analysis of the future of water resources of the Upper Danube. In GLOWA-Danube the impact of Climate Change of a broad range of sectors is investigated. Furthermore the project identifies and simulates strategies for adaptation to and mitigation of the consequences of Climate Change and tests their effectiveness. In GLOWA-Danube a team of researchers from different natural and socio-economic science disciplines work closely together in an interdisciplinary, university-based competence network since 2001.

The aim of GLOWA-Danube is to investigate with different scenarios the impact of change in climate, population and land use on the water resources of the Upper Danube and to develop and evaluate regional adaptation strategies. For this purpose the simulation and decision support system DANUBIA was successfully set up within the first and second project stage (2001-2006).

[1]Danubia is the Latin name of the river Danube
[2]cf. `http://www.glowa-danube.de`

Besides the Informatics group research groups from the following scientific disciplines participated in GLOWA-Danube:

- Hydrology/Remote Sensing
- Meteorology
- Groundwater/Water Supply
- Ecosystems/Plant Ecology
- Glaciology
- Environmental Psychology
- Environmental Economics
- Tourism Research
- Agricultural Economics
- Regional Climate Modelling
- Water Resources Management

The investigation area of GLOWA-Danube is the watershed of the Upper Danube (cf. Figure 11.1). The Upper Danube with its more than 10 million inhabitants and an area of 77.000 km^2 is one of the largest and most important alpine watersheds in Europe. With its strong relief and the altitudinal gradient of up to 3.600 m, the Upper Danube is particularly vulnerable to Climate Change. These conditions also lead to a remarkably broad range of influencing factors on the water resources. The watershed includes glaciers as well as temperate lowlands, which are intensively used by agriculture. The Upper Danube in addition is characterised by a complex and intensive use of the water resources for hydropower, farming (possibly future irrigation) and tourism (e.g. snow cannons). The watershed of the Upper Danube therefore combines in an exemplary way a lot of water use problems of Central Europe.

11.2 The Danubia System

DANUBIA is an integrative simulation system based on the Generic Simulation Framework described in the previous chapters of this thesis. It includes – to our knowledge for the first time – simulation models for natural science as well as socio-economic processes and their interactions. With the intention of being predictive DANUBIA uses results of regional climate models for predictions on Climate Change. Physical and physiological components describe natural processes (hydrology, hydro-geology, plant physiology, yield, and glaciology). For the

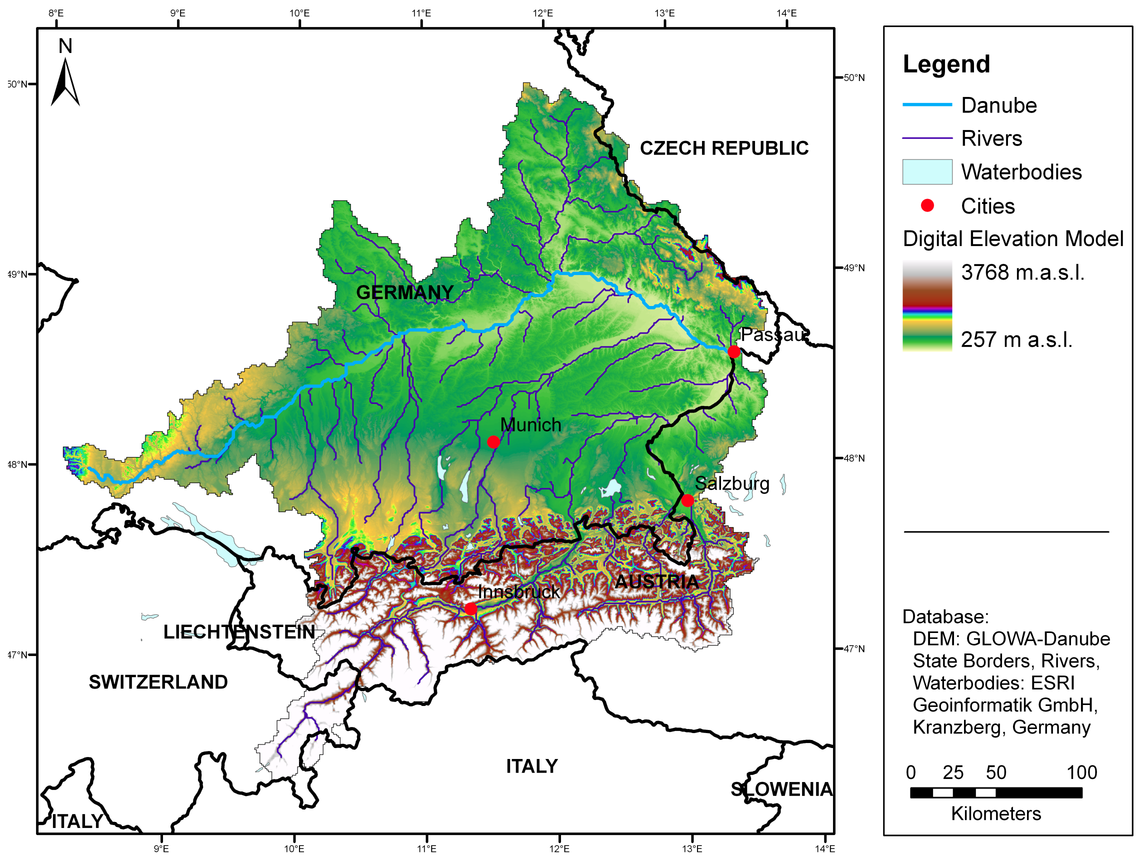


Figure 11.1: The Upper Danube Basin – the Investigation area of GLOWA-Danube

simulation in the social sector (farming, economy, water supply companies, private households and tourism) DANUBIA uses agent-based models which represent the decisions of the involved actors based on the structure of societies, their environment as well as their interests.

DANUBIA was carefully and successfully validated with comprehensive data sets of the years 1970–2005 and is now available in the third stage of the project for common use for project researchers and stakeholder. The single parts of DANUBIA, including the Generic Simulation Framework, are available under an open source licence (OpenDanubia[3]) and will particularly serve decision makers from policy, economy, and administration as tool for a foresighted planning of water resources against the background of Global Change.

The following publications detail on some selected simulation models of the natural science part of DANUBIA: [BBM08] focus on the groundwater model within DANUBIA and, finally, in [LWKS10] the crop growth model within the land surface component of DANUBIA is described.

[3]`http://www.glowa-danube.de/eng/opendanubia/opendanubia.php`

11.2.1 Architecture

Figure 11.2 shows the architecture of DANUBIA in a quite intuitive manner. The Generic Simulation Framework surrounds the simulation models of the various scientific disciplines which are summarised in the five major components Atmosphere, Actor, Landsurface, Groundwater and Rivernetwork. The embedding of the models into the framework is depicted by UML inheritance arrows which indicate the usage of abstract base classes to implement a simulation model by inheritance. The connectors denote the data exchange between the single simulation models at run time.

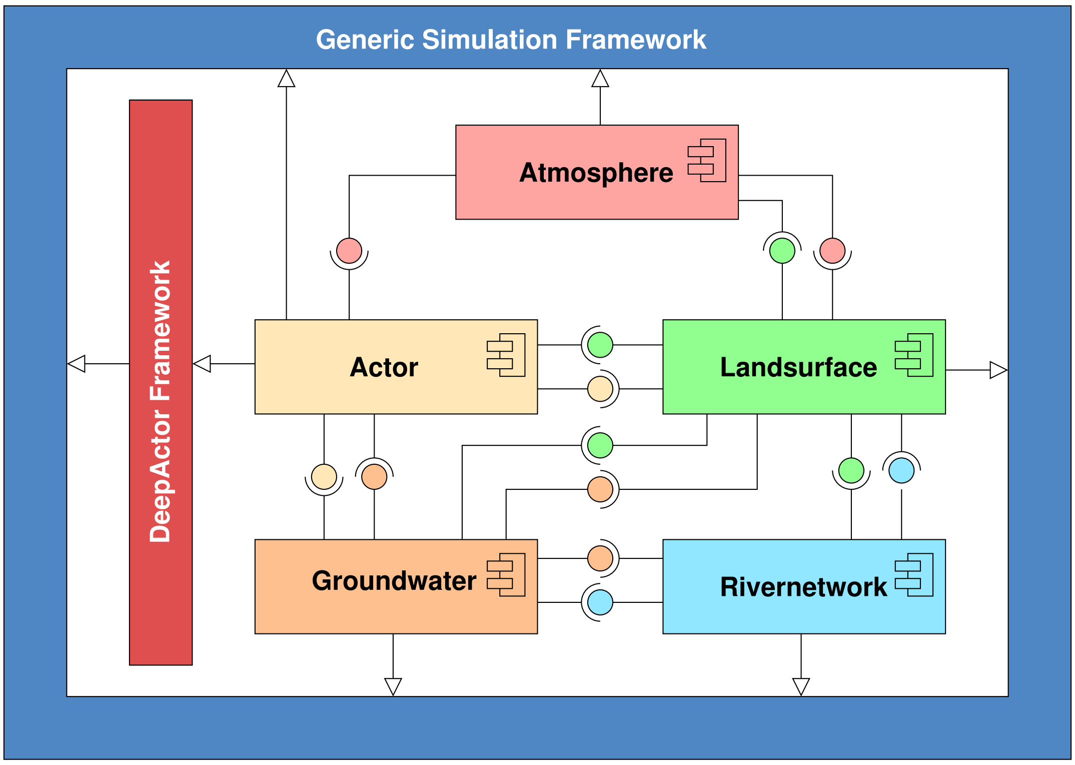


Figure 11.2: Architecture of the DANUBIA system

Within DANUBIA the component Actor plays a special role as it is embedded into the Generic Simulation Framework as well as into the DEEPACTOR Framework which provides support for the special needs of agent-based social simulation models. The DEEPACTOR Framework is in turn embedded into the Generic Simulation Framework again. In the next section we will overview the DEEPACTOR Framework.

11.2.2 Support for Agent-Based Social Simulation

As GLOWA-Danube was – to our knowledge for the first time – the first project where simulation models from natural and social sciences were coupled much effort has been made for supporting social simulation models. Usually, in social sciences the agent-based simulation approach prevails, in contrast to the process-based approach in natural sciences.

To fill the gap between theses simulation approaches, the so called DEEPACTOR framework has been developed providing a common architecture for socio-economic simulation models (cf. [HJL10]). The underlying concepts of the so called DEEPACTOR approach, which extend and reuse the concepts of the Generic Simulation Framework are depicted in Figures 11.3 and 11.4 respectively (the concepts of the Generic Simulation Framework are denoted by a grey background). An ActorModel which extends a general simulation Model refers to a set of Actors. Each Actor has a unique identifier and defines a number of type specific properties (actorProperty1, etc.). The location of an actor within the simulation area is defined by a set of (not necessarily connected) proxels. An actor may be part of a social network which is defined by its collaborators.

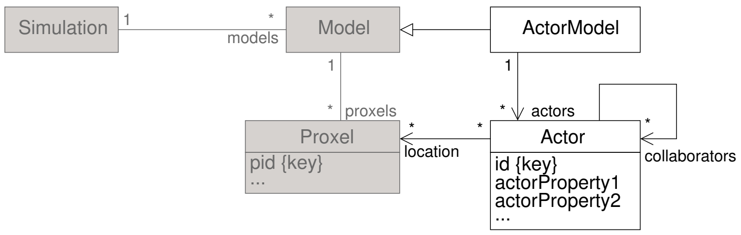


Figure 11.3: Concepts of the DEEPACTOR approach (1)

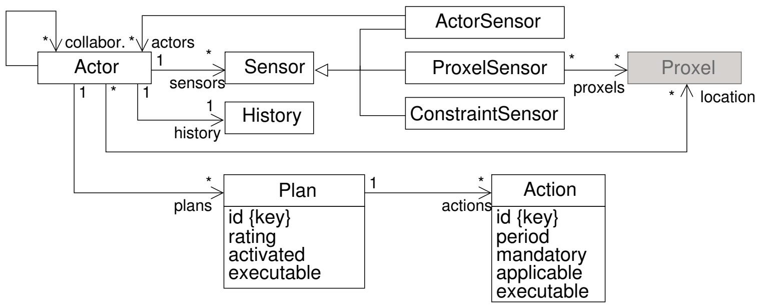


Figure 11.4: Concepts of the DEEPACTOR approach (2)

Actors perceive their environment explicitly via their Sensors. Sensors allow for the reception of data or events. Three concrete kinds of sensors are distinguished: A proxel sensor for the simulation area, an actor sensor for information exchange with other actors of the same model and a constraint sensor. Constraints are an explicit concept for the modelling of further influencing aspects such as legal constraints, for instance.

Besides sensors, actors dispose of a History that allows to "remember" the plan execution status of previous time steps. By this means a basic mechanism for actors with learning capabilities is provided. Based on their local state, their history and their perceived environmental state, actors decide which Plan is to be selected for execution within a given time step. A plan, in turn, is related to a set of Actions which model the concrete impact of plan execution. In this vein, plans represent the possible courses of action of an actor.

The actions of a plan are to be executed if the respective actor decided for the particular plan (i.e. activated the plan), and the plan is executable. The decision to activate a plan might be based on its rating which allows to realise multi-attribute utility based decision algorithms conveniently. A plan is executable only if all of its mandatory actions are executable in the current time step—an action is either mandatory or optional, which is an initially assigned property of an action. Whether an action is executable depends on two type-specific properties that are evaluated dynamically within each time step: first, the action needs to be applicable and, second, the period of the action needs to be consistent with the current simulation time. For instance, harvesting is an action of a farming actor that is relevant only during certain months of a year. Both, the plans of an actor and the actions of a plan are required to be initially known and must not change during a simulation run.

The design of the DEEPACTOR framework follows again our general pattern which distinguishes core classes and (abstract) base classes for model developers, exemplified by the classes ActorModelCore and AbstractActorModel in Figure 11.5. The remaining components of the DEEPACTOR framework are implemented similarly and we omit further details here.

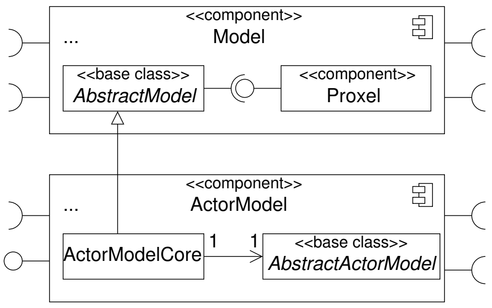


Figure 11.5: Integration of the DEEPACTOR Framework with the Generic Simulation Framework

The integration of the DEEPACTOR framework into the Generic Simulation Framework is achieved by extension of the developer interface of the overall framework. As illustrated by the excerpt in Figure 11.5, the core part of the DEEPACTOR framework specialises (solely) the developer interface of the general framework, thus being transparent for the core layer of the Generic Simulation Framework. Concrete actor models then use the developer interface of the DEEPACTOR framework and, for basic functionality, the developer interface of the Generic Framework such as AbstractProxel.

A concrete application of the DEEPACTOR framework for the simulation of water-related issues in the Upper Danube basin is reported in [ESSJ07], describing a simulation model for the water consumption of households, and in [BJS+08] who developed a model for water supply companies as an important link between socio-economic and natural science simulation models. An agent-based simulation model for the diffusion of environmental innovations is described in [SE09]. Finally, the work of [BJN+10] reports in more detail on the general approach as well as the integrative aspects of aforementioned simulation models.

11.3 Scenario Calculations with Danubia

In this section we report on the application of DANUBIA within GLOWA-Danube and sketch some simulation results that have been obtained.

11.3.1 GLOWA-Danube-Scenarios

The analysis of scenarios form a common technique in environmental modelling (cf., e.g., [KD95]). In our context a scenario is defined as a set of consistent assumptions concerning the future development of the environment, in particular the future of the water cycle in the Upper Danube basin. Scenarios are realised by integrative simulation runs with the DANUBIA system.

The assumptions made for a scenario are expressed by certain variables and quantities (called *key factors*) influencing the behaviour of the simulation models participating in the integrative simulation run which realises the scenario. A scenario is then given by an assignment of a value for each key factor. Two major independent fields of influencing variables have been detected in GLOWA-Danube:

- the future development of the climate, and
- future societal trends.

As it is impossible to perform a simulation run for each potential value of a key factor, a set of such values describing a feasible development of the respective key factor in the future has been developed for each key factor, independently in both fields mentioned above. Thereby the so called *scenario matrix* (cf. Figure 11.6) has been elaborated which we briefly explain in the following.

By means of the scenario matrix a GLOWA-Danube scenario is defined by selecting a particular value in column 1, 2, and 3, and a (possibly empty) set of values of column 4. The first to columns regard to a climate scenario which is combined by a climate trend and a climate variant. While the climate trend defines particular values for the key factors, the climate variant describes certain weather phenomena, like e.g. five consecutive warm winters, etc. As an example for a key factor within the climate trend consider "temperate increase in the next 100 years", the possible values of which are 3,3 %, 5,2 %, or 4,7 % – corresponds to the selections IPCC regional, REMO regional, and MM5 regional, respectively. Columns three and four regard to the societal future trends, which are mainly defined by the so called society scenario, and specified by the choice of certain actions performed by particular actors.

Note that different selections in the scenario matrix lead to different results of the simulation run corresponding to the scenario. These results together with the assumptions made for the scenarios are then discussed with stakeholders from economy or policy in order to provide support for their decisions concerning environmental issues. Detailed definitions of the scenarios, in particular the actual values of the key factors corresponding to each scenario, can be found in the Global Change Atlas for the Upper Danube watershed [GLO10] which is also available online[4].

Selection 1: Climate Trend	Selection 2: Climate Variant	Selection 3: Society Scenario	Selection 4: Action
IPCC Regional	Baseline	Baseline	Action 1
REMO Regional	Five Warm Winters	Performance	Action 2
MM5 Regional	Five Hot Summers	Common Public Interest	Action 3
Foreward Projection	Five Dry Years		. . .

Figure 11.6: Scenario Matrix of GLOWA-Danube (cf. [GLO10]

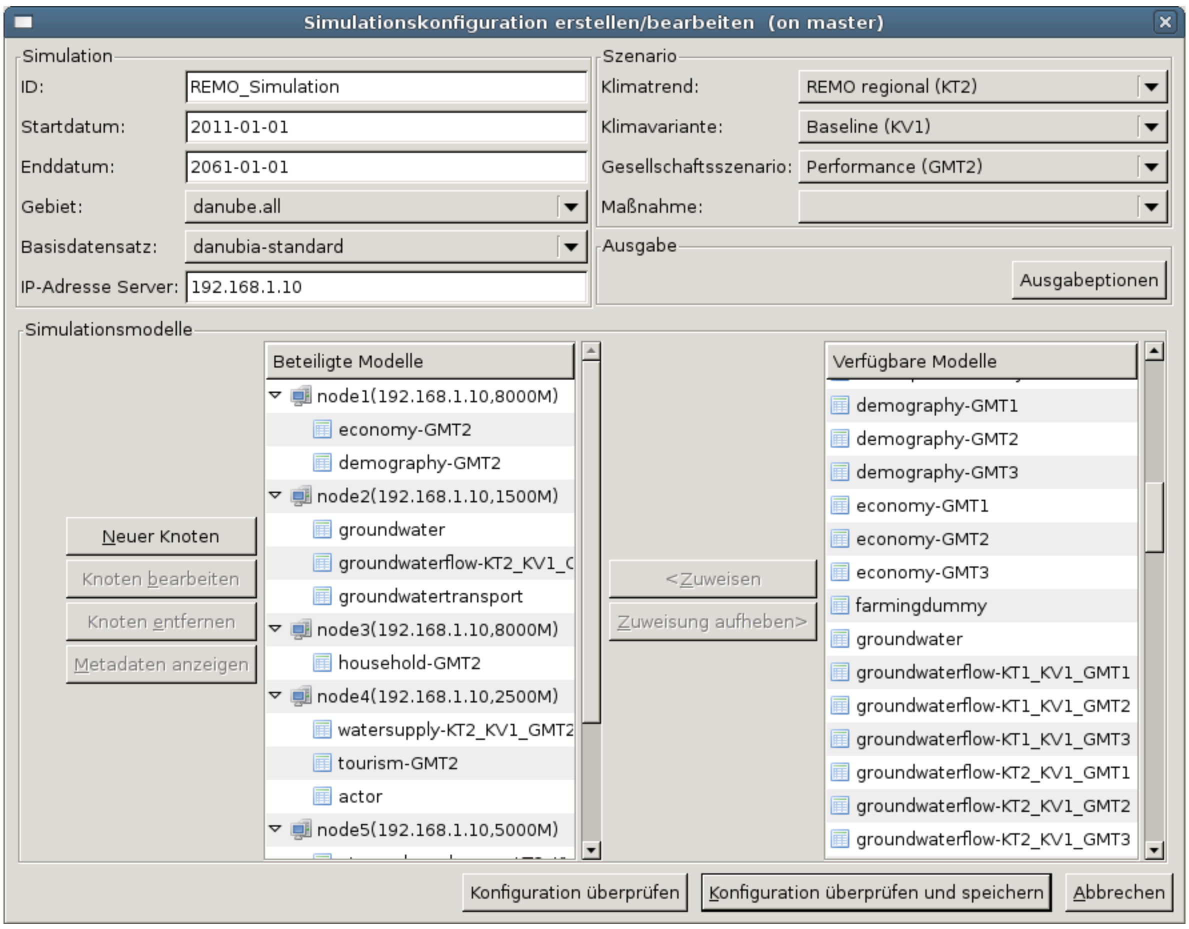
11.3.2 Performing an Integrative Simulation

For the execution of integrative simulation runs DANUBIA comprises a graphical user interface, the so called DANUBIA Monitor, which is implemented using the Eclipse Standard Widget Toolkit (SWT, cf. [NW04]).[5] The following screenshots show typical tasks of the DANUBIA Monitor.

The screen window in Figure 11.7 allows for configuring an integrative simulation. In the upper left compartment one can find text areas for entering values for an identifier, the start and end date, as well as list elements for selecting a simulation area and a set of base data resources from a list of possible values. The lower part of the screen allows for selecting simulation models to participate in the simulation run; simultaneously the models are assigned flexibly to different nodes, i.e. computer resources, in order to gain an optimal distribution of the models with respect to performance.

[4]`http://www.glowa-danube.de/atlas/`

[5]For the prototype of DANUBIA a Web-based user interface was developed applying Web engineering techniques (cf. [Kra07]), but this user interface is no longer supported for security reasons.



Simulation

ID:	REMO_Simulation
Startdatum:	2011-01-01
Enddatum:	2061-01-01
Gebiet:	danube.all
Basisdatensatz:	danubia-standard
IP-Adresse Server:	192.168.1.10

Szenario

Klimatrend:	REMO regional (KT2)
Klimavariante:	Baseline (KV1)
Gesellschaftsszenario:	Performance (GMT2)
Maßnahme:	

Simulationsmodelle

Beteiligte Modelle
- node1(192.168.1.10,8000M)
  - economy-GMT2
  - demography-GMT2
- node2(192.168.1.10,1500M)
  - groundwater
  - groundwaterflow-KT2_KV1_C
  - groundwatertransport
- node3(192.168.1.10,8000M)
  - household-GMT2
- node4(192.168.1.10,2500M)
  - watersupply-KT2_KV1_GMT2
  - tourism-GMT2
  - actor
- node5(192.168.1.10,5000M)

Verfügbare Modelle
- demography-GMT1
- demography-GMT2
- demography-GMT3
- economy-GMT1
- economy-GMT2
- economy-GMT3
- farmingdummy
- groundwater
- groundwaterflow-KT1_KV1_GMT1
- groundwaterflow-KT1_KV1_GMT2
- groundwaterflow-KT1_KV1_GMT3
- groundwaterflow-KT2_KV1_GMT1
- groundwaterflow-KT2_KV1_GMT2
- groundwaterflow-KT2_KV1_GMT3

Figure 11.7: Configuration of a GLOWA-Danube Scenario Simulation

In the upper right compartment of the screen the GLOWA-Danube scenario is determined by selecting values for climate trend, climate variant, society scenario, and action from a list element. The buttons on the bottom of the screen allow for checking if the entered simulation configuration is valid. The validity of a simulation configuration is necessary for starting the respective simulation run.

Note that with this screen the user interface DANUBIA Monitor complies with the requirements for a user interface concerning the creation and validation of a simulation configuration stated in Section 9.4. The possibility of defining GLOWA-Danube scenarios denotes a project-specific extension.

The screen window depicted in Figure 11.8 shows the main window of the DANUBIA Monitor, yet the screenshot has been made during a simulation run. It shows in the upper half the configuration and in the lower half the state and the progress of the current simulation, and displays, in particular, the current activities of the simulation models within the computation cycle. The protocol of these activities can be used for further performance analysis and optimisation.

Again, this screen complies with the requirements of the Generic Simulation Framework by displaying the current state of the simulation run, and by giving the opportunity to start a simulation run with the respective button. All other features of the user interface are considered as project-specific extensions.

11.3.3 Some Simulation Results

The results of the manifold simulation runs during the GLOWA-Danube project are reflected in numerous scientific publications, reports and dissertation projects of the participating disciplines. Concluding project results are described, among others, in the following publications. [BME$^+$10] covers the impacts of Climate Change on water resources, groundwater recharge, groundwater levels, and groundwater quality; [BMS$^+$10] concentrates on the socio-economic impacts of Climate Change on water use and land use in the Upper Danube basin. A complete list of publications and reports is available on the GLOWA-Danube web page[6].

The core results of the GLOWA-Danube project regarding the future of the water cycle in the Upper Danube basin are subsumed in Section E6 “GLOWA-Danube results and key messages” of [GLO10] which we briefly summarise in the following.

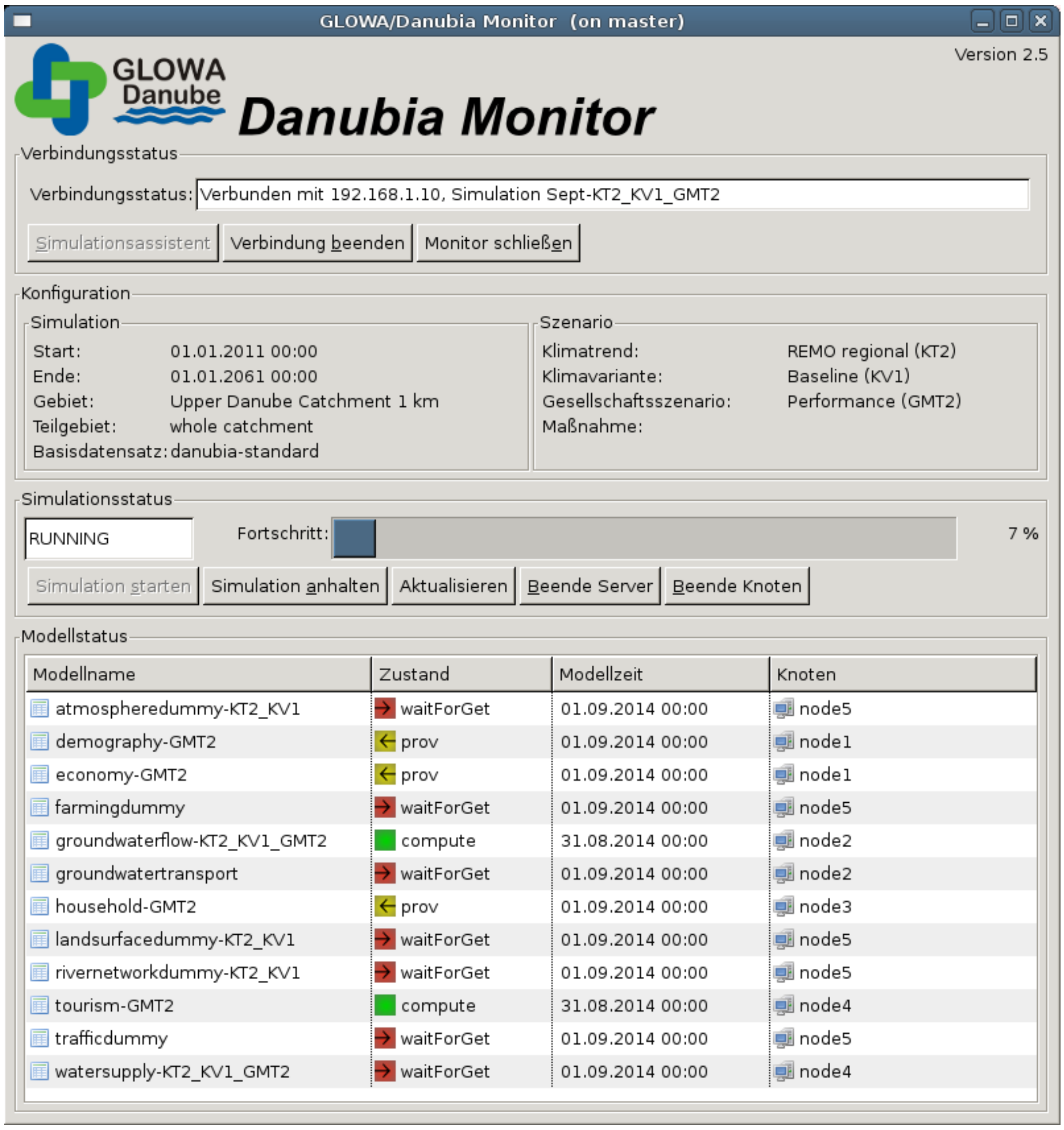
Scenario runs with DANUBIA are based on the findings of the IPCC [IPC07] and use results of regional climate models as well as statistical ensemble approaches for the estimation of the future regional Climate Change in the Upper Danube watershed. The analyses show that the average air temperature at the Upper Danube has already increased by approximately 1,5 °C in the last 30 years.

The IPCC climate scenarios predict a temperature increase between 3,3 °C and 5,2 °C between 1990 and 2090. The trends for the future precipitation are more rainfall in winter, less in summer, altogether per year a precipitation decrease of 3,5 % to 16,4 % is expected.

Consequences of the predicted changes could be a reduction of waterpower production, restrictions for ship traffic in summer due to low water levels, less snow cover per year in lower alpine regions due to temperature increase but possible improvements in high-level alpine regions, which leads to less winter tourism but a moderate increase of summer tourism.

Less private water use is expected (around 20 %) due to changing behaviours and new technologies for saving water. A shortage of drinking water is not expected, but the need for temporary adaptation strategies of water suppliers is likely, like e.g. more cooperation and networks. (Almost) all glaciers in the Upper Danube catchment will vanish until 2045.

[6]http://www.glowa-danube.de/eng/publikationen/publikationen.php



Verbindungsstatus

Verbindungsstatus: Verbunden mit 192.168.1.10, Simulation Sept-KT2_KV1_GMT2

Konfiguration

Simulation

Start:	01.01.2011 00:00
Ende:	01.01.2061 00:00
Gebiet:	Upper Danube Catchment 1 km
Teilgebiet:	whole catchment
Basisdatensatz:	danubia-standard

Szenario

Klimatrend:	REMO regional (KT2)
Klimavariante:	Baseline (KV1)
Gesellschaftsszenario:	Performance (GMT2)
Maßnahme:	

Simulationsstatus

RUNNING Fortschritt: 7 %

Modellstatus

Modellname	Zustand	Modellzeit	Knoten
atmospheredummy-KT2_KV1	→ waitForGet	01.09.2014 00:00	node5
demography-GMT2	← prov	01.09.2014 00:00	node1
economy-GMT2	← prov	01.09.2014 00:00	node1
farmingdummy	→ waitForGet	01.09.2014 00:00	node5
groundwaterflow-KT2_KV1_GMT2	🟩 compute	31.08.2014 00:00	node2
groundwatertransport	→ waitForGet	01.09.2014 00:00	node2
household-GMT2	← prov	01.09.2014 00:00	node3
landsurfacedummy-KT2_KV1	→ waitForGet	01.09.2014 00:00	node5
rivernetworkdummy-KT2_KV1	→ waitForGet	01.09.2014 00:00	node5
tourism-GMT2	🟩 compute	31.08.2014 00:00	node4
trafficdummy	→ waitForGet	01.09.2014 00:00	node5
watersupply-KT2_KV1_GMT2	→ waitForGet	01.09.2014 00:00	node4

Figure 11.8: Main screen of the DANUBIA Monitor

12 Conclusions

This thesis presented the development of a generic framework for integrative environmental modelling and simulation. The framework supports the development and the coupling of simulation models from various disciplines to perform integrative simulations. In particular it allows for the parallel execution of an arbitrary number of dependable simulation models (due to the fact that the models are distributable over a network) which is – to our knowledge – not the case with other simulation frameworks or systems. Any other framework or system considered comparable to ours lacks in at least one of the major characteristics of our approach.

Our framework is generic in the sense, that it is, in principle, applicable to any kind of model which simulates spatially distributed environmental processes on an arbitrary, but discrete time scale. During an integrative simulation for some simulation period, the framework coordinates the coupled models which run in parallel exchanging iteratively data via their interfaces.

For the development of the framework we developed and used a methodology which accommodates best practices of software engineering and takes into account view-based modelling on different abstraction levels – and is in principal applicable for the development of any system where a dispartment of the system into several significant views, i.e. functional aspects, is meaningful. With the help of formal methods the correctness of the temporal coordination (being the heart of the whole framework) could be verified.

With this work a complex system like the Generic Simulation Framework is described in a rigorous manner, including formal specifications making the implementation more reliable. This kind of documentation is valuable for a number of stakeholders like, e.g. natural and social scientists, environmental project executives, model developers, and in particular people dealing with framework development itself.

We identified several advantages of this approach. At first, the view-based development allows for easy replacing (or even omitting) single views, or for easy incorporating additional functionality by simply adding a new view in future development. The framework could so be made applicable for a wide range of problems treated by simulation, even outside the sector of environmental modelling. In particular the use of components which rely only on interface specifications supports further development and re-engineering. Secondly, the clear distinction of abstraction levels facilitates the single stakeholders to extract exactly the information required for their particular purpose. While for project executives it might be sufficient to get a clear picture of the requirements and underlying concepts of the framework, the model developers, and in particular framework developers are certainly interested also in the design which explains how the requirements are met, and the component architecture which shows how the framework is modularised and assembled.

Applying the framework paradigm to integrative environmental modelling provides several advantages: common routines and services like network support, time coordination or space initialisation can be separated from the scientific code of the simulation models. The model developer only has to implement distinguished extension points of the framework, thus one can be sure that generally valid rules are respected by each model. The simulation framework is generic in the sense that it is independent from actual simulation models. In fact, it scales up to an arbitrary number of simulation models and can be applied to any simulation area as long as the requirements of the framework are satisfied.

The framework has been developed and successfully instantiated in the interdisciplinary research project GLOWA-Danube by the implementation of the distributed simulation and decision support system DANUBIA. With DANUBIA a number of scenarios concerning changes of climate and society have been simulated which shall show their impacts on the future of water resources in the Upper Danube watershed thus giving hints for sustainable planning.

During the project it emerged that well-founded methods and techniques of software engineering are essential for the integration of different scientific disciplines, on the technical, as well as on the conceptual side. On the technical side the development and integration of simulation models has been supported by application of the framework technology, and even more, by demanding compliance with common rules and requirements the reliability of the system has been considerably increased. On the conceptual side methods like abstraction, separation of concerns, and – last but not least – the application of the Unified Modeling Language as a common graphical language have accounted for a better and more precise communication among the project partners and enabled the understanding of the integrative aspects of the DANUBIA system. Our experiences in the interdisciplinary project GLOWA-Danube have shown that the impact of informatics on other sciences is of increasing importance. Since many complex problems in natural sciences are approached with computers, software engineers and researchers in natural and social sciences have to work hand in hand.

Of course there are further issues concerning integrative environmental simulation which have not been addressed within this thesis, but give reason for future work to integrate them into the framework. One of the major issues might be the treatment of large amounts of data which typically appear in simulation systems considering large simulation areas. Another question which is widely unanswered arises from the impacts of error propagation and uncertainty within the coupled models on the simulation results. In future versions, also interactive simulation runs are thinkable, which means that the user may interrupt the simulation run at a certain point in time, e.g. when a certain condition becomes true, and resume the run with a modified configuration. This possibility is in particular helpful for the application as decision support system.

The work on the Generic Simulation Framework is not finished with the end of the GLOWA-Danube project, as the framework as well as the simulation models applied in the GLOWA-Danube project have been published under the name OPENDANUBIA under an Open Source Licence. Therewith the framework is accessible for a wide range of interested developers in the open source community, and along with the thesis at hand as a useful documentation for it, it will hopefully also in the future contribute to the solution of environmental problems.

Bibliography

[ACN02] Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava: connecting software architecture to implementation. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 187–197, New York, NY, USA, 2002. ACM.

[Arg04] R. M. Argent. An overview of model integration for environmental applications – components, frameworks and semantics. *Environmental Modelling & Software*, 19:219–234, 2004.

[Bar95] N. Bartelme. *Geoinformatik*. Springer, Berlin, 1995.

[BB07] J. K. F. Bowles and B. Bordbar. A Formal Model for Integrating Multiple Views. *International Conference on Application of Concurrency to System Design*, pages 71–79, 2007.

[BBM08] R. Barthel, J. Braun, and W. Mauser. Integrated modelling of global change effects on the water cycle in the upper Danube catchment (Germany) - the groundwater management perspective. In J. J. Carillo Rivera and M. A. Ortega Guerrero, editors, *Groundwater flow understanding: from local to regional scale. Selected Papers on Hydrogeology SP12*, pages 47–72. Taylor & Francis, 2008.

[BCL$^+$06] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J. Stefani. The FRACTAL component model and its support in Java: Experiences with Auto-adaptive and Reconfigurable Systems. *Softw. Pract. Exper.*, 36(11-12):1257–1284, 2006.

[Beu02] Ottmar Beucher. *MATLAB und Simulink – Grundlegende Einführung*. Pearson Studium, 2002.

[BHH$^+$06] Hubert Baumeister, Florian Hacklinger, Rolf Hennicker, Alexander Knapp, and Martin Wirsing. A Component Model for Architectural Programming. *Electronic Notes in Theoretical Computer Science*, 160:75 – 96, 2006. Proceedings of the International Workshop on Formal Aspects of Component Software (FACS 2005).

[BHKL04] M. Barth, R. Hennicker, A. Kraus, and M. Ludwig. DANUBIA: An Integrative Simulation System for Global Change Research in the Upper Danube Basin. *Cybernetics and Systems*, 35(7-8):639–666, 2004.

[BHP06] T. Bures, P. Hnetynka, and F. Plasil. SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model. In *Software Engineering Research, Management and Applications, 2006. Fourth International Conference on*, pages 40 –48, 2006.

[BJN+10] Roland Barthel, Stephan Janisch, Darla Nickel, Aleksandar Trifkovic, and Thomas Hörhan. Using the Multiactor-Approach in GLOWA-Danube to Simulate Decisions for the Water Supply Sector Under Conditions of Global Climate Change. *Water Resour Manag*, 24(2):239–275, Jan 2010.

[BJS+08] R. Barthel, S. Janisch, N. Schwarz, A. Trifkovic, D. Nickel, C. Schulz, and W. Mauser. An integrated modelling framework for simulating regional-scale actor responses to global change in the water domain. *Environmental Modelling & Software*, 23(9):1095 – 1121, 2008.

[BK04] M. Barth and A. Knapp. A Coordination Architecture for Time-Dependent Components. In M. H. Hamza, editor, *Proc. 22nd Int. Multi-Conf. Applied Informatics. Software Engineering (IASTED SE'04)*, pages 6–11. ACTA Press, 2004.

[BME+10] R. Barthel, W. Mauser, A. Ernst, S. Dabbert, J. Schmude, J. Wackerbauer, K. Schneider, and R. Ziller. Integrative Analyse der sozioökonomischen Auswirkungen des Globalen Wandels auf Wasser- und Landnutzung im Einzugsgebiet der Oberen Donau – Abschließende Ergebnisse des GLOWA-Danube-Projekts. In Levin et al. [LGK+10], page 125 ff. ISBN 978-3-510-49213-7.

[BMS+10] R. Barthel, W. Mauser, K. Schneider, A. Gundel, R. Ziller, and D. Bendel. Auswirkungen des Klimawandels auf Wasserhaushalt, Grundwasserneubildung, Grundwasserstände und Grundwasserqualität im Einzugsgebiet der Oberen Donau - Abschließende Ergebnisse des GLOWA-Danube-Projekts. In Levin et al. [LGK+10], page 107 ff. ISBN 978-3-510-49213-7.

[Bow06] Juliana Bowles. Decomposing Interactions. In Michael Johnson and Varmo Vene, editors, *Algebraic Methodology and Software Technology*, volume 4019 of *Lecture Notes in Computer Science*, pages 189–203. Springer Berlin / Heidelberg, 2006.

[Bra06] A. Brack. Verwaltung der Landnutzung in DANUBIA – der Unified Process angewendet. Master's thesis, Institut für Informatik, Ludwig-Maximilians-Universität München, 2006.

[CKM09] María Victoria Cengarle, Alexander Knapp, and Heribert Mühlberger. Interactions. In Lano [Lan09], chapter 9, pages 205–248.

[DW99] D. D'Souza and A. Wills. *Objects, Components and Frameworks with UML – The Catalysis Approach.* Addison-Wesley, Reading, Massachusetts, 1999.

[EEEP08] Hartmut Ehrig, Karsten Ehrig, Claudia Ermel, and Ulrike Prange. Consistent integration of models based on views of visual languages. In *Proceedings of the Theory and practice of software, 11th international conference on Fundamental approaches to software engineering*, FASE'08/ETAPS'08, pages 62–76, Berlin, Heidelberg, 2008. Springer-Verlag.

[End03] M. B. Endejan. *Entwicklung einer Software-Architektur für Systeme zum integrierten simulationsbasierten Assessment des globalen Wandels.* kassel university press, Kassel, 2003. Zugl.: Kassel, Univ., Diss. 2003.

[ESSJ07] Andreas Ernst, Carsten Schulz, Nina Schwarz, and Stephan Janisch. Modeling of Water Use Decisions in a Large, Spatially Explicit Coupled Simulation System. In *Social Simulation: Technologies, Advances and New Discoveries.* Idea Group Inc., 2007.

[Eur00] European Union. Directive 2000/60/EC of the European Parliament and of the Council of 23 October 2000 establishing a framework for Community action in the field of water policy. *Official Journal*, L 327:1–73, 2000.

[Fis97] Clemens Fischer. CSP-OZ: A Combination of Object-Z and CSP. Technical report, University of Oldenburg, 1997.

[FM04] Stephan Flake and Wolfgang Müller. Formal Semantics of OCL Messages. *Electr. Notes Theor. Comput. Sci.*, 102:77–97, 2004.

[Gar01] Jose M. Garrido. *Object-Oriented Discrete-Event Simulation with Java: A Practical Introduction.* Kluwer Academic Publishers, Norwell, MA, USA, 2001.

[GGW07] J. B. Gregersen, P. J. A. Gijsbers, and S. J. P. Westen. OpenMI: Open modelling interface. *Journal of Hydroinformatics*, 9(3):175–191, 2007.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, 1995.

[GJKH06] C. Giupponi, A. J. Jakeman, D. Karssenberg, and M. P. Hare, editors. *Sustainable Management of Water Resources – An Integrated Approach.* Edward Elgar Publishing, Cheltenham, UK, 2006.

[GLO10] GLOWA-Danube Projekt, editor. *Global Change Atlas – Einzugsgebiet Obere Donau*. Ludwig-Maximilians-Universität München, 2010. ISBN 978-3-00-026548-8.

[Gro01] William Grosso. *Java RMI*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1^{st} edition, 2001.

[HBJL09] Rolf Hennicker, Sebastian S. Bauer, Stephan Janisch, and Matthias Ludwig. Principles of Integrative Environmental Simulations. In *Proc. European Computer Science Summit (ECSS 09)*, Paris, France, 2009.

[HBJL10] Rolf Hennicker, Sebastian S. Bauer, Stephan Janisch, and Matthias Ludwig. A Generic Framework for Multi-Disciplinary Environmental Modelling. In Swayne et al. [SYV+10].

[HBvEL03] C. Hillyer, J. Bolte, F. van Evert, and A. Lamaker. The ModCom modular simulation system. *European Journal of Agronomy*, 18(3):333–343(11), January 2003.

[HH06] Oddleif Halvorsen and Oystein Haugen. Proposed Notation for Exception Handling in UML 2 Sequence Diagrams. *Software Engineering Conference, Australian*, 0:29–40, 2006.

[HHRS05] Øystein Haugen, Knut Husa, Ragnhild Runde, and Ketil Stølen. STAIRS towards formal design with sequence diagrams. *Software and Systems Modeling*, 4:355–357, 2005. 10.1007/s10270-005-0087-0.

[HJL10] Rolf Hennicker, Stephan Janisch, and Matthias Ludwig. Agent-Based Social Simulation within a Generic Framework for Environmental Modelling. In *Proc. WCSS 2010, 3rd World Congress on Social Simulation*, Kassel, Germany, 2010. Center for Environmental Systems Research, University of Kassel.

[HKKR05] M. Hitz, G. Kappel, E. Kapsamer, and W. Retschnitzegger. *UML@Work – Objektorientierte Modellierung mit UML 2 (3. Auflage)*. dpunkt.verlag, Heidelberg, 2005.

[HL05] R. Hennicker and M. Ludwig. Property-Driven Development of a Coordination Model for Distributed Simulations. In *Formal Methods for Open Object-Based Distributed Systems, 7th IFIP WG 6.1 International Conference, FMOODS 2005, Athens, Greece*, volume 3535 of *Lecture Notes in Computer Science*, pages 290–305. Springer, 2005.

[HL06] R. Hennicker and M. Ludwig. Design and Implementation of a Coordination Model for Distributed Simulations. In H. C. Mayr and R. Breu, editors, *Proc. Modellierung 2006 (MOD'06)*, volume P-82 of *Lect. Notes Informatics*, pages 83–97. Gesellschaft für Informatik, 2006.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[Hol04] G. Holzmann. *The SPIN Model Checker – Primer and Reference Manual.* Addison-Wesley, 2004.

[HS03] Øystein Haugen and Ketil Stølen. STAIRS - Steps to Analyze Interactions with Refinement Semantics. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *UML*, volume 2863 of *Lecture Notes in Computer Science*, pages 388–402. Springer, 2003.

[IEE00] IEEE. IEEE Std 1471:2000 Recommended Practice For Architectural Description of Software-Intensive Systems. Technical report, IEEE, 2000.

[IPC07] IPCC. Summary for Policymakers. In S. Solomon, D. Qin, M. Manning, Z. Chen, M. Marquis, K.B. Averyt, M.Tignor, and H.L. Miller, editors, *Climate Change 2007: The Physical Science Basis. Contribution of Working Group I to the Fourth Assessment Report of the Intergovernmental Panel on Climate Change*. Cambridge University Press, Cambridge, United Kingdom and New York, NY, USA, 2007.

[ITU96] ITU-TS. ITU-TS Recommendation Z.120: Message Sequence Chart (MSC). Technical report, ITU-TS, Geneva, 1996.

[Jag10] H. R. A. Jagers. Linking Data, Models and Tools: An Overview. In Swayne et al. [SYV$^+$10].

[JBR05] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Modeling Language User Guide*. The Addison-Wesley Object Technology Series. Addison-Wesley, 2nd edition, 2005.

[KCH04] Jacques Klein, Benoit Caillaud, and Loic Hélouet. Merging Scenarios. In *9th International Workshop on Formal Methods for Industrial Critical Systems (FMICS)*, pages 209–226, Linz, Austria, sep 2004.

[KD95] H. Kächele and S. Dabbert. Szenariotechnik. In H.-R. Bork, C. Dalchow, H. Kächele, H.-P. Piorr, and O. Wenkel, editors, *Agrarlandschaftswandel in Nordost-Deutschland unter veränderten Rahmenbedingungen: ökologische und ökonomische Konsequenzen*, pages 39–47. Ernst & Sohn, Berlin, 1995.

[KFJ07] Jacques Klein, Franck Fleurey, and Jean-Marc Jézéquel. Weaving Multiple Aspects in Sequence Diagrams. In Awais Rashid and Mehmet Aksit, editors, *Transactions on Aspect-Oriented Software Development III*, volume 4620 of *Lecture Notes in Computer Science*, pages 167–199. Springer Berlin / Heidelberg, 2007.

[KKO05] S. Kralisch, P. Krause, and O.David. Using the Object Modeling System for hydrological model development and application. *Advances in Geosciences*, 4:75–81, 2005.

[Kou09] Anna Kourtessis. Integrative Umweltsimulationssysteme und -frameworks: Eine vergleichende Studie zum Stand der Technik. Fortgeschrittenenpraktikum, Institut für Informatik, Ludwig-Maximilians-Universität München, 2009.

[Kra07] A. Kraus. *Model Driven Software Engineering for Web Applications*. PhD thesis, Institut für Informatik, Ludwig-Maximilians-Universität München, 2007.

[Lan09] Kevin Lano, editor. *UML 2 Semantics and Applications*. John Wiley & Sons, 2009.

[LB06] Rebecca A. Letcher and John Bromley. Typology of Models and Methods of Integration. In Giupponi et al. [GJKH06], chapter 11, pages 287–323.

[LGK$^+$10] C. Levin, P. Grathwohl, A. Kappler, R. Kaufmann-Knoke, M. Lodemann, and H. Rügner, editors. *Grundwasser für die Zukunft*, volume 67 of *Schriftenreihe der Deutschen Gesellschaft für Geowissenschaften*, 2010. ISBN 978-3-510-49213-7.

[LMN$^+$03] R. Ludwig, W. Mauser, S. Niemeyer, A. Colgan, R. Stolz, H. Escher-Vetter, M. Kuhn, M. Reichstein, J. Tenhunen, A. Kraus, M. Ludwig, M. Barth, and R. Hennicker. Web-based Modeling of Water, Energy and Matter Fluxes to Support Decision Making in Mesoscale Catchments – the Integrative Perspective of GLOWA-Danube. *Physics and Chemistry of the Earth*, 28:621–634, 2003.

[LTS11] Labelled Transition System Analyser. `http://www.doc.ic.ac.uk/ltsa/`, January 2011.

[LWKS10] V.I.S. Lenz-Wiedemann, C.W. Klar, and K. Schneider. Development and test of a crop growth model for application within a Global Change decision support system. *Ecol Model*, 221(2):314–329, 2010.

[MB09] Sun Meng and Luís S. Barbosa. Co-Algebraic Semantic Framework for Reasoning about Interaction Designs. In Lano [Lan09].

[MBD00] Ralph Miarka, Eerke Boiten, and John Derrick. Guards, preconditions, and refinement in z. In *ZB 2000: Formal Specification and Development in Z and B*, volume 1878 of *Lecture Notes in Computer Science*, pages 286–303. Springer Berlin / Heidelberg, 2000. 10.1007/3-540-44525-0_17.

[MK06] J. Magee and J. Kramer. *Concurrency : state models & Java programming*. Wiley, Chichester, 2nd edition, 2006.

[NW04] Steve Northover and Mike Wilson. *SWT: The Standard Widget Toolkit*, volume 1. Addison-Wesley Professional, 1st edition, 2004.

[Obj09] Object Management Group. *OMG Unified Modeling Language (OMG UML), Superstructure, V2.2*, 2009.

[PH94] B. Page and L. M. Hilty, editors. *Umweltinformatik*. Oldenbourg, München, 1994.

[RA06] Andrea E. Rizzoli and Robert M. Argent. Software and Software Systems: Platforms and Issues for IWRM Problems. In Giupponi et al. [GJKH06], chapter 12, pages 324–346.

[Rie04] Martin Rieland. Das BMBF-Programm GLOWA: Instrumente für ein vorausschauendes Management großer Flusseinzugsgebiete. *Hydrologie und Wasserbewirtschaftung*, 48(2):83–84, 2004.

[RJB05] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Pearson Education, Inc, 2nd edition, 2005.

[RSP^{+}03] J. M. Rahman, S. P. Seaton, J-M. Perraud, H. Hotham, D. I. Verrelli, and J. R. Coleman. It's TIME for a New Environmental Modelling Framework. In *Proceedings of MODSIM 2003 International Congress on Modelling and Simulation*, volume 4, Townsville, Australia, July 2003. Modelling and Simulation Society of Australia and New Zealand Inc.

[Rum04] B. Rumpe. *Modellierung mit UML – Sprache, Konzepte und Methodik*. Springer, Berlin, 2004.

[SE09] Nina Schwarz and Andreas Ernst. Agent-based modeling of the diffusion of environmental innovations – An empirical approach. *Technol Forecast Soc*, 76(4):497–511, 2009.

[SYV^{+}10] David A. Swayne, Wanhong Yang, Alexey A. Voinov, Andrea Rizzoli, and Tatiana Filatova, editors. *Proceedings of the iEMSs Fifth Biennial Meeting: International Congress on Environmental Modelling and Software (iEMSs 2010)*, Ottawa, Canada, July 2010. International Environmental Modelling and Software Society.

[TK99] J. D. Tenhunen and P. Kabat, editors. *Integrating Hydrology, Ecosystem Dynamics, and Biogeochemistry in Complex Landscapes*. Wiley, Chichester, 1999.

[Wag07] Susanne Wagner. Specification and Tool-Based Analysis of Danubia Simulation Conifgurations. Master's thesis, Institut für Informatik, Ludwig-Maximilians-Universität München, 2007.

[WK03] J. Warmer and A. Kleppe. *The Object Constraint Language*. Addison-Wesley, 2nd edition, 2003.

[WN95] Glynn Winskel and Mogens Nielsen. Models for concurrency. In S. Abramsky, Dov M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of logic in computer science*, volume 4, pages 1–148. Oxford University Press, Oxford, UK, 1995.

Appendix

A

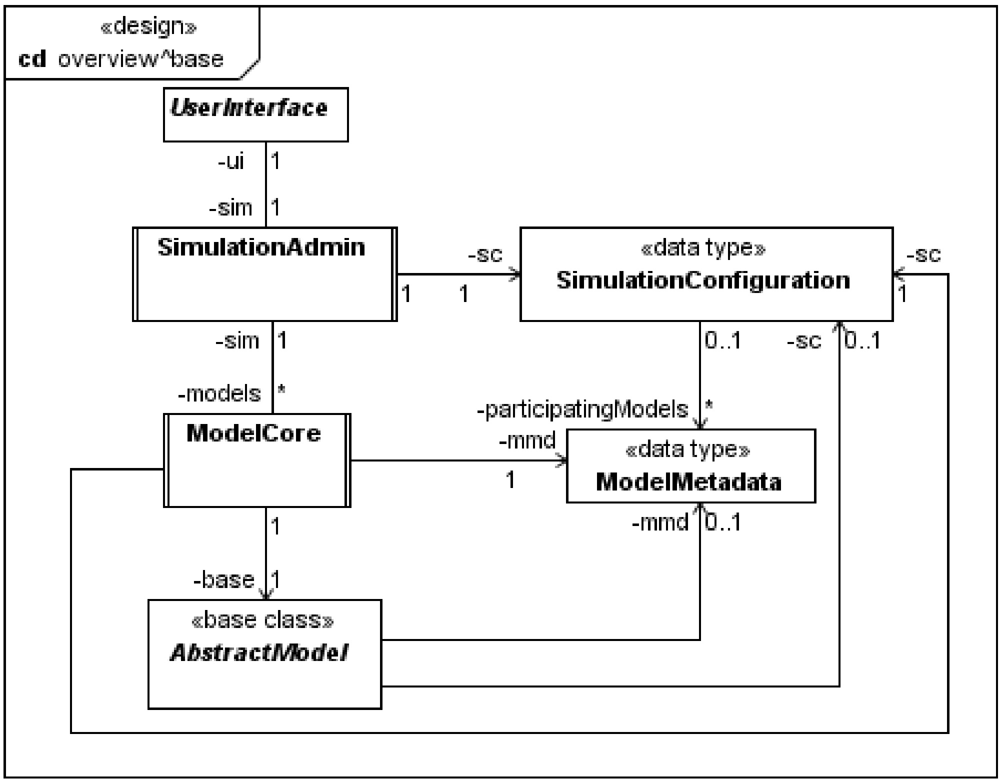
Detailed Framework Model

In this appendix we provide the detailed model of the Generic Simulation Framework developed in Chapters 4 to 8.

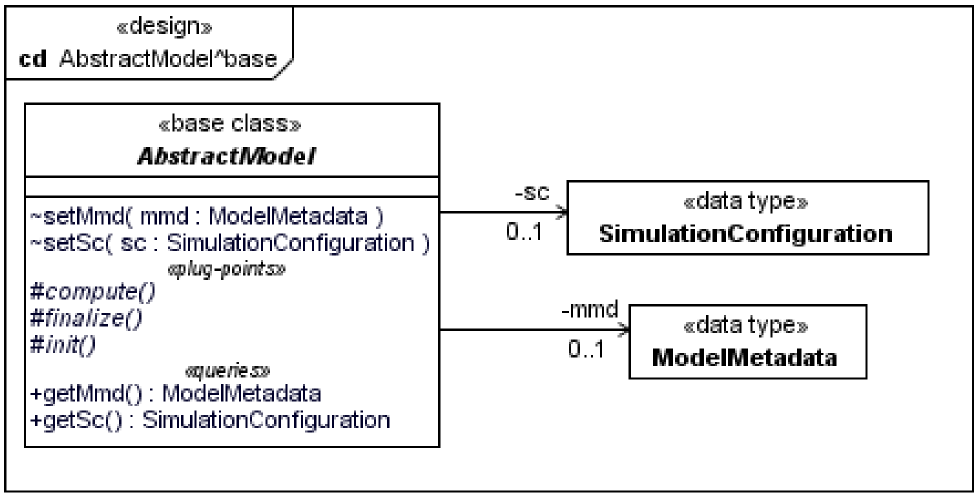
Preliminary note. If dots in the operation compartment of a class symbol indicate an elision, then the non-standard constructor of the respective class is omitted for lack of space. In this case the parameter list of the constructor comprises an entry for each property of the class.

A.1 Base View

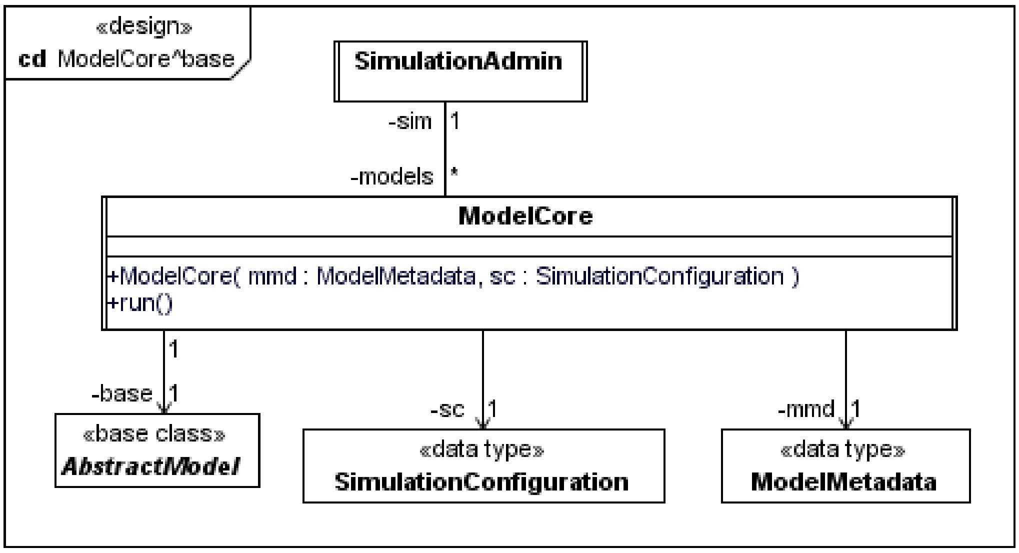
A.1.1 Structural Design



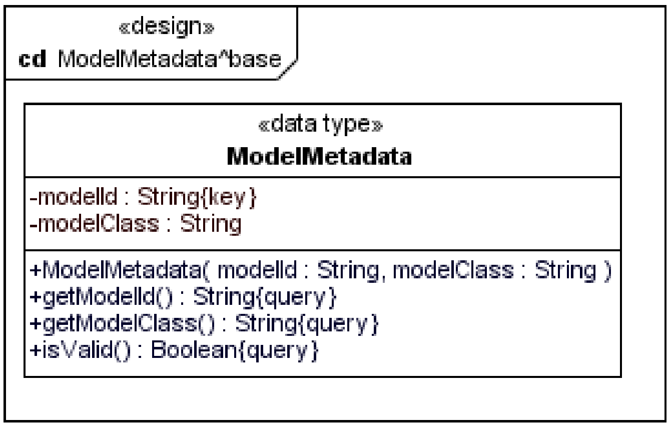
AbstractModel



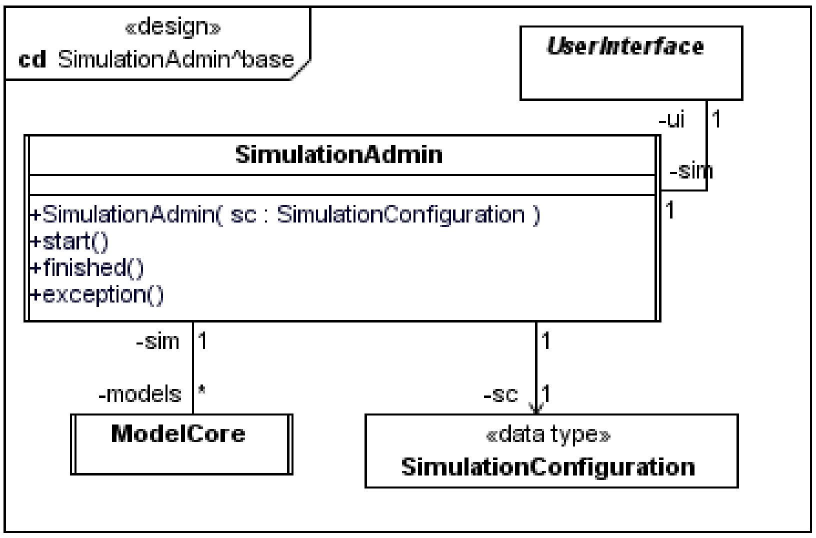
ModelCore



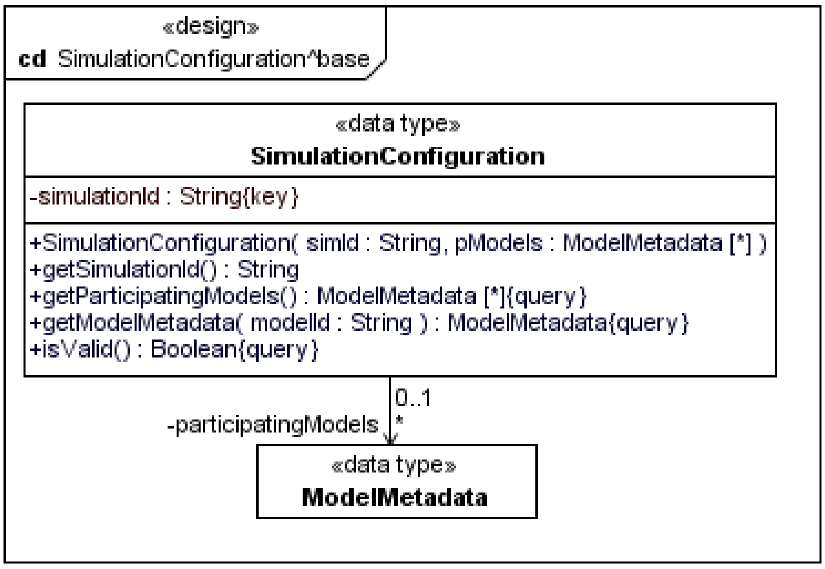
ModelMetadata


«design»
cd ModelMetadata^base
«data type»
ModelMetadata
-modelId : String{key}
-modelClass : String
+ModelMetadata(modelId : String, modelClass : String)
+getModelId() : String{query}
+getModelClass() : String{query}
+isValid() : Boolean{query}

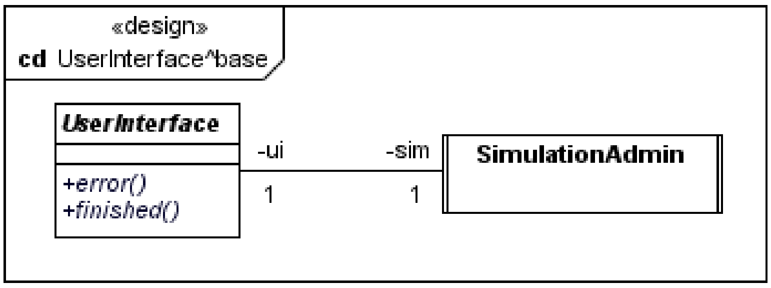
SimulationAdmin



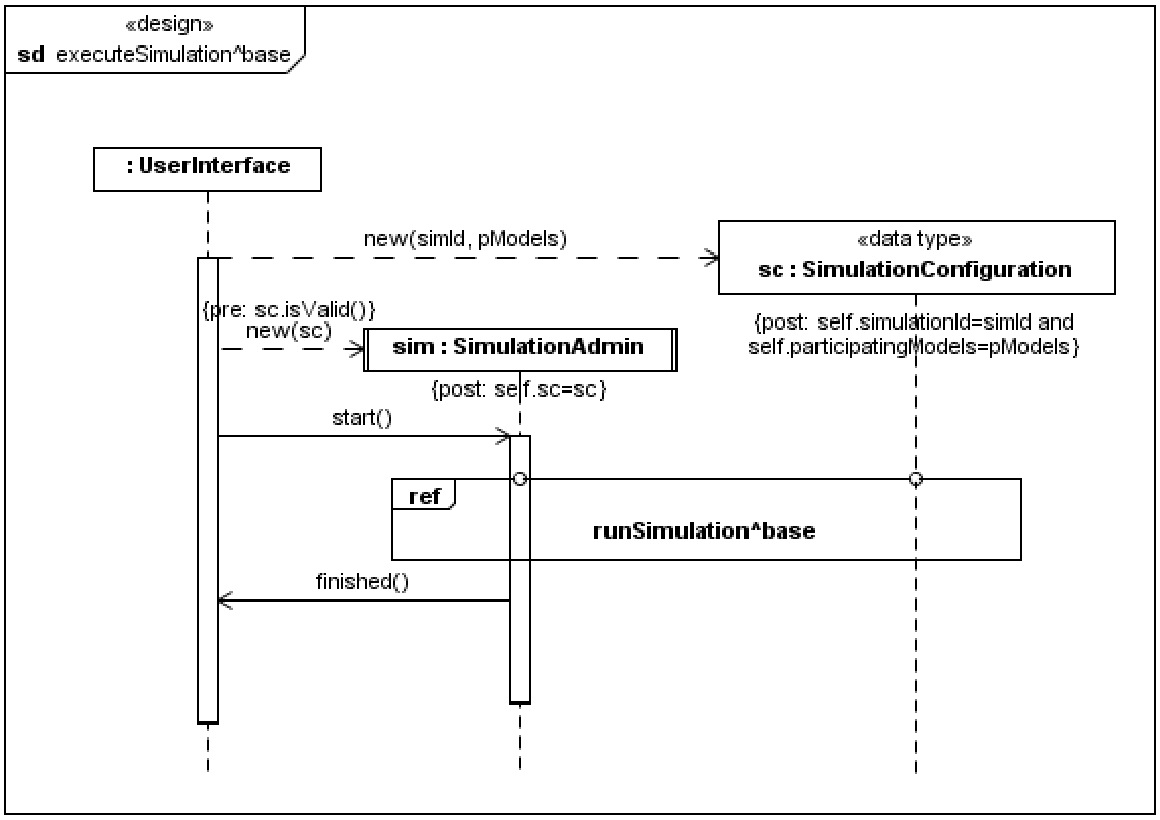
SimulationConfiguration

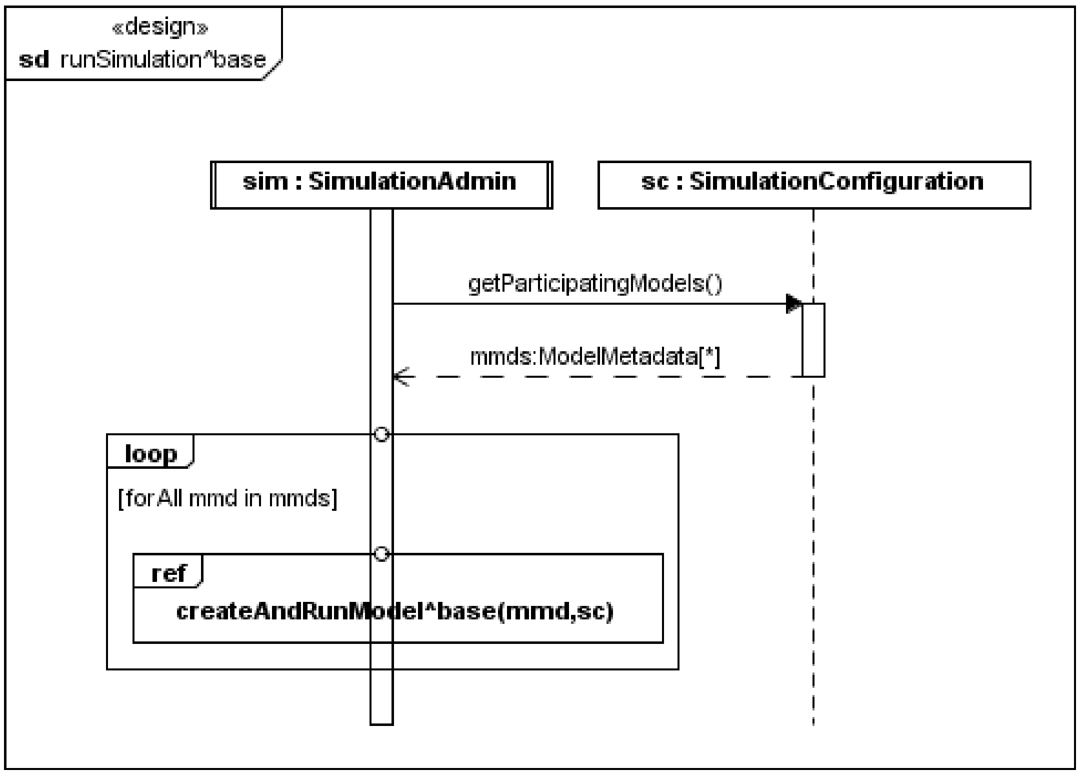

«design»
cd SimulationConfiguration^base
«data type»
SimulationConfiguration
-simulationId : String{key}
+SimulationConfiguration(simId : String, pModels : ModelMetadata [*])
+getSimulationId() : String
+getParticipatingModels() : ModelMetadata [*]{query}
+getModelMetadata(modelId : String) : ModelMetadata{query}
+isValid() : Boolean{query}
0..1
-participatingModels *
«data type»
ModelMetadata

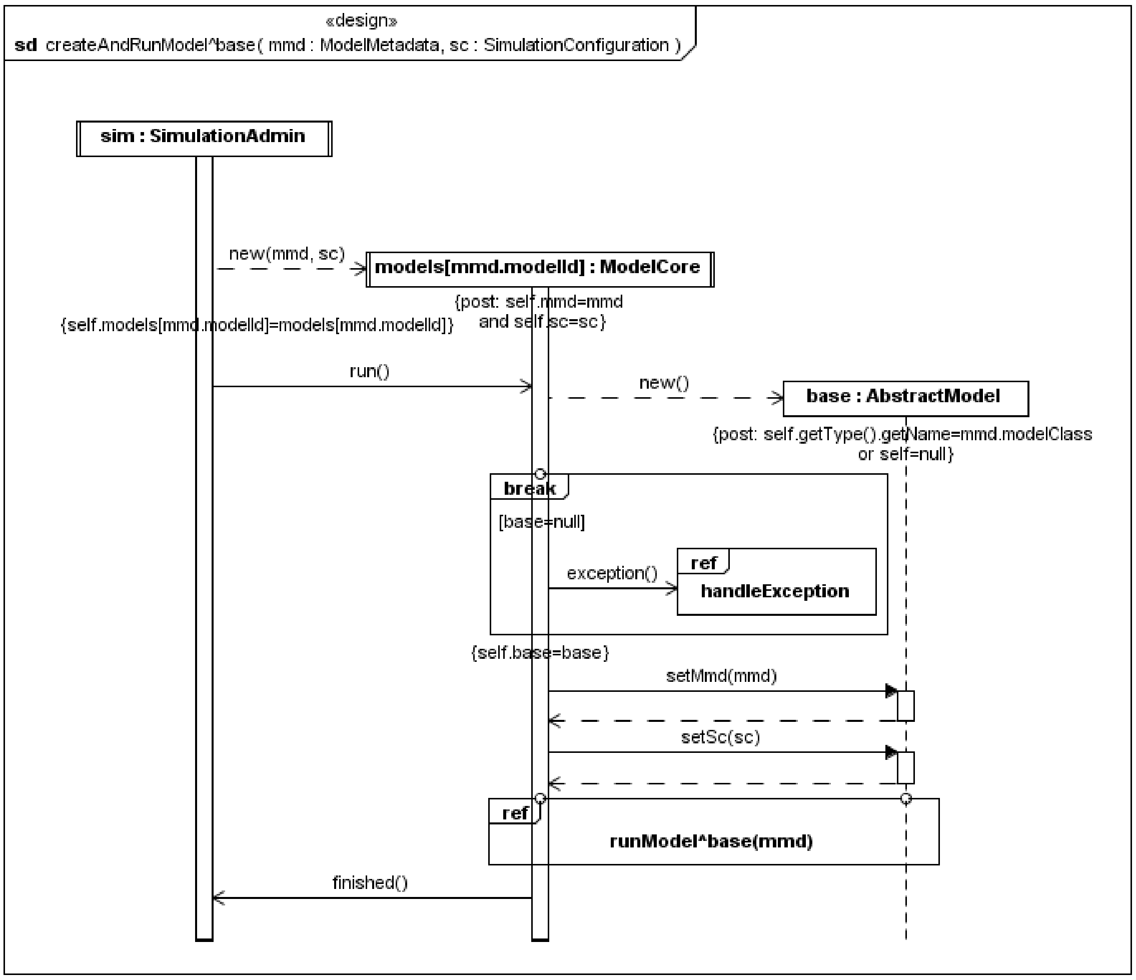
UserInterface

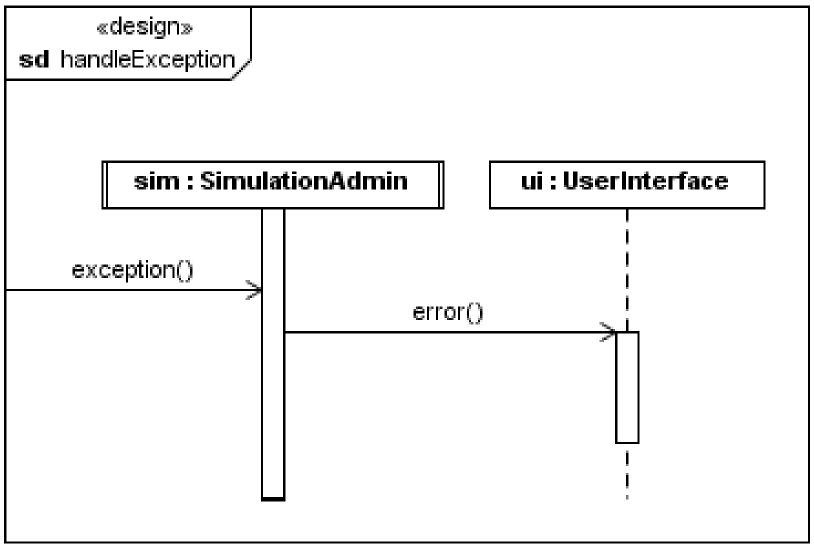


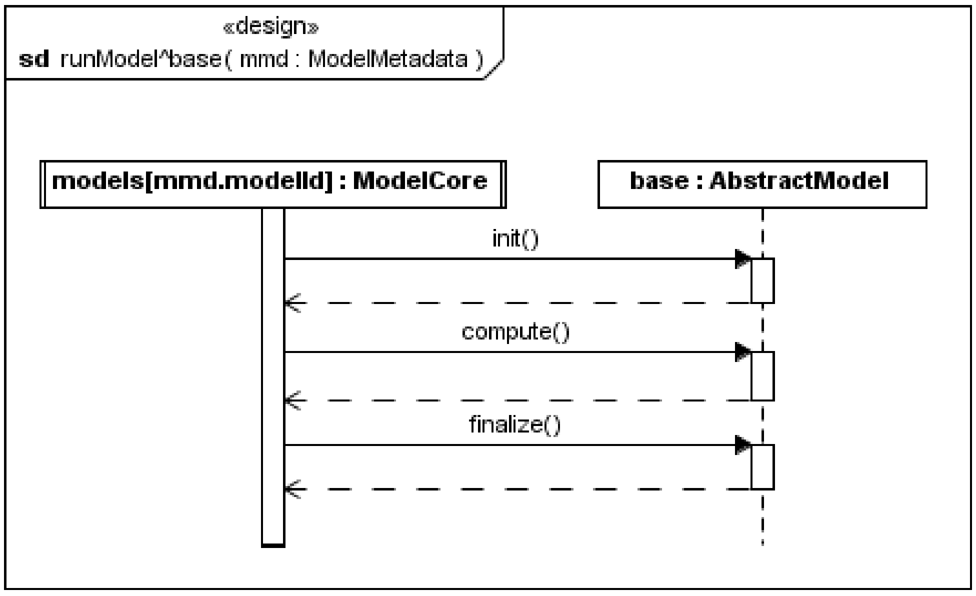
A.1.2 Behavioural Design



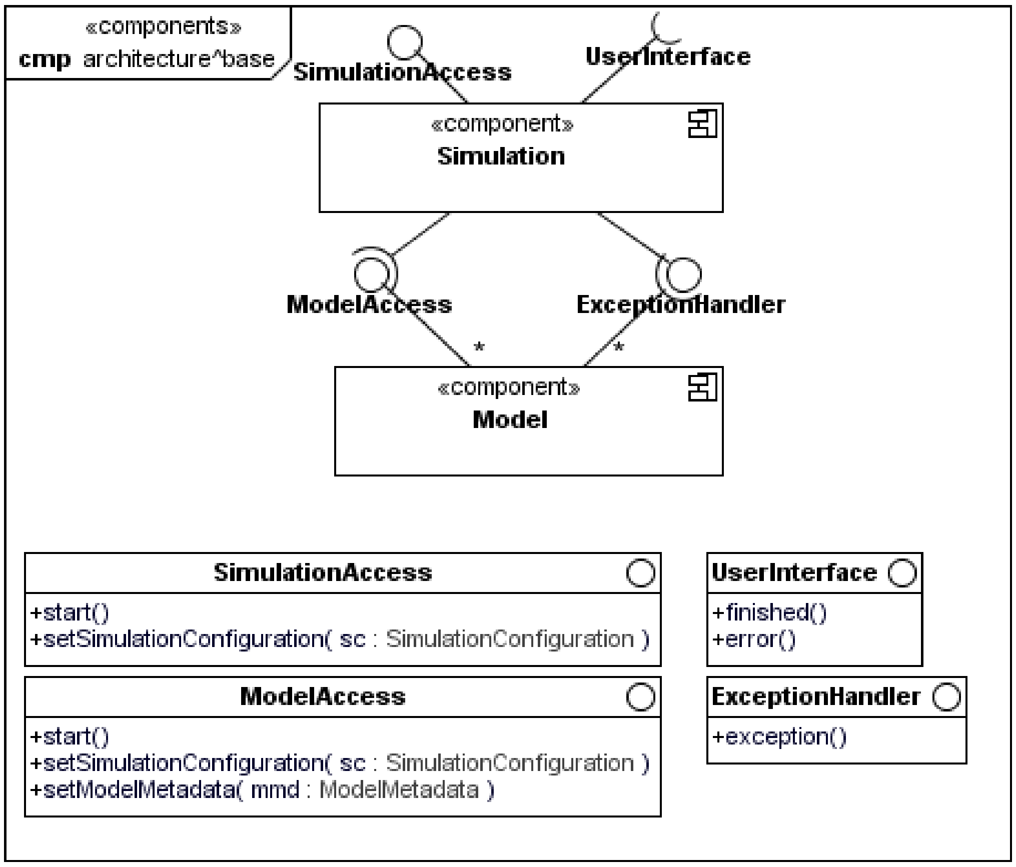

«design»
sd runSimulation^base
sim : SimulationAdmin
sc : SimulationConfiguration
getParticipatingModels()
mmds:ModelMetadata[*]
loop
[forAll mmd in mmds]
ref
createAndRunModel^base(mmd,sc)

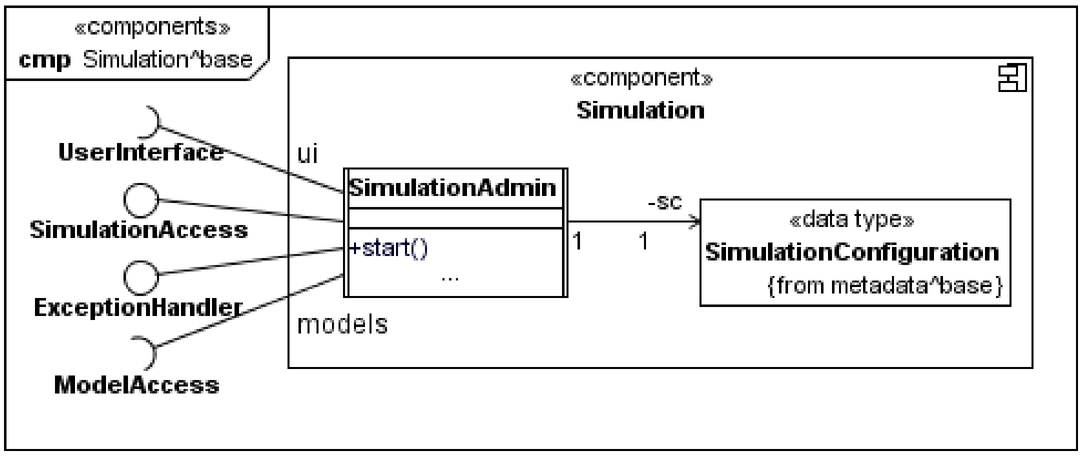

«design»
sd createAndRunModel^base(mmd : ModelMetadata, sc : SimulationConfiguration)
sim : SimulationAdmin
new(mmd, sc)
models[mmd.modelId] : ModelCore
{post: self.mmd=mmd and self.sc=sc}
{self.models[mmd.modelId]=models[mmd.modelId]}
run()
new()
base : AbstractModel
{post: self.getType().getName=mmd.modelClass or self=null}
break
[base=null]
exception()
ref
handleException
{self.base=base}
setMmd(mmd)
setSc(sc)
ref
runModel^base(mmd)
finished()

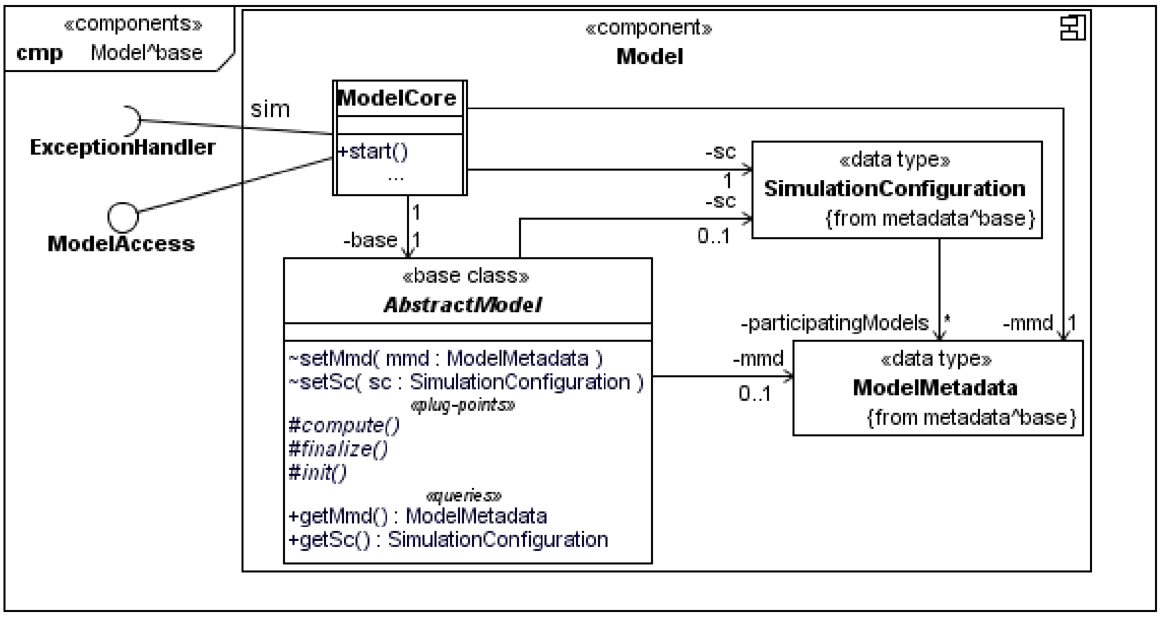

«design»
sd handleException
sim : SimulationAdmin
ui : UserInterface
exception()
error()

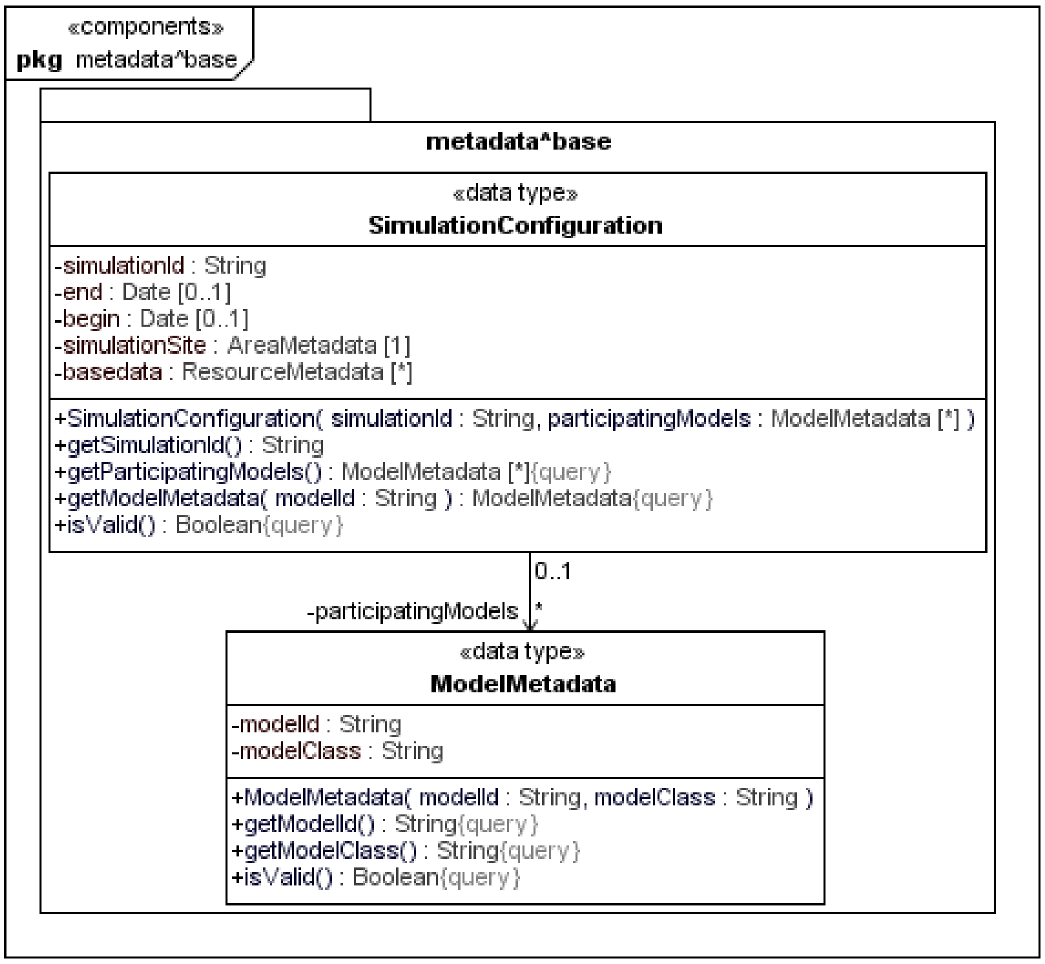

«design»
sd runModel^base(mmd : ModelMetadata)
models[mmd.modelId] : ModelCore
base : AbstractModel
init()
compute()
finalize()

A.1.3 Components



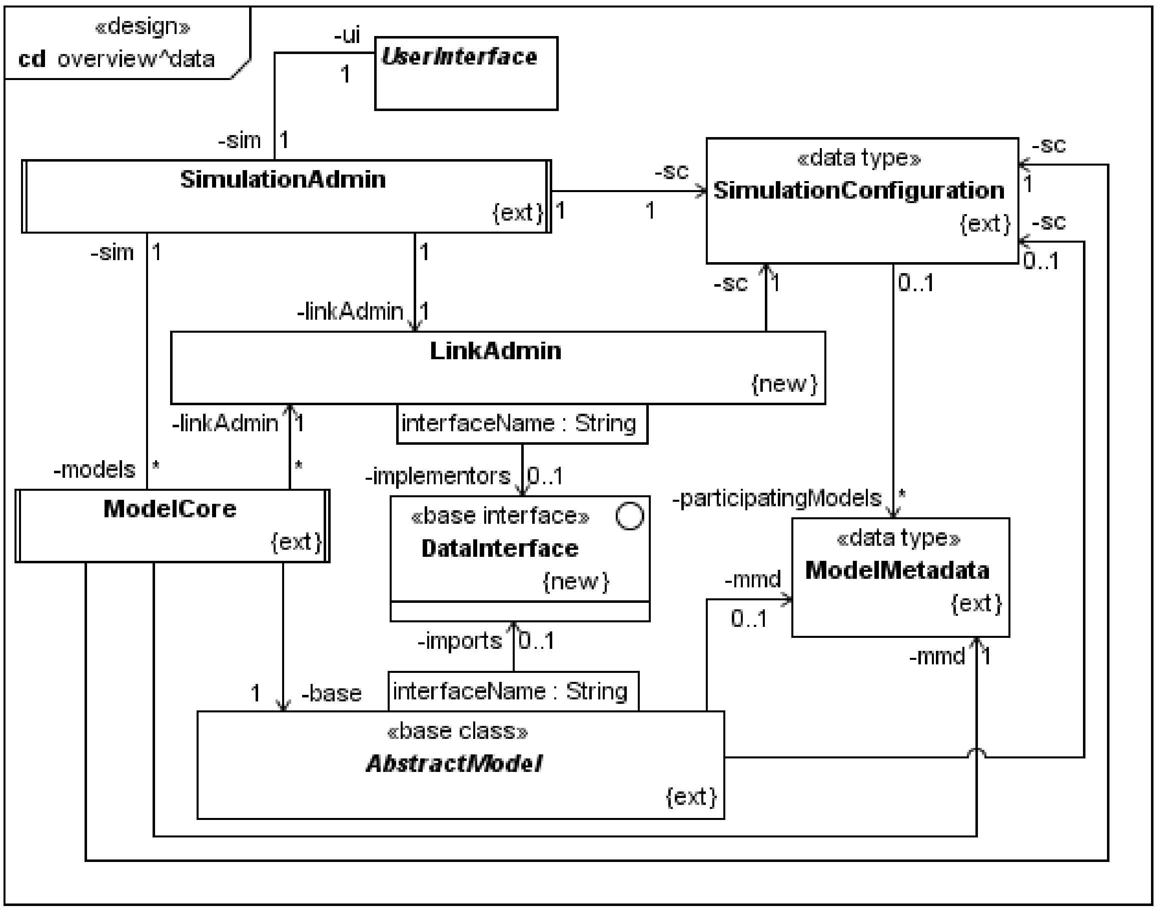

«components»
cmp Simulation^base
«component»
Simulation
UserInterface
SimulationAccess
ExceptionHandler
ModelAccess
ui
models
SimulationAdmin
+start()
...
-sc
1
1
«data type»
SimulationConfiguration
{from metadata^base}


«components»
cmp Model^base
«component»
Model
ExceptionHandler
ModelAccess
sim
ModelCore
+start()
...
-sc
1
-sc
0..1
«data type»
SimulationConfiguration
{from metadata^base}
-base
1
1
«base class»
AbstractModel
~setMmd(mmd : ModelMetadata)
~setSc(sc : SimulationConfiguration)
«plug-points»
#compute()
#finalize()
#init()
«queries»
+getMmd() : ModelMetadata
+getSc() : SimulationConfiguration
-participatingModels
*
-mmd
1
-mmd
0..1
«data type»
ModelMetadata
{from metadata^base}

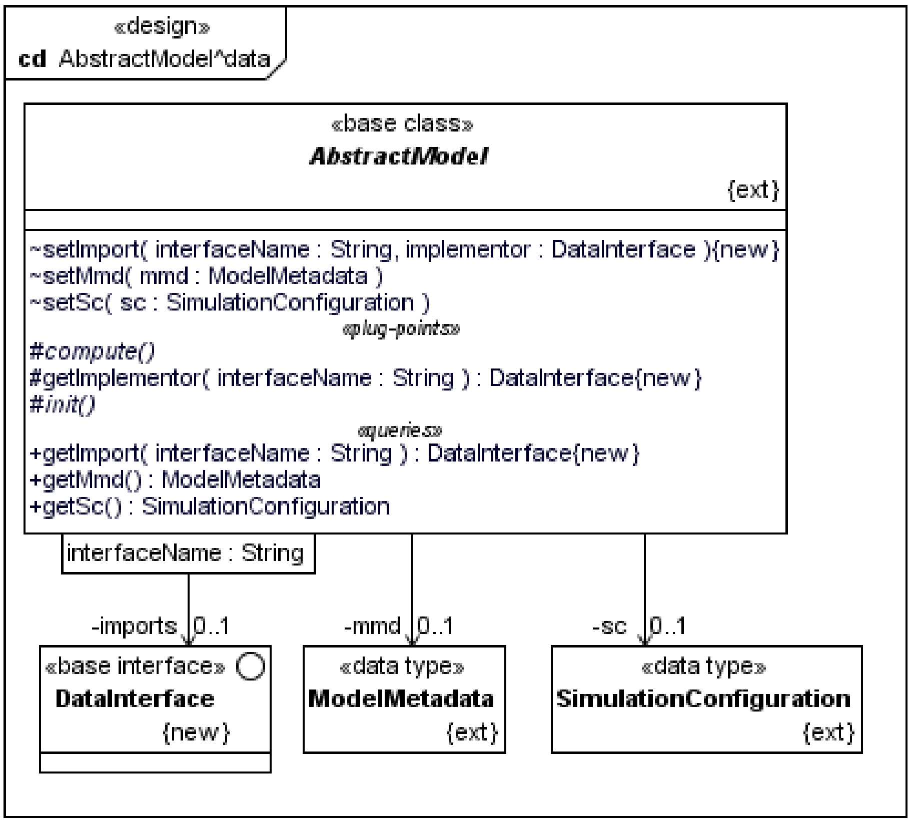

«components»
pkg metadata^base
metadata^base
«data type»
SimulationConfiguration
-simulationId : String
-end : Date [0..1]
-begin : Date [0..1]
-simulationSite : AreaMetadata [1]
-basedata : ResourceMetadata [*]
+SimulationConfiguration(simulationId : String, participatingModels : ModelMetadata [*])
+getSimulationId() : String
+getParticipatingModels() : ModelMetadata [*]{query}
+getModelMetadata(modelId : String) : ModelMetadata{query}
+isValid() : Boolean{query}
0..1
-participatingModels
*
«data type»
ModelMetadata
-modelId : String
-modelClass : String
+ModelMetadata(modelId : String, modelClass : String)
+getModelId() : String{query}
+getModelClass() : String{query}
+isValid() : Boolean{query}

A.2 View "Data Exchange"

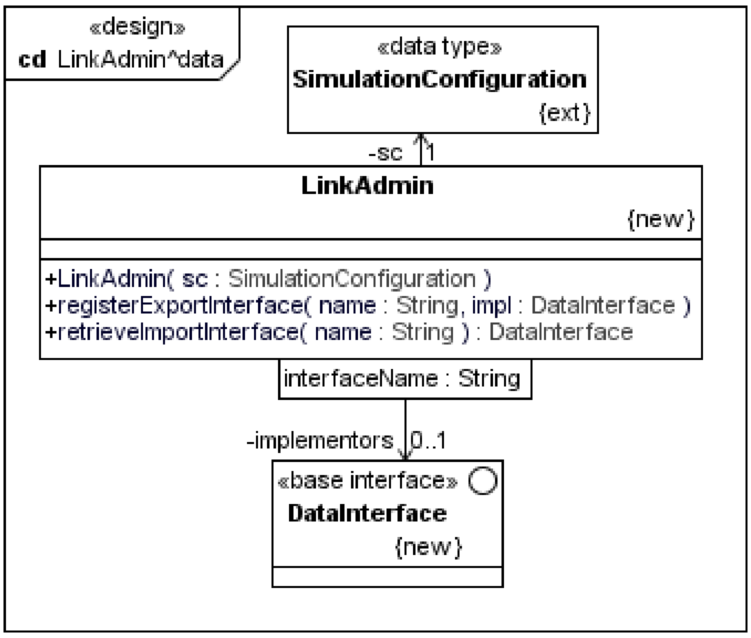
A.2.1 Structural Design



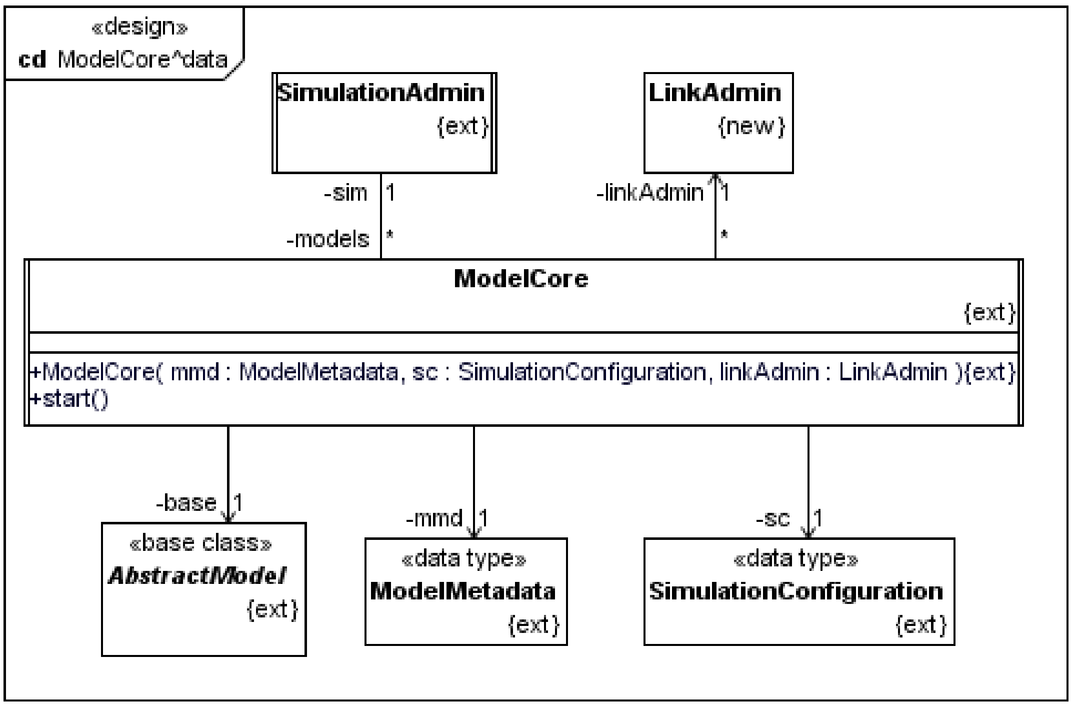
AbstractModel


«design»
cd AbstractModel^data
«base class»
AbstractModel
{ext}
~setImport(interfaceName : String, implementor : DataInterface){new}
~setMmd(mmd : ModelMetadata)
~setSc(sc : SimulationConfiguration)
«plug-points»
#compute()
#getImplementor(interfaceName : String) : DataInterface{new}
#init()
«queries»
+getImport(interfaceName : String) : DataInterface{new}
+getMmd() : ModelMetadata
+getSc() : SimulationConfiguration
interfaceName : String
-imports 0..1
-mmd 0..1
-sc 0..1
«base interface»
DataInterface
{new}
«data type»
ModelMetadata
{ext}
«data type»
SimulationConfiguration
{ext}

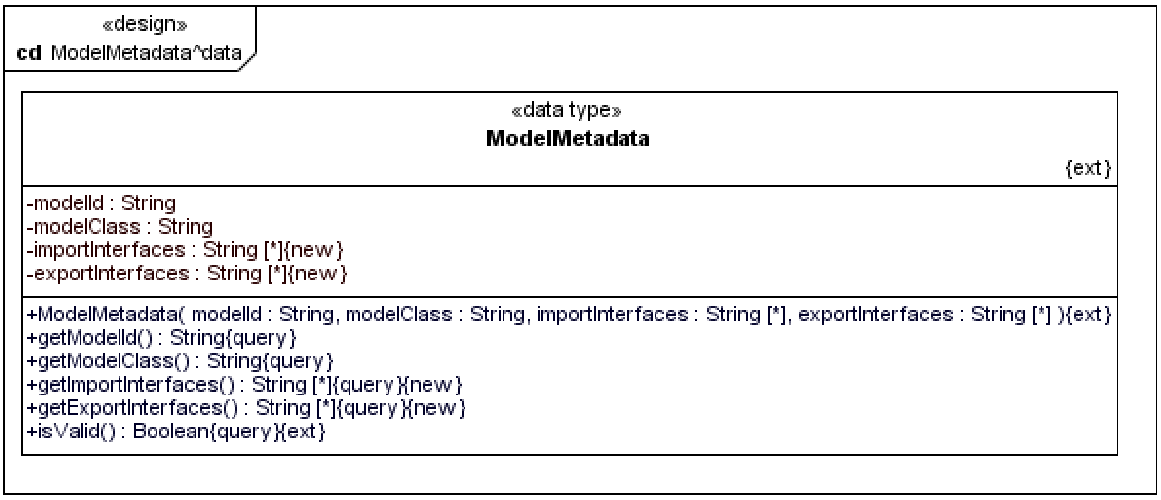
LinkAdmin


«design»
cd LinkAdmin^data
«data type»
SimulationConfiguration
{ext}
-sc
1
LinkAdmin
{new}
+LinkAdmin(sc : SimulationConfiguration)
+registerExportInterface(name : String, impl : DataInterface)
+retrieveImportInterface(name : String) : DataInterface
interfaceName : String
-implementors
0..1
«base interface»
DataInterface
{new}

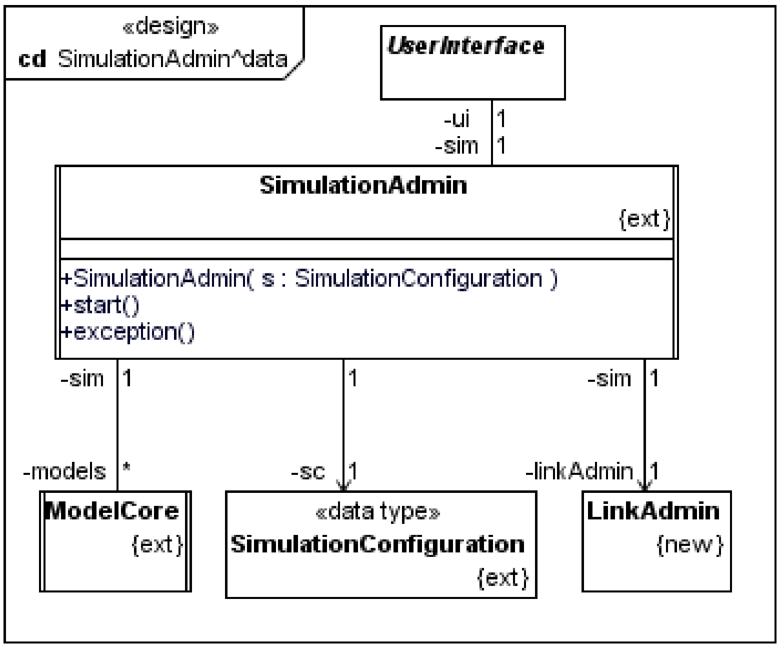
ModelCore



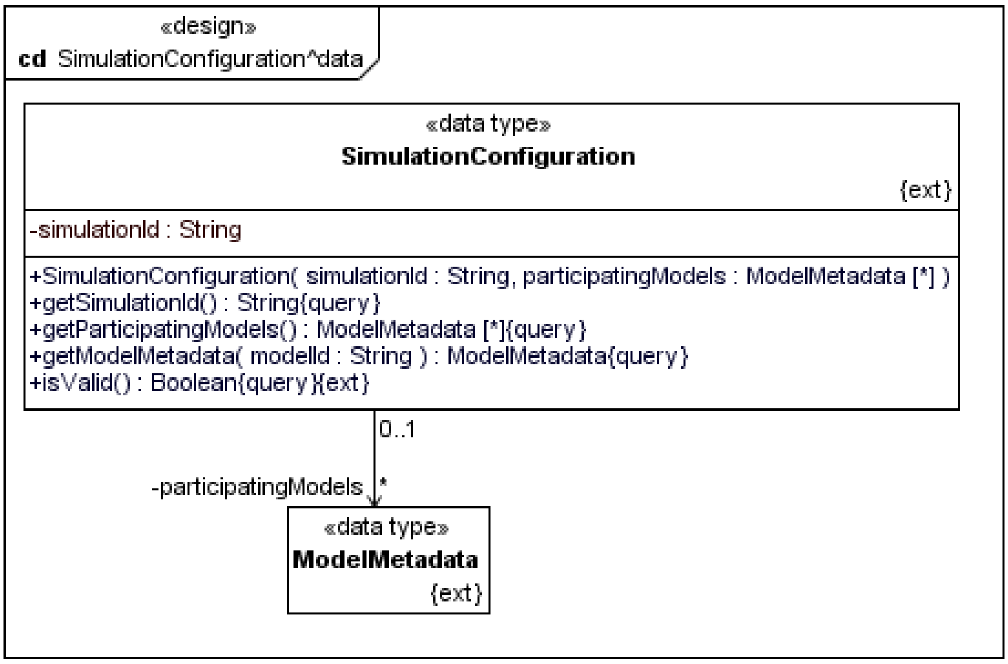
ModelMetadata


«design»
cd ModelMetadata^data
«data type»
ModelMetadata
{ext}
-modelId : String
-modelClass : String
-importInterfaces : String [*]{new}
-exportInterfaces : String [*]{new}
+ModelMetadata(modelId : String, modelClass : String, importInterfaces : String [*], exportInterfaces : String [*]){ext}
+getModelId() : String{query}
+getModelClass() : String{query}
+getImportInterfaces() : String [*]{query}{new}
+getExportInterfaces() : String [*]{query}{new}
+isValid() : Boolean{query}{ext}

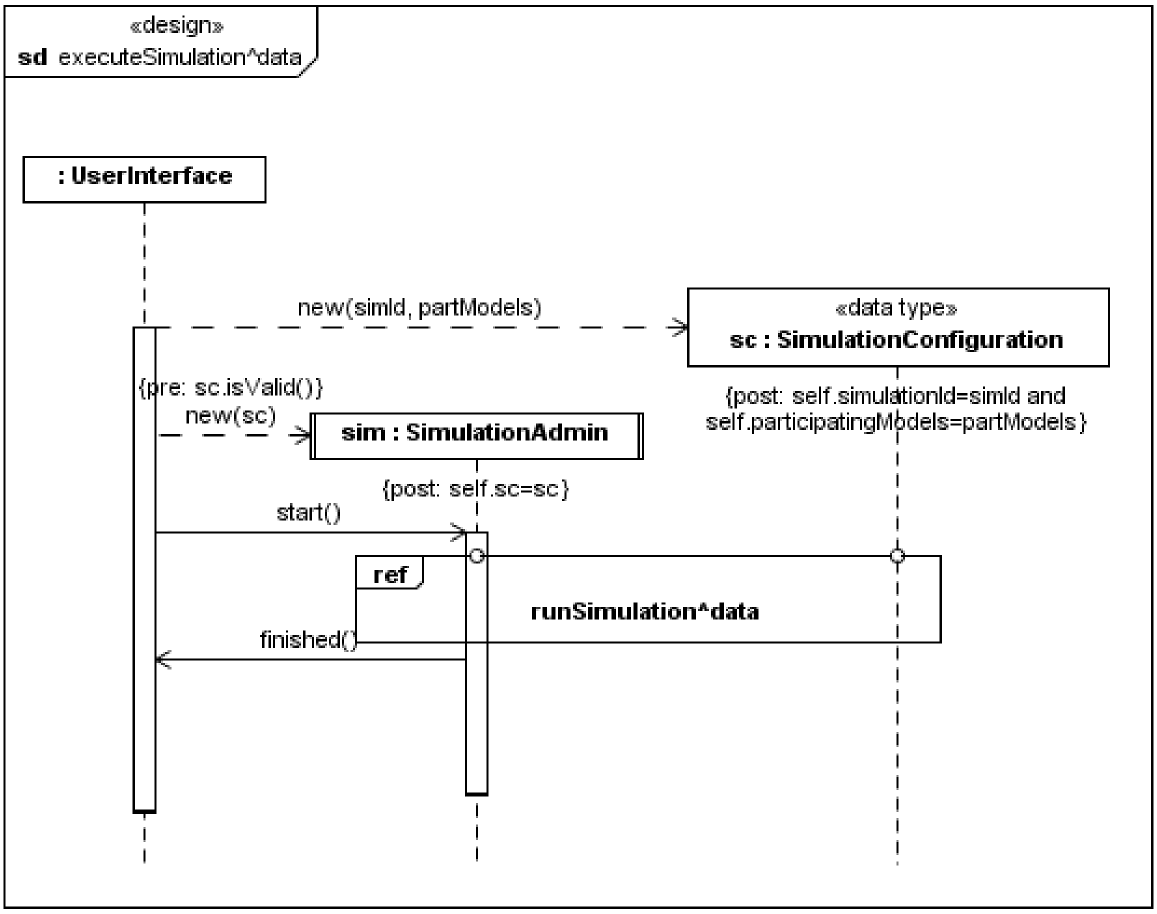
SimulationAdmin

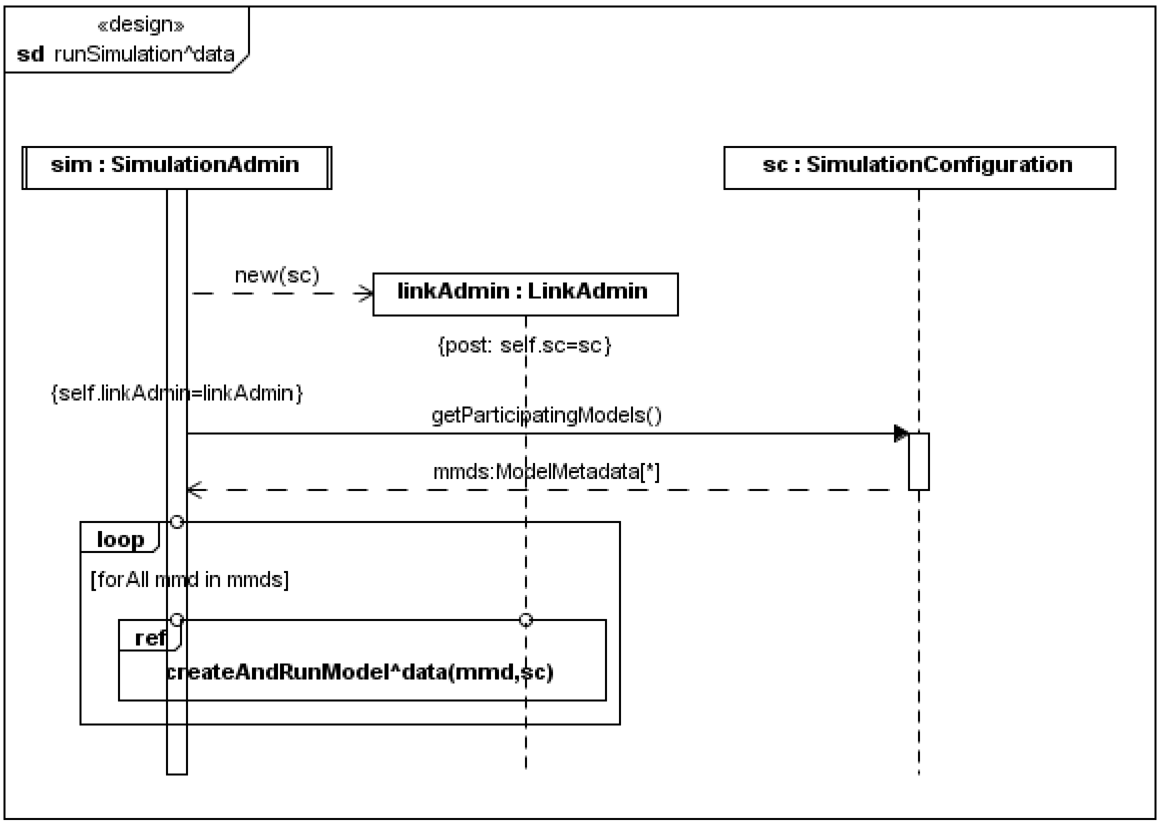


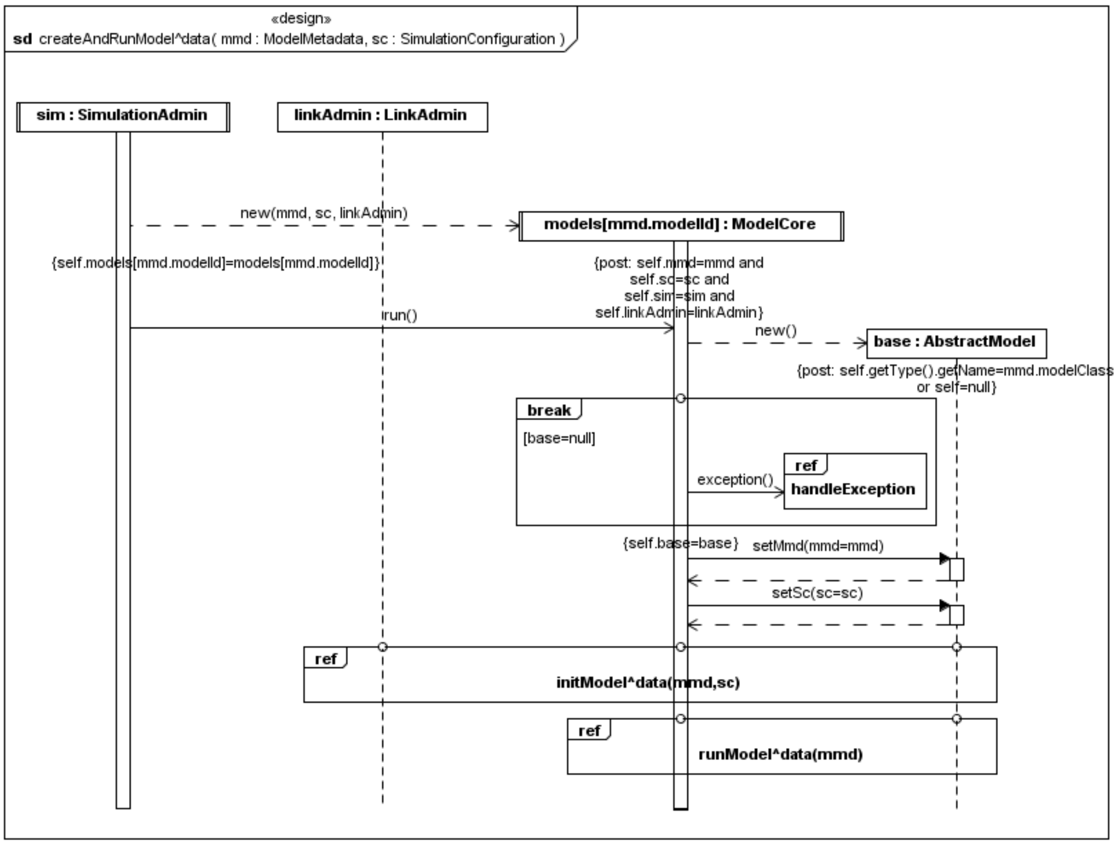
SimulationConfiguration

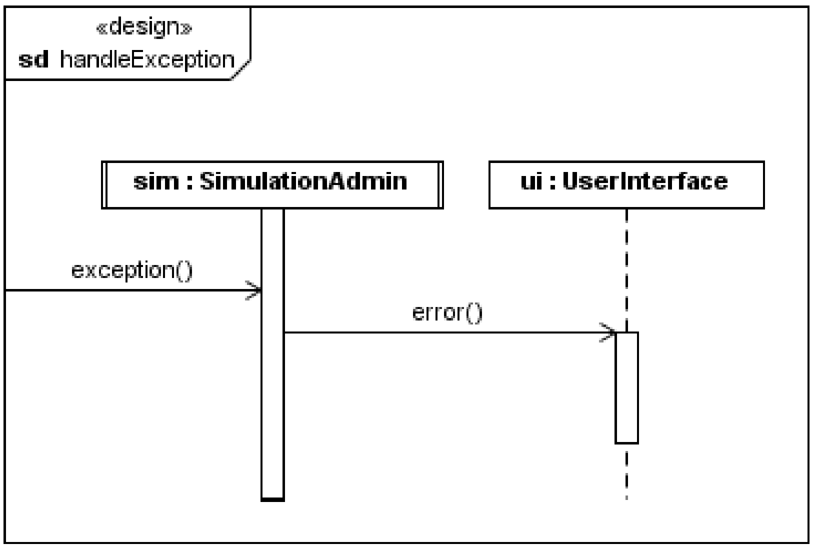

«design»
cd SimulationConfiguration^data
«data type»
SimulationConfiguration
{ext}
-simulationId : String
+SimulationConfiguration(simulationId : String, participatingModels : ModelMetadata [*])
+getSimulationId() : String{query}
+getParticipatingModels() : ModelMetadata [*]{query}
+getModelMetadata(modelId : String) : ModelMetadata{query}
+isValid() : Boolean{query}{ext}
0..1
-participatingModels
*
«data type»
ModelMetadata
{ext}

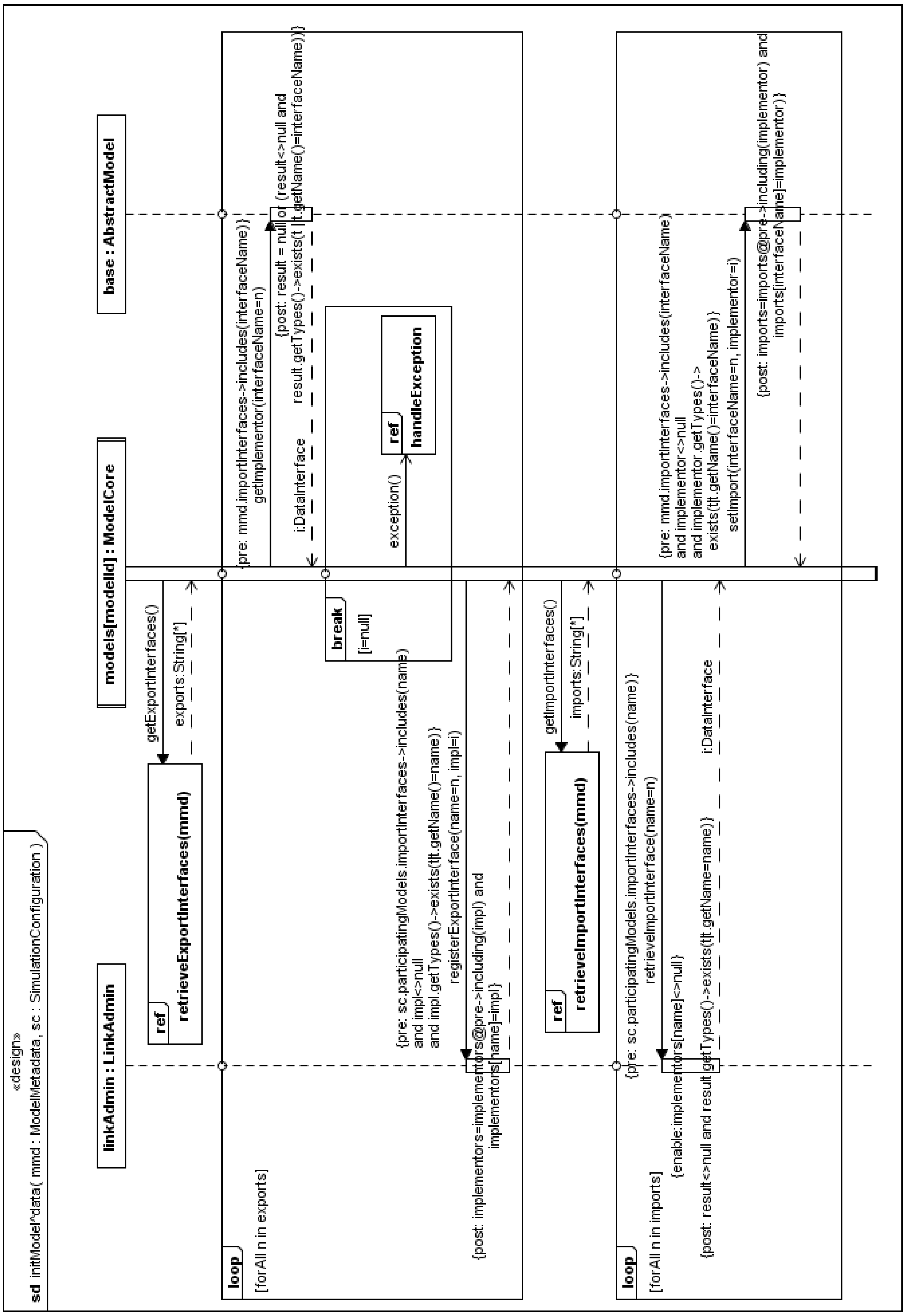
A.2.2 Behavioural Design

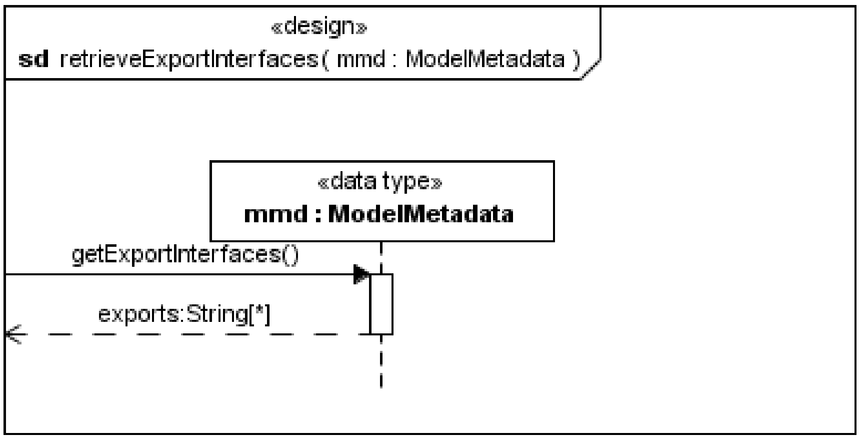


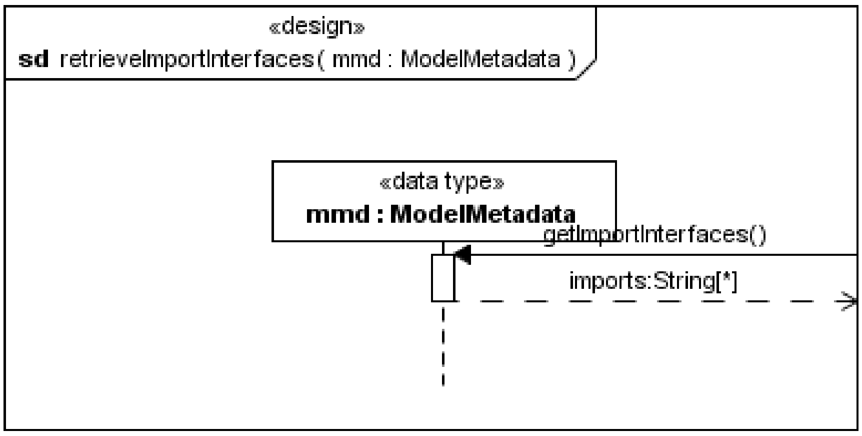

«design»
sd runSimulation^data
sim : SimulationAdmin
sc : SimulationConfiguration
new(sc)
linkAdmin : LinkAdmin
{post: self.sc=sc}
{self.linkAdmin=linkAdmin}
getParticipatingModels()
mmds:ModelMetadata[*]
loop
[forAll mmd in mmds]
ref
createAndRunModel^data(mmd,sc)

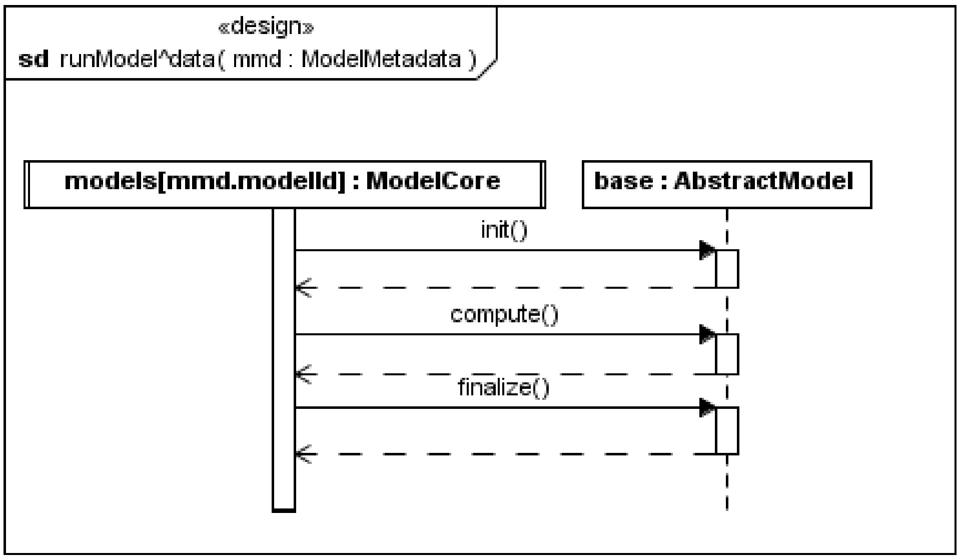

«design»
sd createAndRunModel^data(mmd : ModelMetadata, sc : SimulationConfiguration)
sim : SimulationAdmin
linkAdmin : LinkAdmin
new(mmd, sc, linkAdmin)
models[mmd.modelId] : ModelCore
{self.models[mmd.modelId]=models[mmd.modelId]}
{post: self.mmd=mmd and self.sc=sc and self.sim=sim and self.linkAdmin=linkAdmin}
run()
new()
base : AbstractModel
{post: self.getType().getName=mmd.modelClass or self=null}
break
[base=null]
exception()
ref
handleException
{self.base=base}
setMmd(mmd=mmd)
setSc(sc=sc)
ref
initModel^data(mmd,sc)
ref
runModel^data(mmd)


«design»
sd handleException
sim : SimulationAdmin
ui : UserInterface
exception()
error()

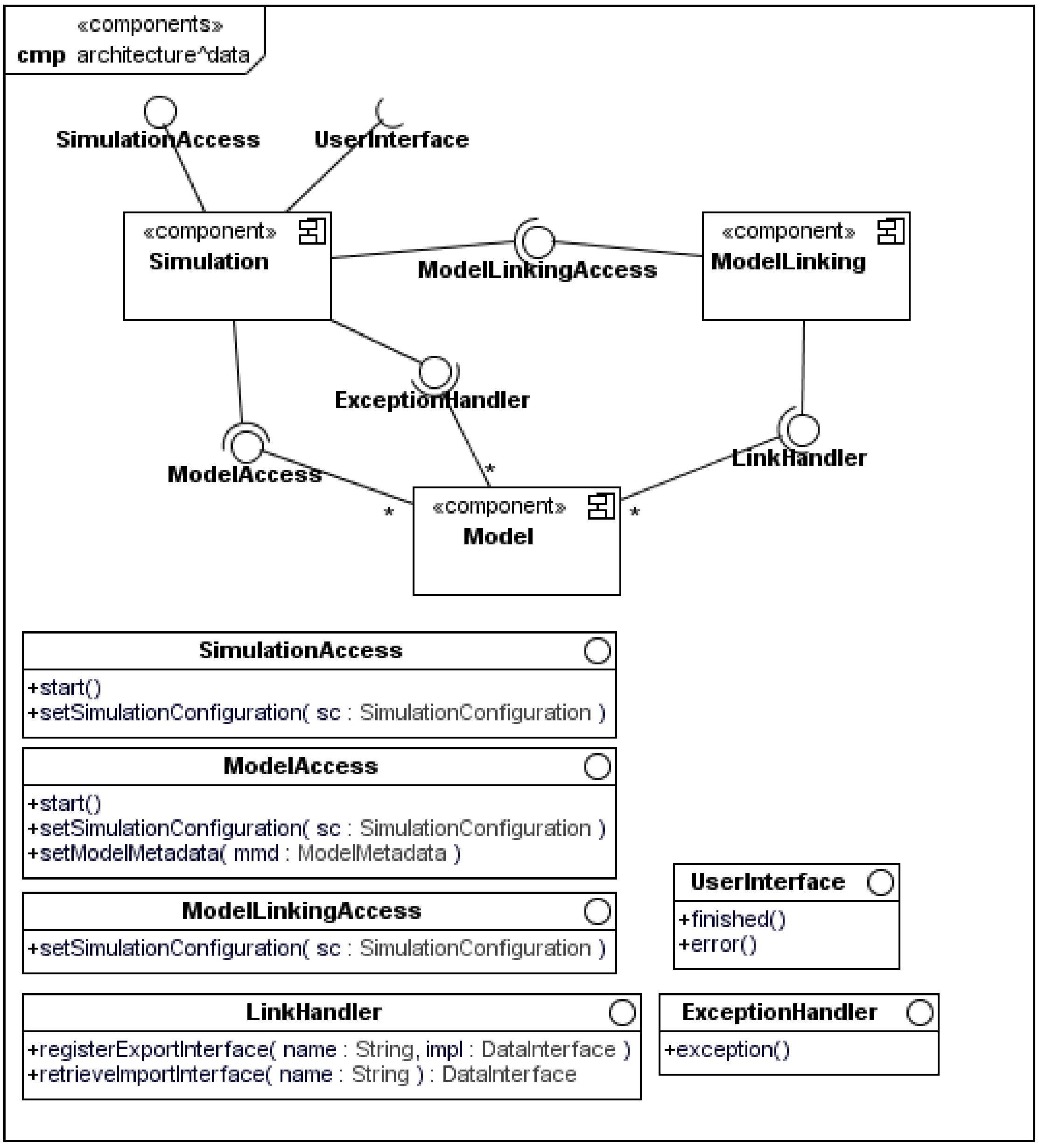

«design»
sd initModel^data(mmd : ModelMetadata, sc : SimulationConfiguration)
linkAdmin : LinkAdmin
models[modelId] : ModelCore
base : AbstractModel
getExportInterfaces()
ref
retrieveExportInterfaces(mmd)
exports:String[*]
loop
[forAll n in exports]
{pre: sc.participatingModels.importInterfaces->includes(name)
and impl<>null
and impl.getTypes()->exists(t|t.getName()=name)}
registerExportInterface(name=n, impl=i)
{post: implementors=implementors@pre->including(impl) and
implementors[name]=impl}
{pre: mmd.importInterfaces->includes(interfaceName)}
getImplementor(interfaceName=n)
{post: result = null or (result<>null and
result.getTypes()->exists(t|t.getName()=interfaceName))}
i:DataInterface
break
[i=null]
exception()
ref
handleException
getImportInterfaces()
imports:String[*]
ref
retrieveImportInterfaces(mmd)
loop
[forAll n in imports]
{pre: sc.participatingModels.importInterfaces->includes(name)}
retrieveImportInterface(name=n)
{enable:implementors[name]<>null}
{post: result<>null and result.getTypes()->exists(t|t.getName=name)}
i:DataInterface
{pre: mmd.importInterfaces->includes(interfaceName)
and implementor<>null
and implementor.getTypes()->
exists(t|t.getName()=interfaceName)}
setImport(interfaceName=n, implementor=i)
{post: imports=imports@pre->including(implementor) and
imports[interfaceName]=implementor}

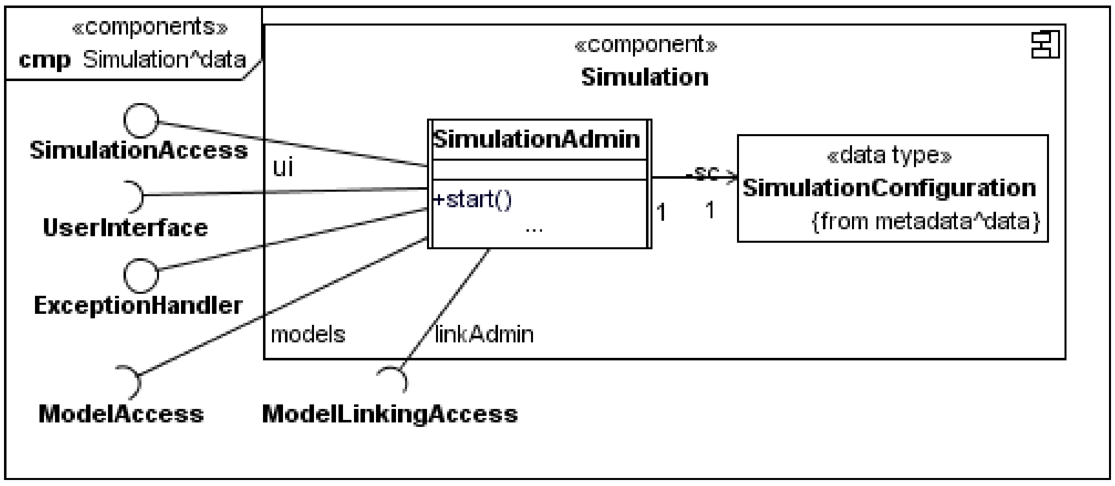

«design»
sd retrieveExportInterfaces(mmd : ModelMetadata)
«data type»
mmd : ModelMetadata
getExportInterfaces()
exports:String[*]

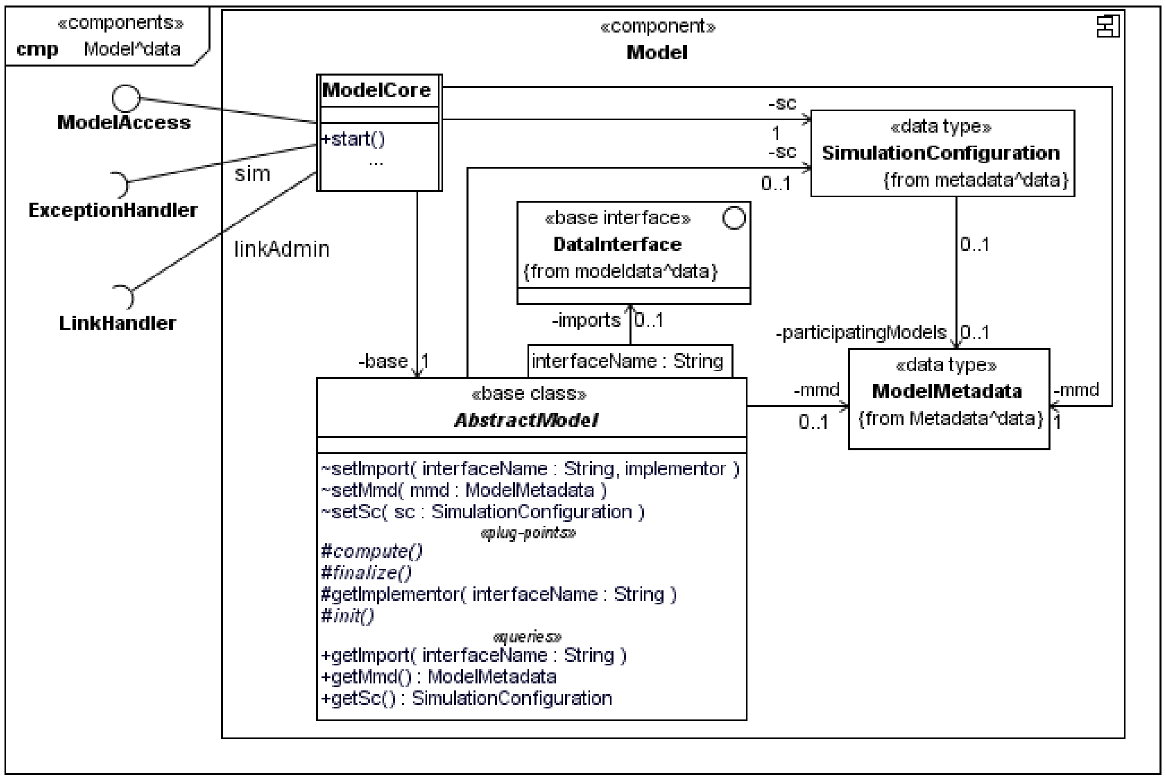

«design»
sd retrieveImportInterfaces(mmd : ModelMetadata)
«data type»
mmd : ModelMetadata
getImportInterfaces()
imports:String[*]

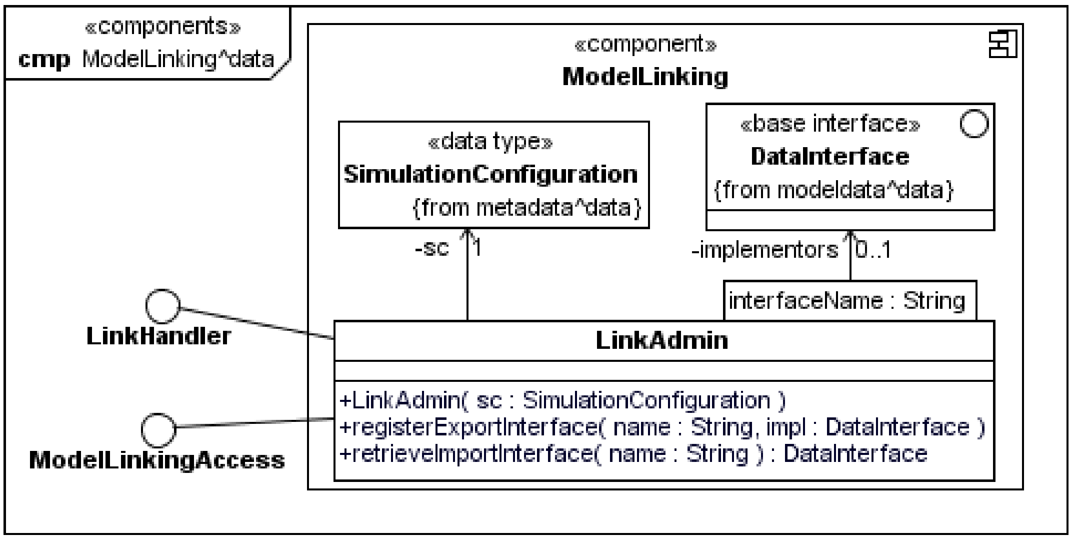

«design»
sd runModel^data(mmd : ModelMetadata)
models[mmd.modelId] : ModelCore
base : AbstractModel
init()
compute()
finalize()

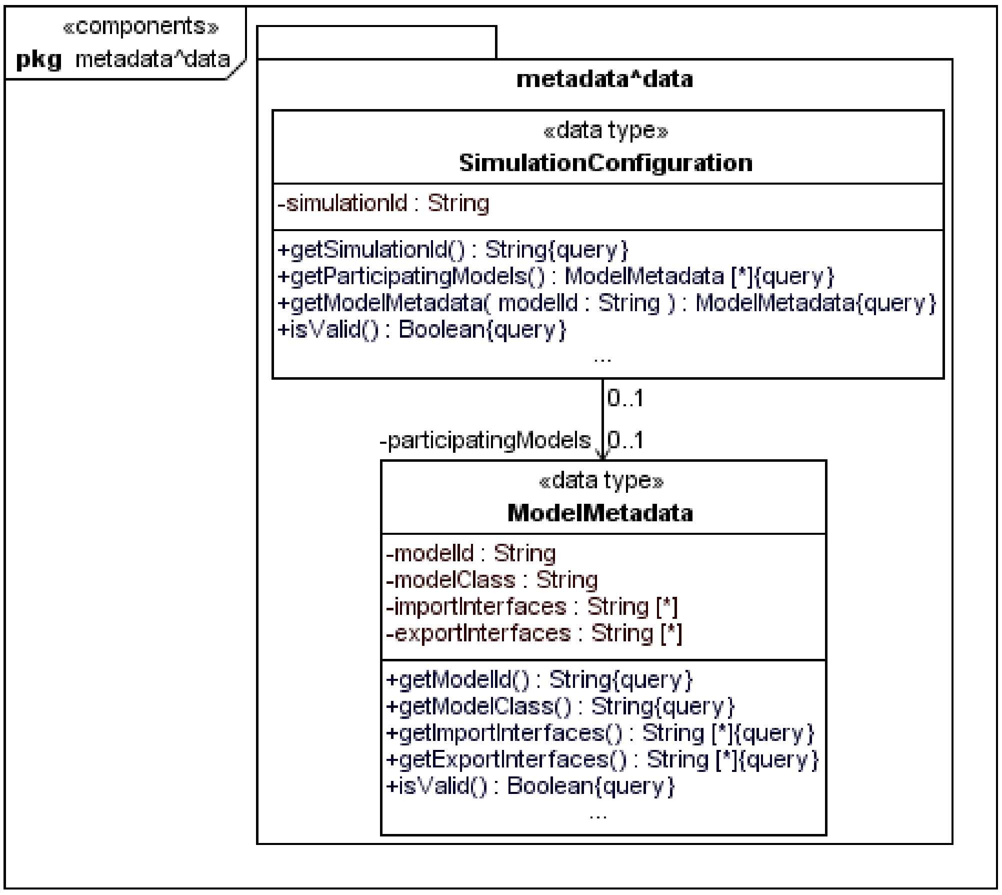
A.2.3 Components

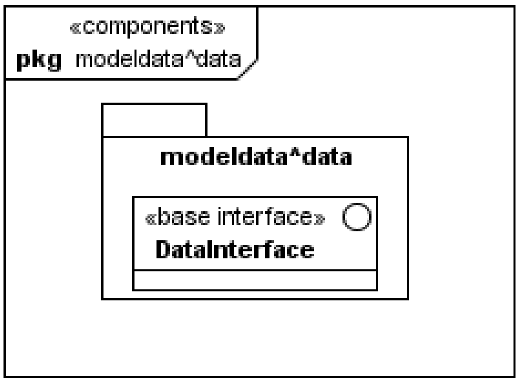

«components»
cmp architecture^data
SimulationAccess
UserInterface
«component» Simulation
ModelLinkingAccess
«component» ModelLinking
ExceptionHandler
ModelAccess
LinkHandler
«component» Model
*
*
*
SimulationAccess
+start()
+setSimulationConfiguration(sc : SimulationConfiguration)
ModelAccess
+start()
+setSimulationConfiguration(sc : SimulationConfiguration)
+setModelMetadata(mmd : ModelMetadata)
ModelLinkingAccess
+setSimulationConfiguration(sc : SimulationConfiguration)
UserInterface
+finished()
+error()
LinkHandler
+registerExportInterface(name : String, impl : DataInterface)
+retrieveImportInterface(name : String) : DataInterface
ExceptionHandler
+exception()


«components»
cmp Simulation^data
«component»
Simulation
SimulationAccess
UserInterface
ExceptionHandler
ModelAccess
ModelLinkingAccess
ui
models
linkAdmin
SimulationAdmin
+start()
...
-sc
1
1
«data type»
SimulationConfiguration
{from metadata^data}


«components»
cmp Model^data
«component»
Model
ModelAccess
ExceptionHandler
LinkHandler
sim
linkAdmin
ModelCore
+start()
...
-sc
1
-sc
0..1
«data type»
SimulationConfiguration
{from metadata^data}
«base interface»
DataInterface
{from modeldata^data}
-imports
0..1
interfaceName : String
-base
1
0..1
-participatingModels
0..1
«data type»
ModelMetadata
{from Metadata^data}
-mmd
0..1
-mmd
1
«base class»
AbstractModel
~setImport(interfaceName : String, implementor)
~setMmd(mmd : ModelMetadata)
~setSc(sc : SimulationConfiguration)
«plug-points»
#compute()
#finalize()
#getImplementor(interfaceName : String)
#init()
«queries»
+getImport(interfaceName : String)
+getMmd() : ModelMetadata
+getSc() : SimulationConfiguration

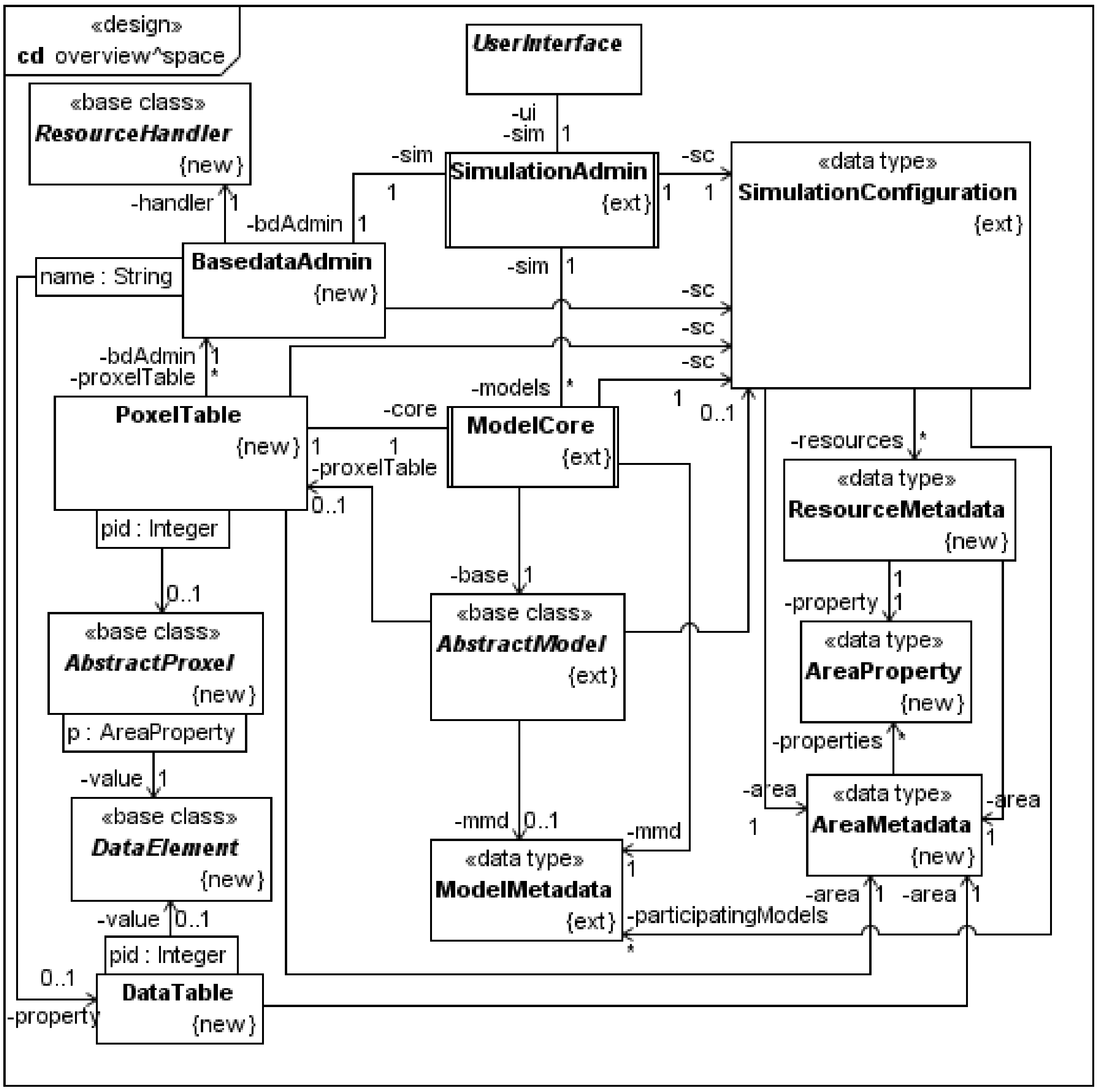

«components»
cmp ModelLinking^data
«component»
ModelLinking
«data type»
SimulationConfiguration
{from metadata^data}
«base interface»
DataInterface
{from modeldata^data}
-sc
1
-implementors
0..1
interfaceName : String
LinkHandler
ModelLinkingAccess
LinkAdmin
+LinkAdmin(sc : SimulationConfiguration)
+registerExportInterface(name : String, impl : DataInterface)
+retrieveImportInterface(name : String) : DataInterface


«components»
pkg metadata^data
metadata^data
«data type»
SimulationConfiguration
-simulationId : String
+getSimulationId() : String{query}
+getParticipatingModels() : ModelMetadata [*]{query}
+getModelMetadata(modelId : String) : ModelMetadata{query}
+isValid() : Boolean{query}
...
0..1
-participatingModels
0..1
«data type»
ModelMetadata
-modelId : String
-modelClass : String
-importInterfaces : String [*]
-exportInterfaces : String [*]
+getModelId() : String{query}
+getModelClass() : String{query}
+getImportInterfaces() : String [*]{query}
+getExportInterfaces() : String [*]{query}
+isValid() : Boolean{query}
...

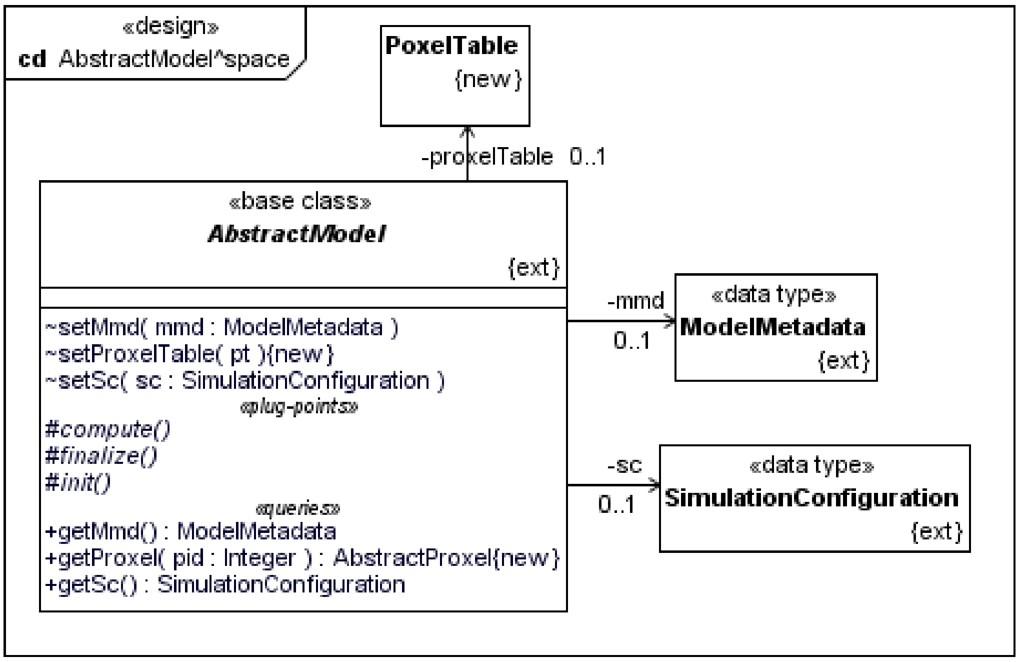

«components»
pkg modeldata^data
modeldata^data
«base interface»
DataInterface

A.3 View “Simulation Space”

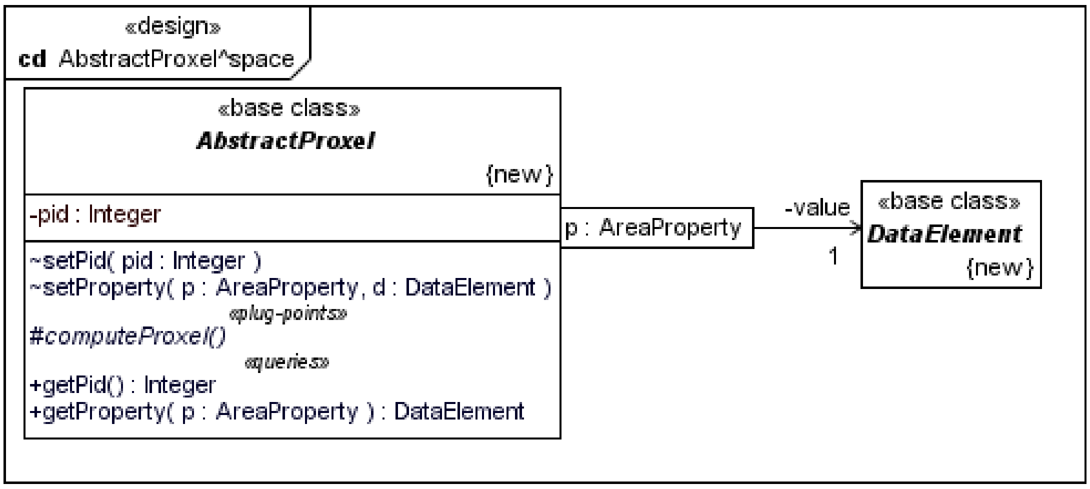
A.3.1 Structural Design


«design»
cd overview^space
UserInterface
«base class» ResourceHandler {new}
SimulationAdmin {ext}
«data type» SimulationConfiguration {ext}
BasedataAdmin {new}
name : String
PoxelTable {new}
pid : Integer
ModelCore {ext}
«data type» ResourceMetadata {new}
«base class» AbstractProxel {new}
p : AreaProperty
«base class» AbstractModel {ext}
«data type» AreaProperty {new}
«base class» DataElement {new}
«data type» ModelMetadata {ext}
«data type» AreaMetadata {new}
DataTable {new}
-ui
-sim
-sc
-handler
-bdAdmin
-proxelTable
-core
-models
-resources
-base
-property
-properties
-value
-mmd
-area
-participatingModels

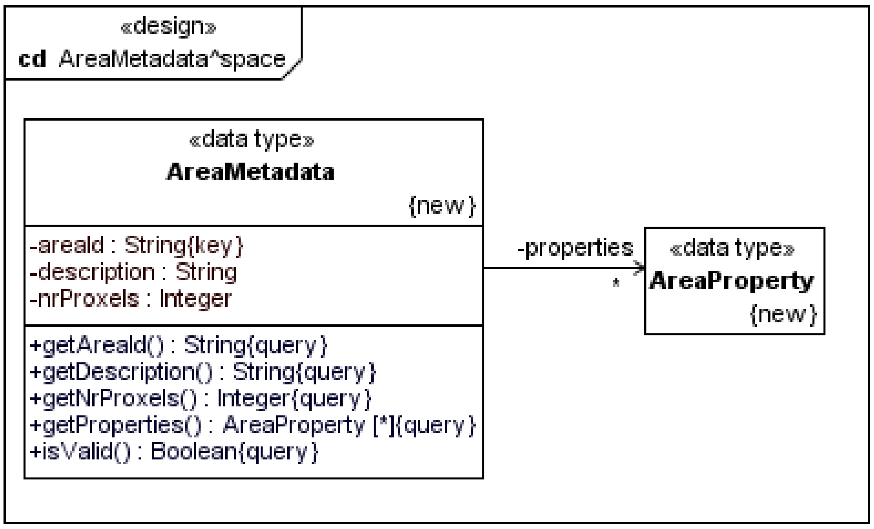
AbstractModel



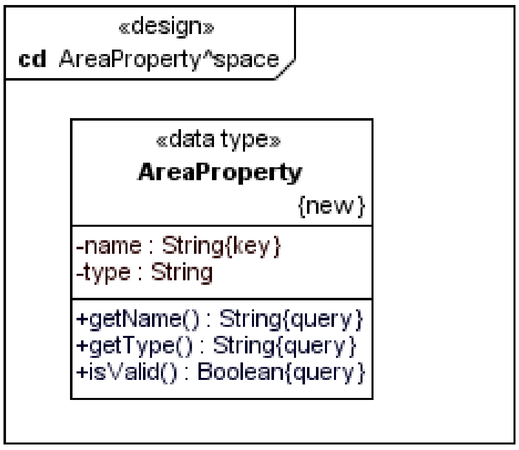
AbstractProxel



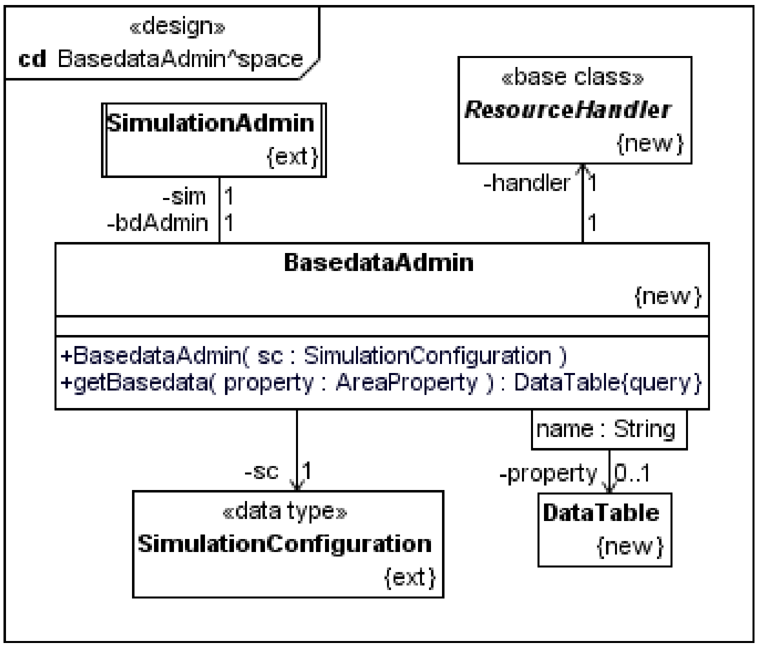
AreaMetadata


«design»
cd AreaMetadata^space
«data type»
AreaMetadata
{new}
-areaId : String{key}
-description : String
-nrProxels : Integer
+getAreaId() : String{query}
+getDescription() : String{query}
+getNrProxels() : Integer{query}
+getProperties() : AreaProperty [*]{query}
+isValid() : Boolean{query}
-properties
*
«data type»
AreaProperty
{new}

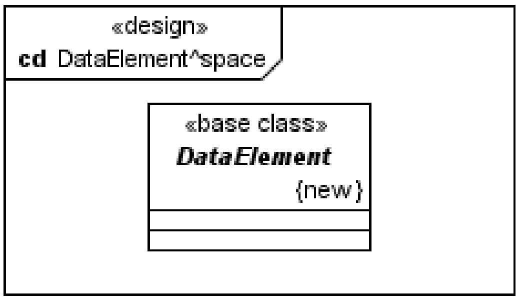
AreaProperty



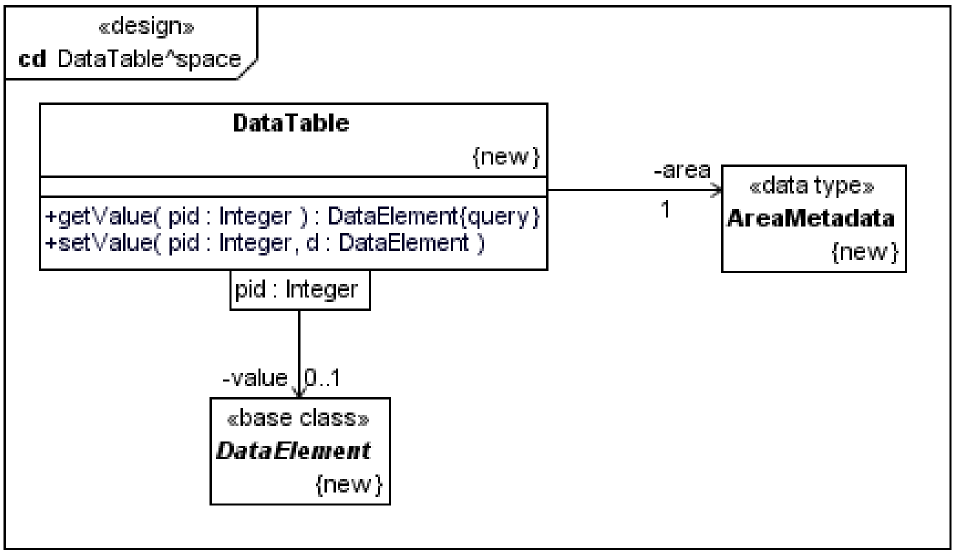
BasedataAdmin



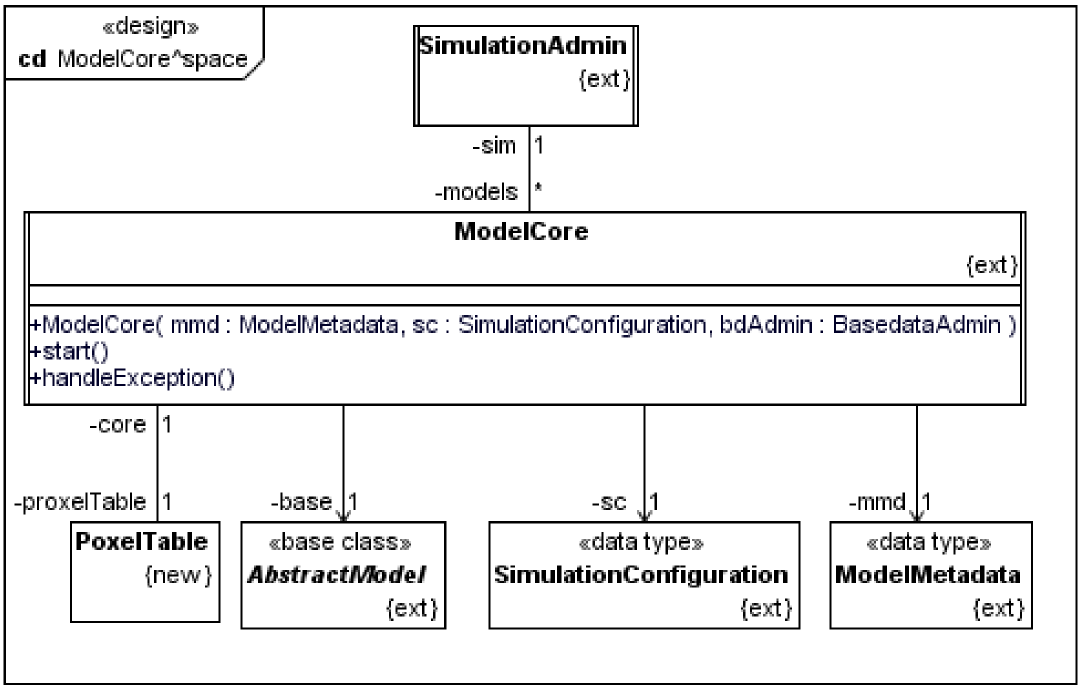
DataElement



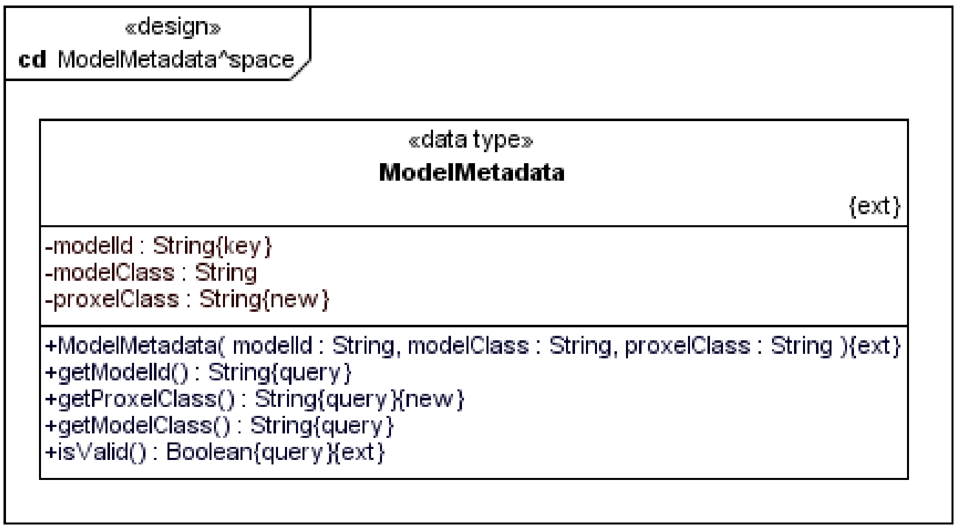
DataTable



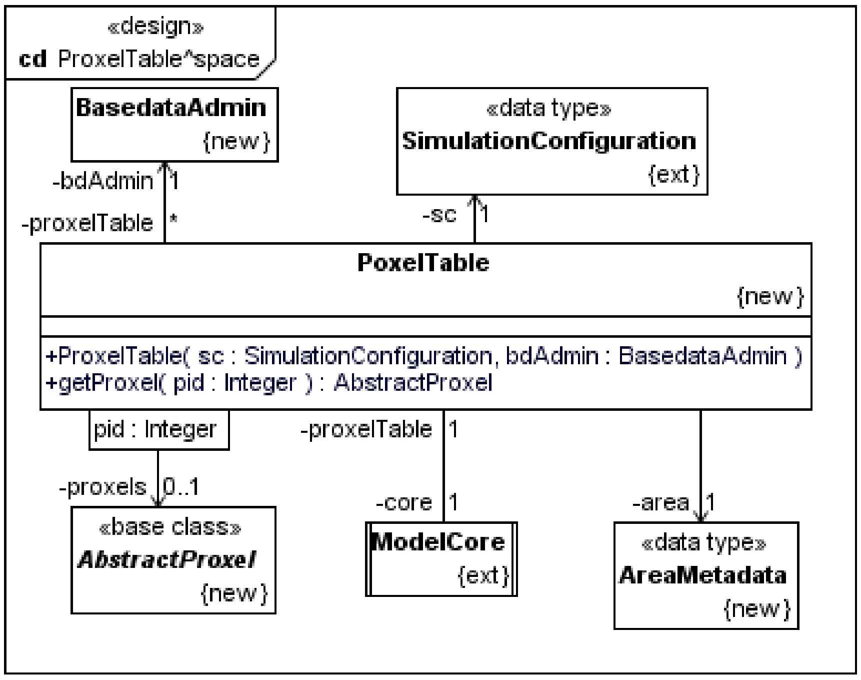
ModelCore



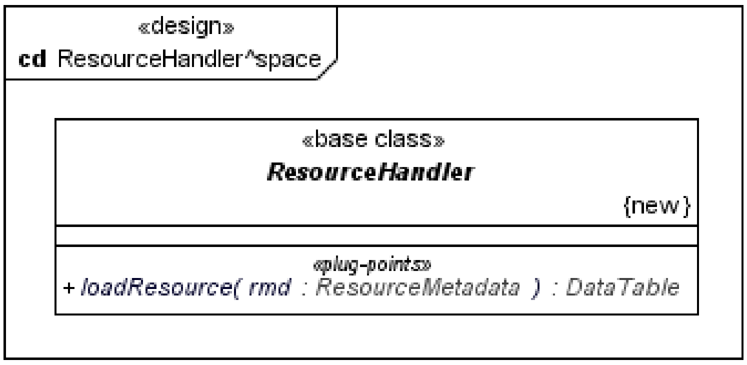
ModelMetadata


«design»
cd ModelMetadata^space
«data type»
ModelMetadata
{ext}
-modelId : String{key}
-modelClass : String
-proxelClass : String{new}
+ModelMetadata(modelId : String, modelClass : String, proxelClass : String){ext}
+getModelId() : String{query}
+getProxelClass() : String{query}{new}
+getModelClass() : String{query}
+isValid() : Boolean{query}{ext}

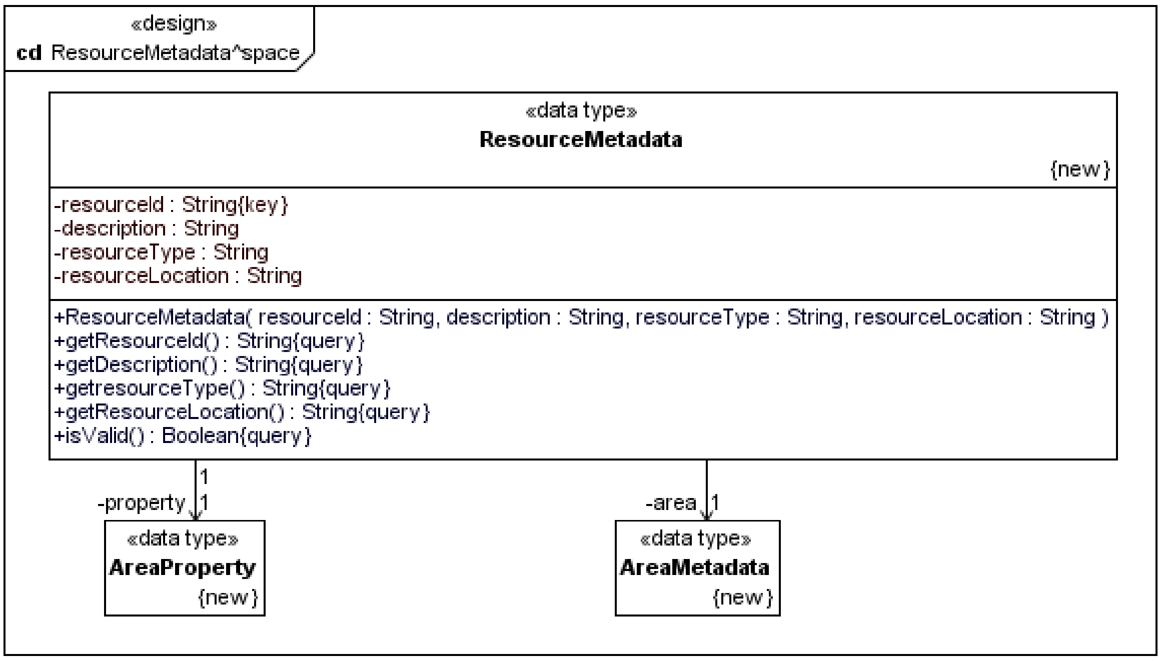
ProxelTable



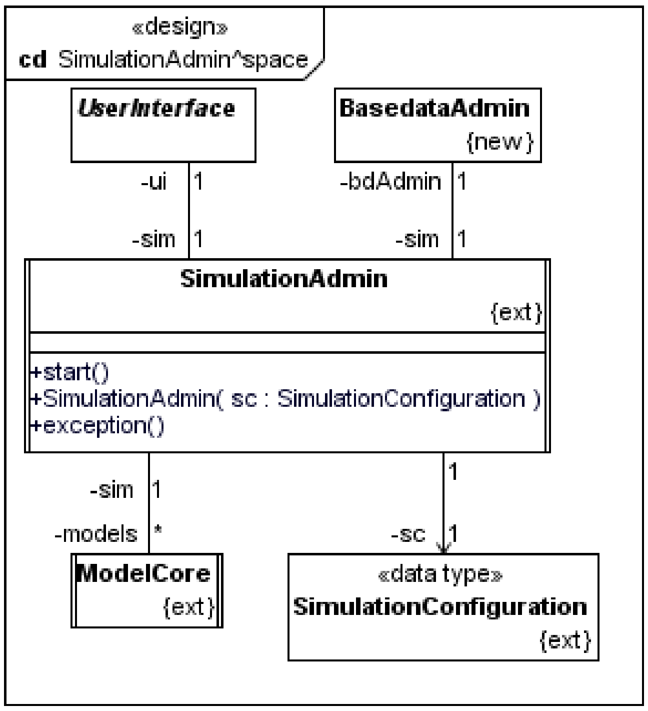
ResourceHandler



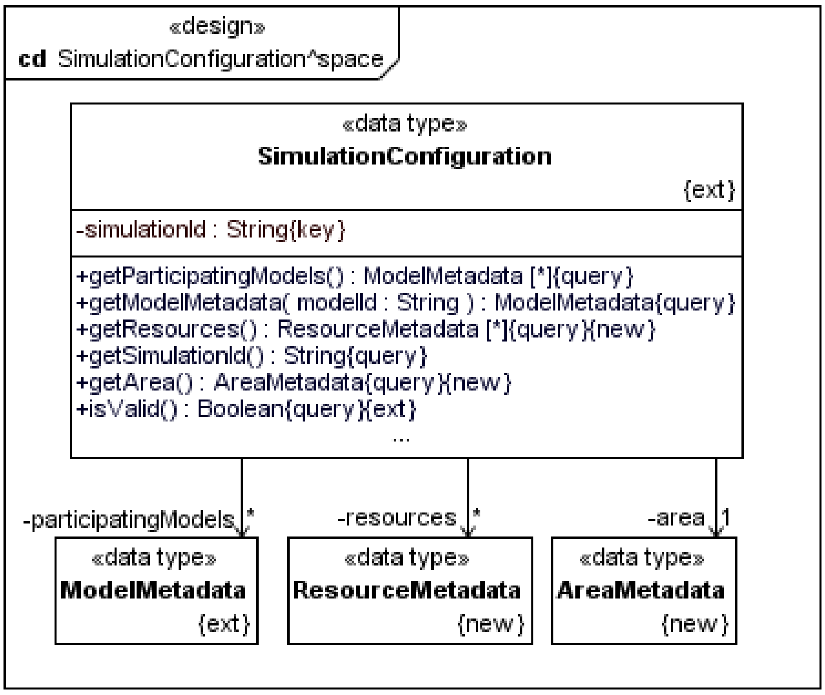
ResourceMetadata


«design»
cd ResourceMetadata^space
«data type»
ResourceMetadata
{new}
-resourceId : String{key}
-description : String
-resourceType : String
-resourceLocation : String
+ResourceMetadata(resourceId : String, description : String, resourceType : String, resourceLocation : String)
+getResourceId() : String{query}
+getDescription() : String{query}
+getresourceType() : String{query}
+getResourceLocation() : String{query}
+isValid() : Boolean{query}
1
-property
1
-area
1
«data type»
AreaProperty
{new}
«data type»
AreaMetadata
{new}

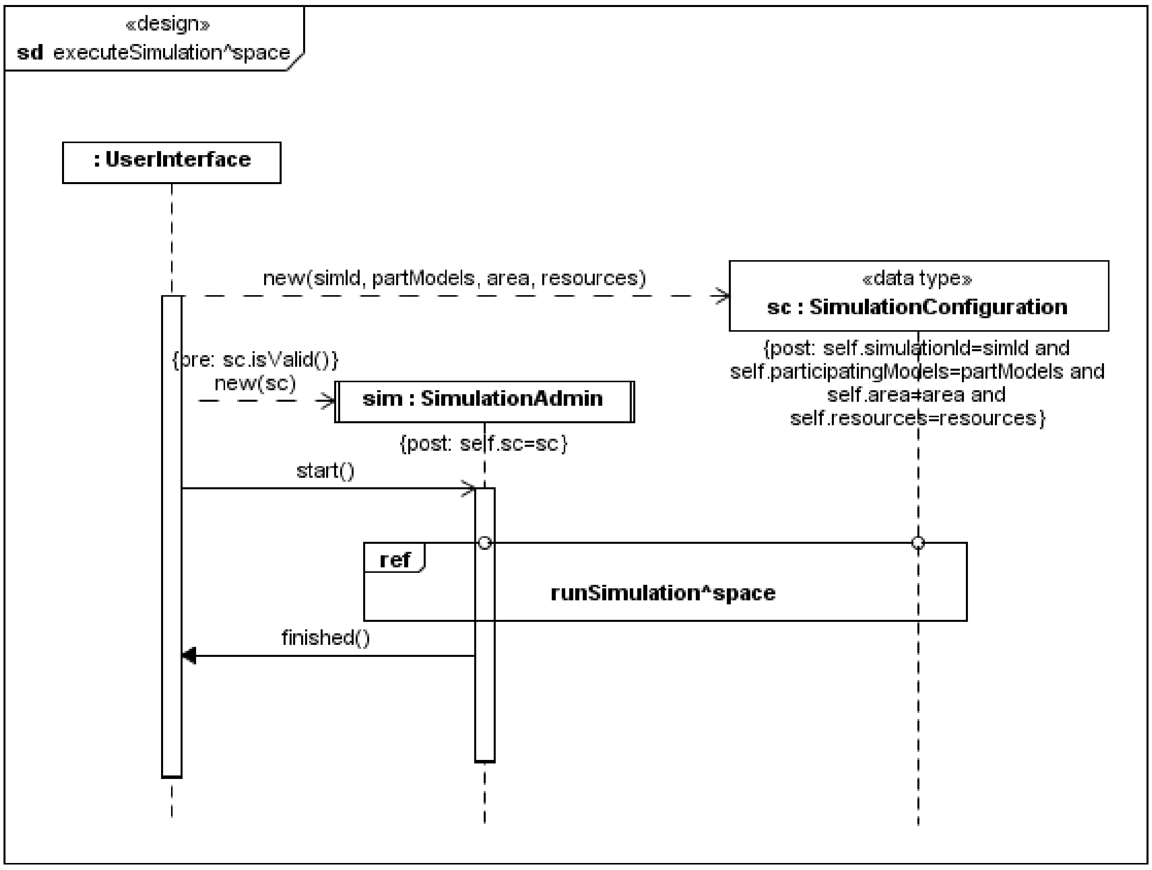
SimulationAdmin

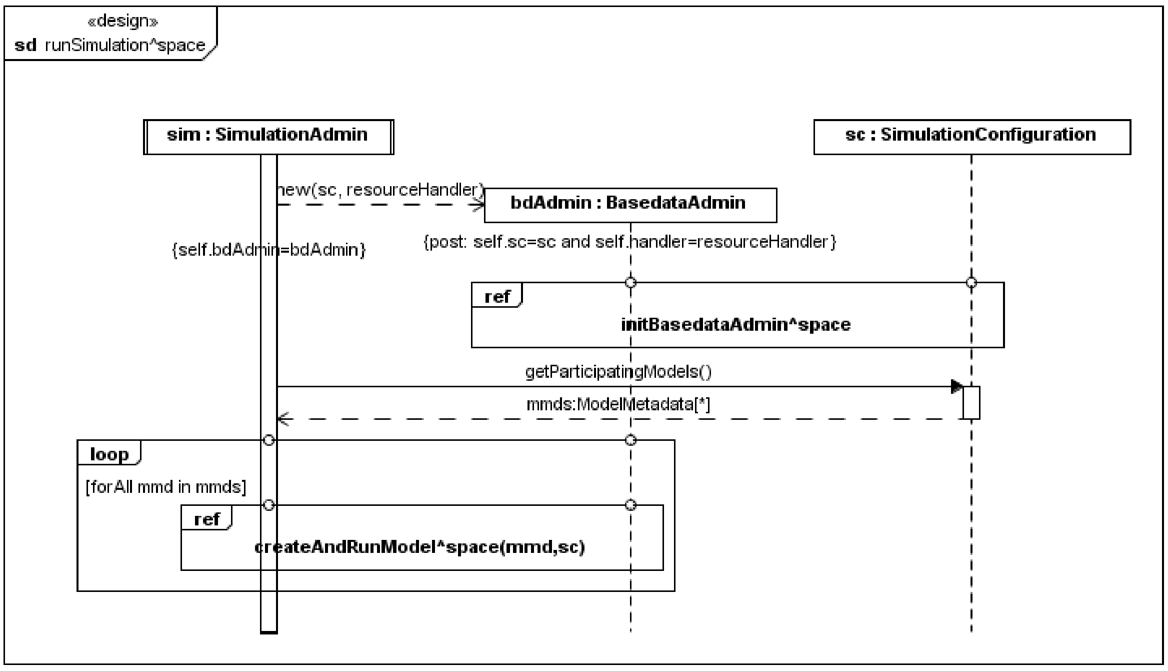


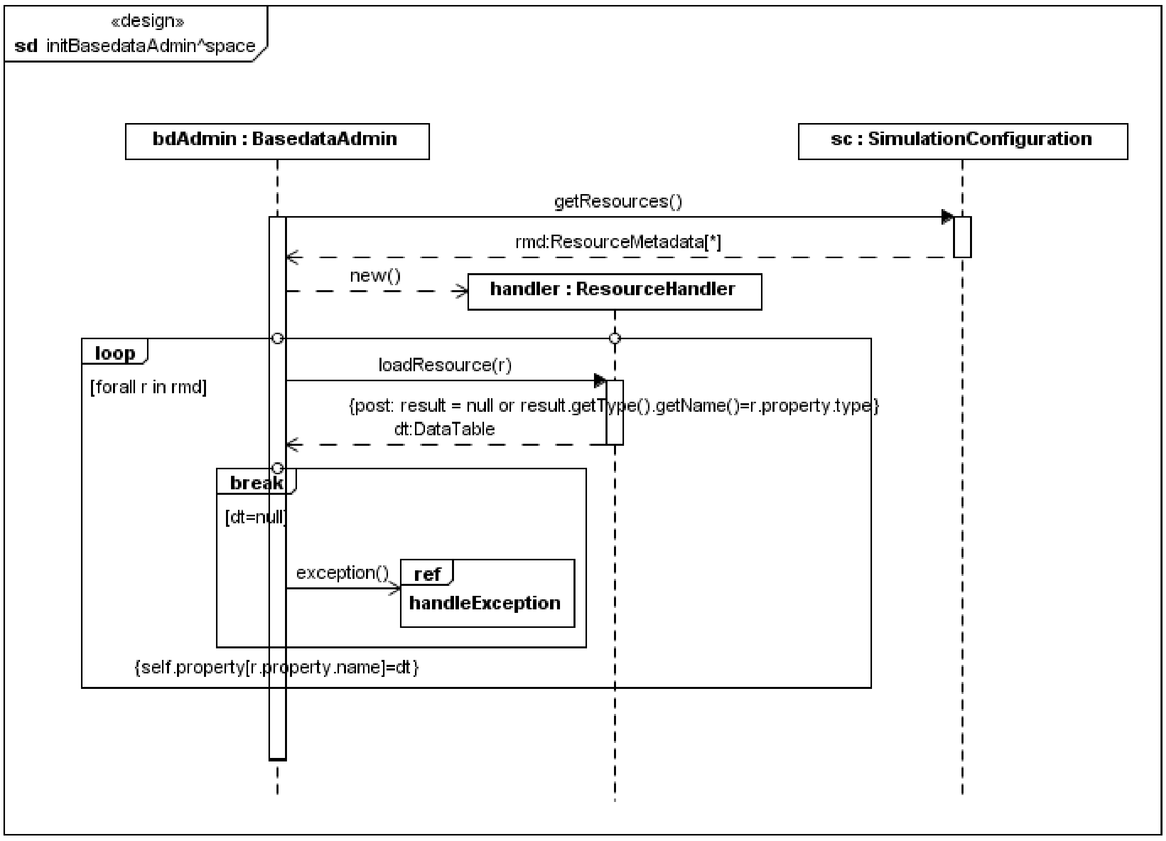
SimulationConfiguration

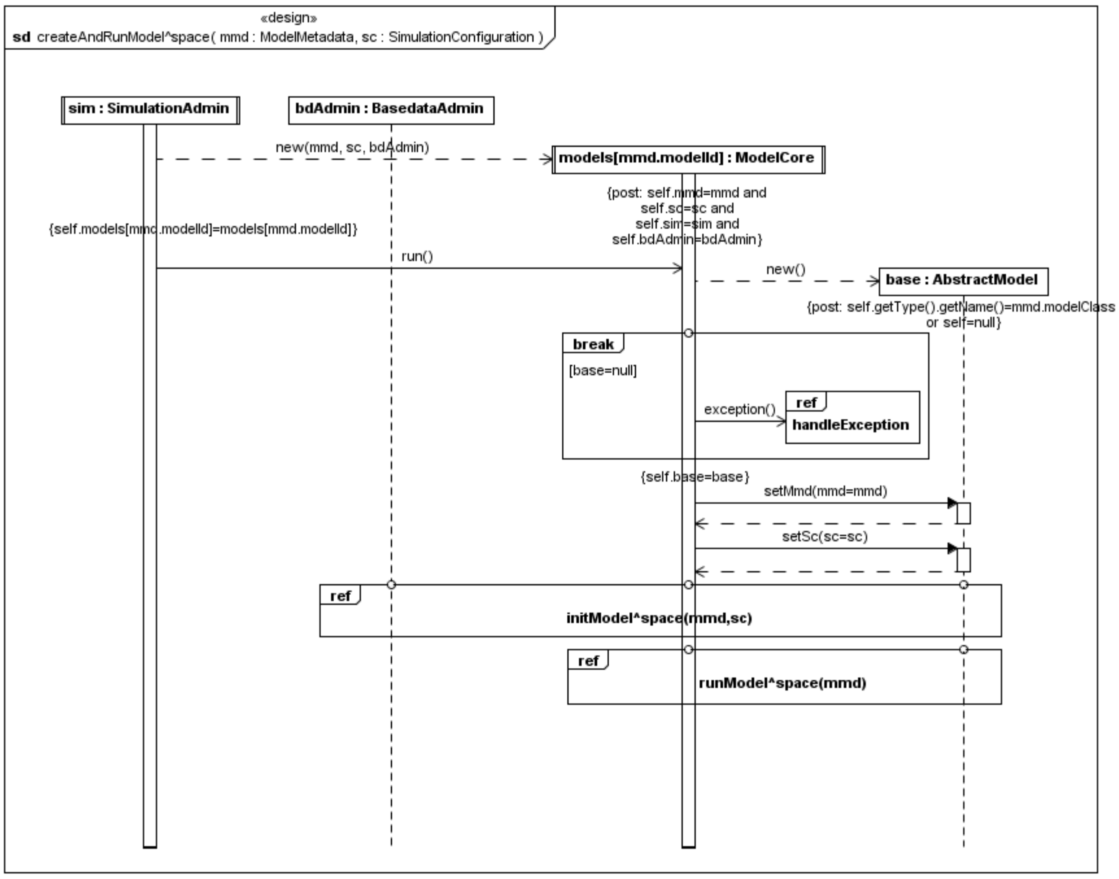

«design»
cd SimulationConfiguration^space
«data type»
SimulationConfiguration
{ext}
-simulationId : String{key}
+getParticipatingModels() : ModelMetadata [*]{query}
+getModelMetadata(modelId : String) : ModelMetadata{query}
+getResources() : ResourceMetadata [*]{query}{new}
+getSimulationId() : String{query}
+getArea() : AreaMetadata{query}{new}
+isValid() : Boolean{query}{ext}
...
-participatingModels *
-resources *
-area 1
«data type»
ModelMetadata
{ext}
«data type»
ResourceMetadata
{new}
«data type»
AreaMetadata
{new}

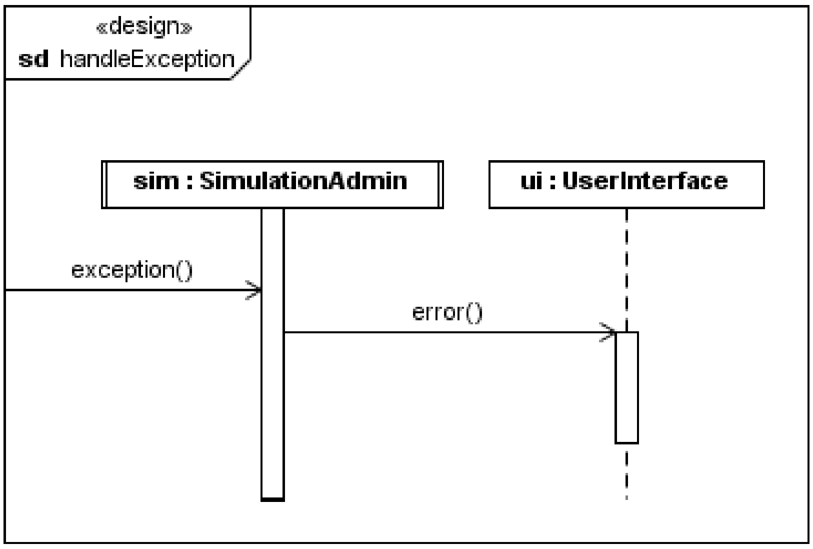
A.3.2 Behavioural Design

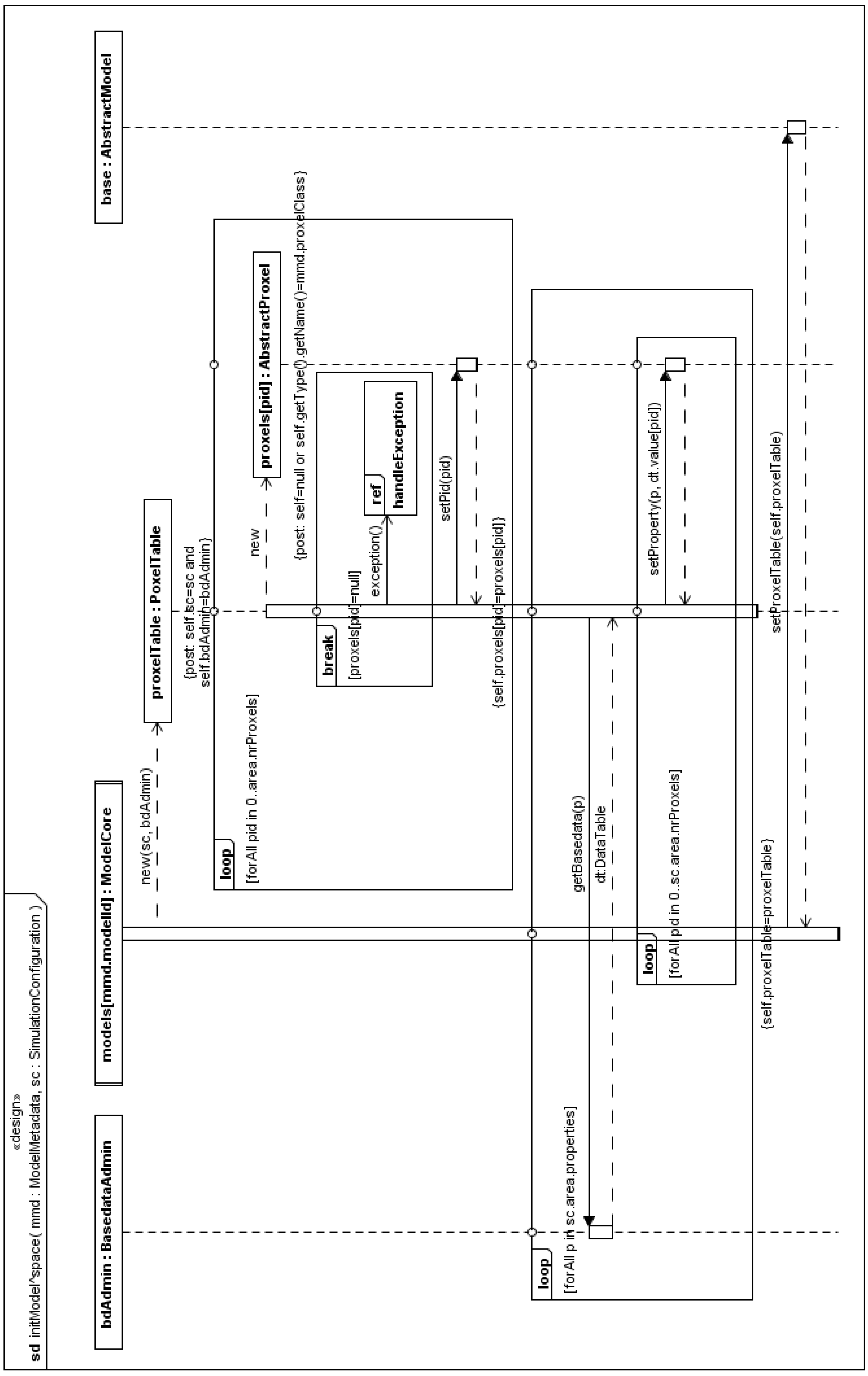

«design»
sd executeSimulation^space
: UserInterface
new(simId, partModels, area, resources)
«data type»
sc : SimulationConfiguration
{post: self.simulationId=simId and self.participatingModels=partModels and self.area=area and self.resources=resources}
{pre: sc.isValid()}
new(sc)
sim : SimulationAdmin
{post: self.sc=sc}
start()
ref
runSimulation^space
finished()

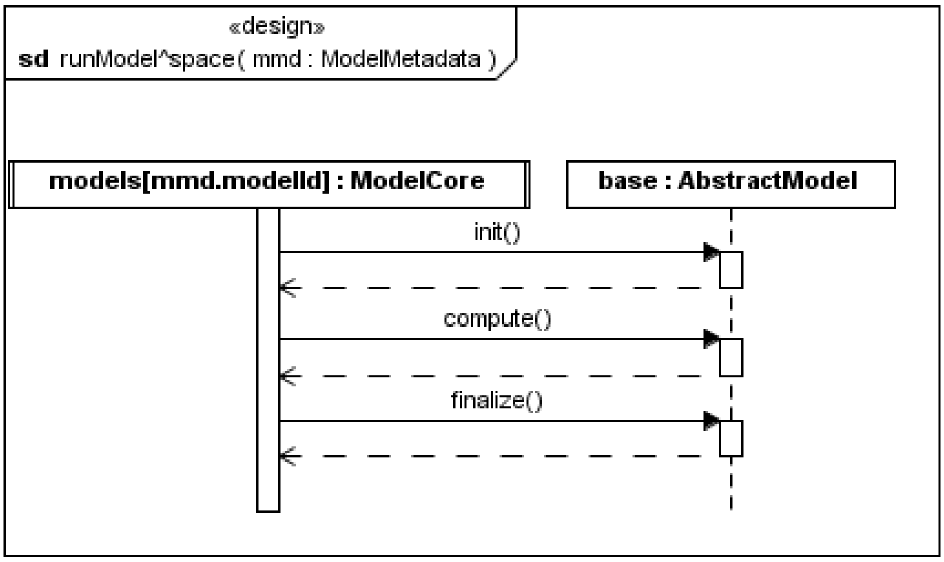

«design»
sd runSimulation^space
sim : SimulationAdmin
sc : SimulationConfiguration
new(sc, resourceHandler)
bdAdmin : BasedataAdmin
{self.bdAdmin=bdAdmin}
{post: self.sc=sc and self.handler=resourceHandler}
ref
initBasedataAdmin^space
getParticipatingModels()
mmds:ModelMetadata[*]
loop
[forAll mmd in mmds]
ref
createAndRunModel^space(mmd,sc)


«design»
sd initBasedataAdmin^space
bdAdmin : BasedataAdmin
sc : SimulationConfiguration
getResources()
rmd:ResourceMetadata[*]
new()
handler : ResourceHandler
loop
[forall r in rmd]
loadResource(r)
{post: result = null or result.getType().getName()=r.property.type}
dt:DataTable
break
[dt=null]
exception()
ref
handleException
{self.property[r.property.name]=dt}

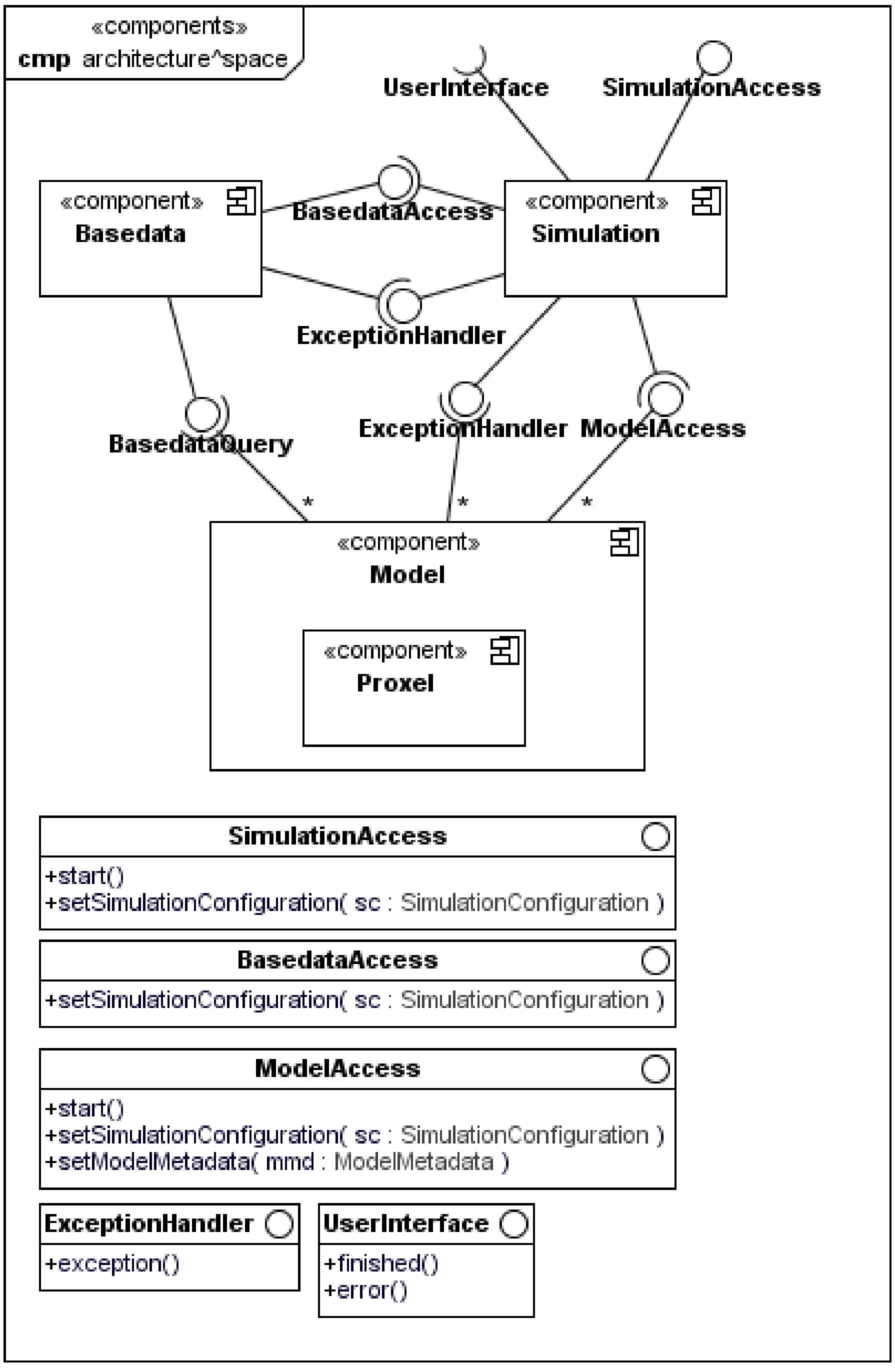

«design»
sd createAndRunModel^space(mmd : ModelMetadata, sc : SimulationConfiguration)
sim : SimulationAdmin
bdAdmin : BasedataAdmin
new(mmd, sc, bdAdmin)
models[mmd.modelId] : ModelCore
{post: self.mmd=mmd and
self.sc=sc and
self.sim=sim and
self.bdAdmin=bdAdmin}
{self.models[mmd.modelId]=models[mmd.modelId]}
run()
new()
base : AbstractModel
{post: self.getType().getName()=mmd.modelClass
or self=null}
break
[base=null]
exception()
ref
handleException
{self.base=base}
setMmd(mmd=mmd)
setSc(sc=sc)
ref
initModel^space(mmd,sc)
ref
runModel^space(mmd)

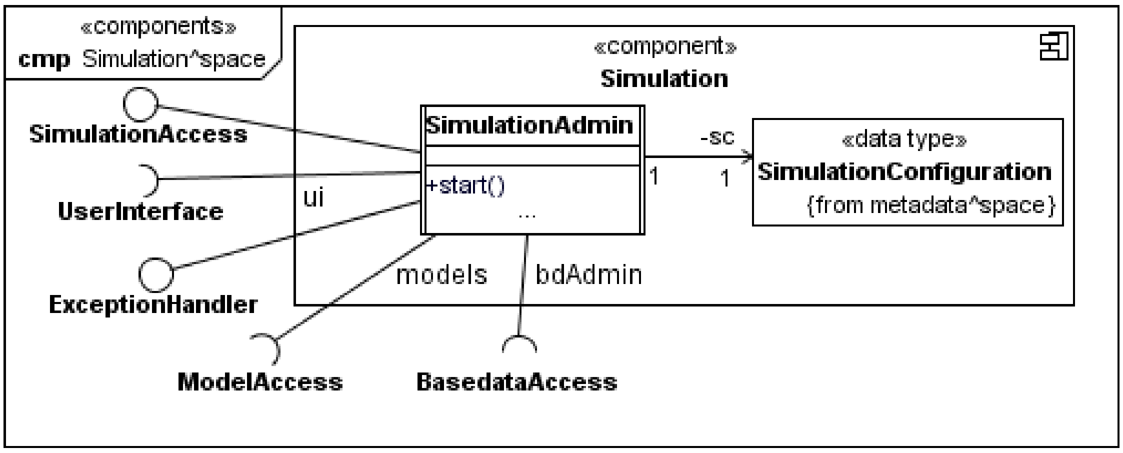

«design»
sd handleException
sim : SimulationAdmin
ui : UserInterface
exception()
error()

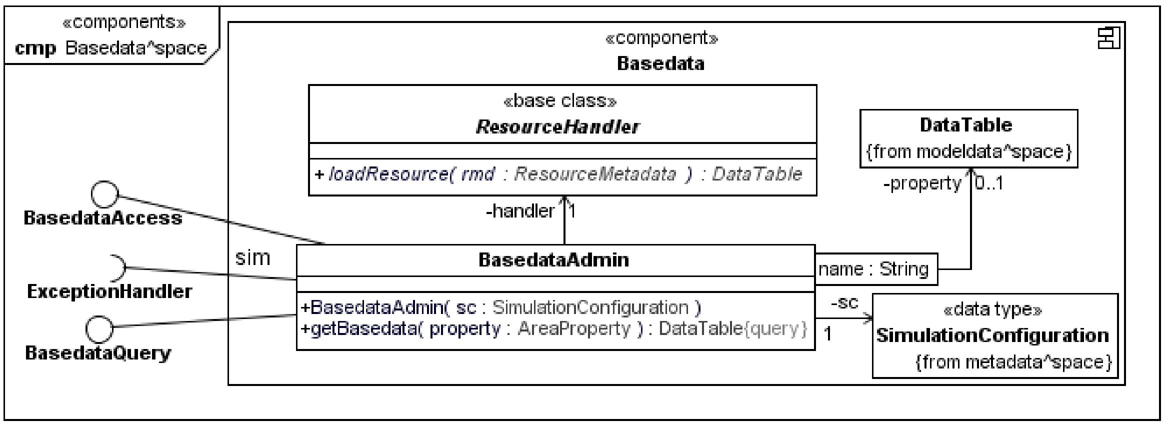

«design»
sd initModel^space(mmd : ModelMetadata, sc : SimulationConfiguration)
bdAdmin : BasedataAdmin
models[mmd.modelId] : ModelCore
proxelTable : ProxelTable
base : AbstractModel
new(sc, bdAdmin)
{post: self.sc=sc and self.bdAdmin=bdAdmin}
loop
[forAll pid in 0..area.nrProxels]
new
proxels[pid] : AbstractProxel
{post: self=null or self.getType().getName()=mmd.proxelClass}
break
[proxels[pid]=null]
exception()
ref
handleException
setPid(pid)
{self.proxels[pid]=proxels[pid]}
loop
[forAll p in sc.area.properties]
getBasedata(p)
dt:DataTable
loop
[forAll pid in 0..sc.area.nrProxels]
setProperty(p, dt.value[pid])
setProxelTable(self.proxelTable)
{self.proxelTable=proxelTable}

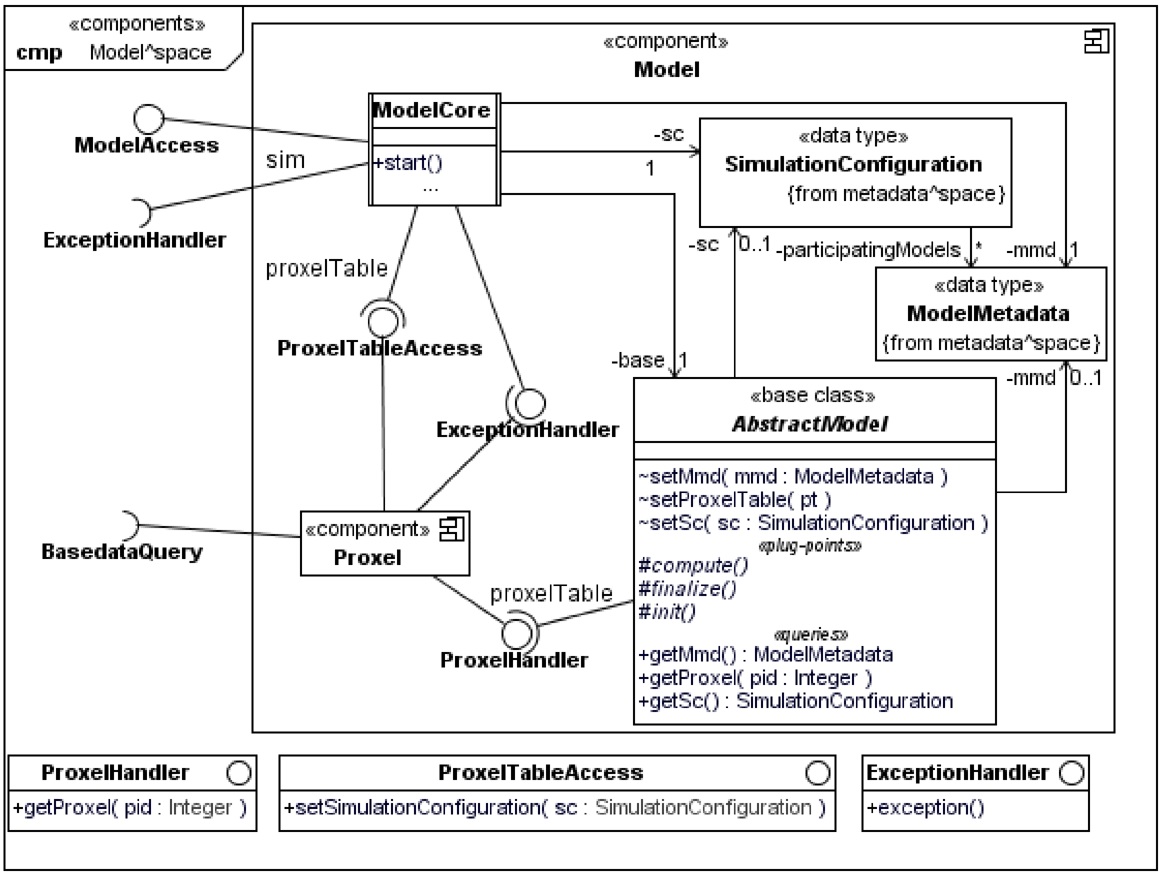

«design»
sd runModel^space(mmd : ModelMetadata)
models[mmd.modelId] : ModelCore
base : AbstractModel
init()
compute()
finalize()

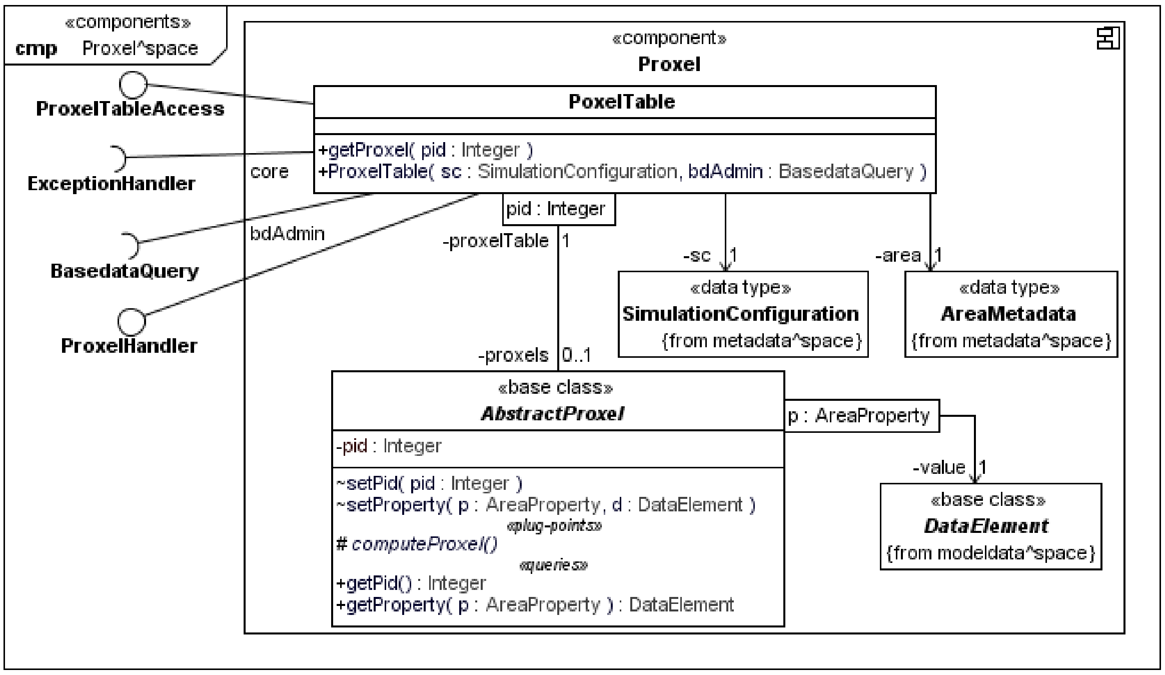
A.3.3 Components

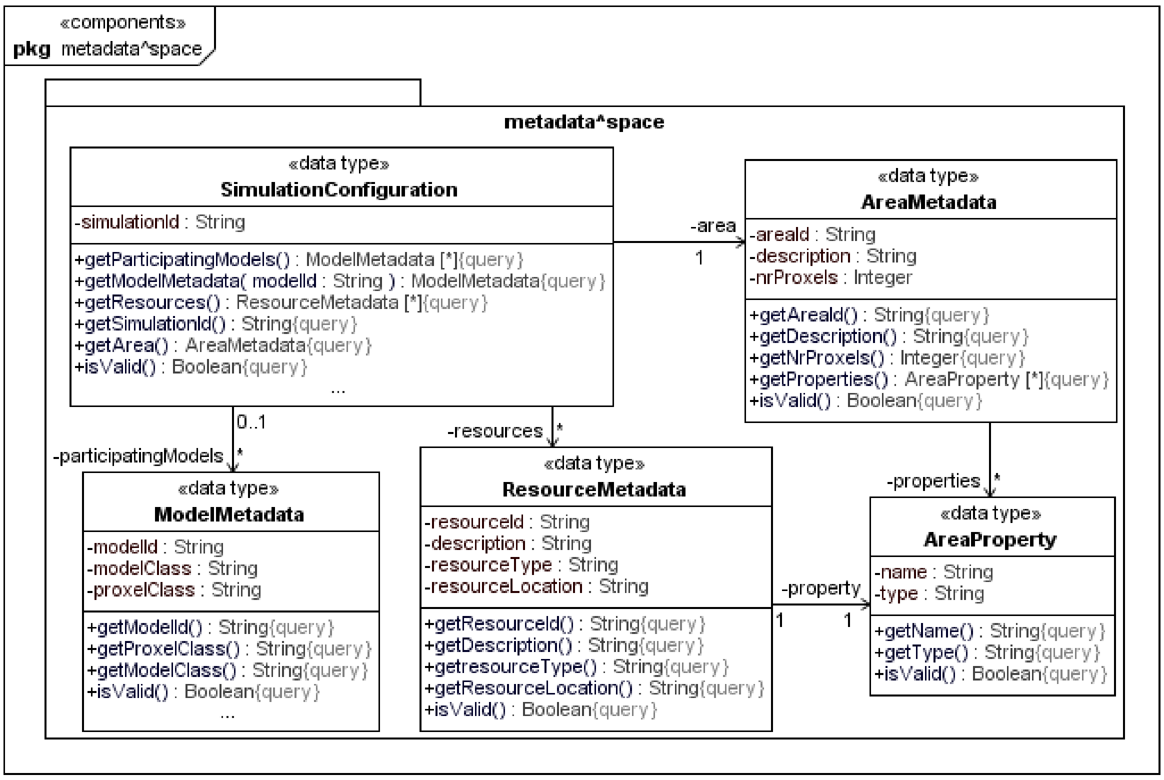


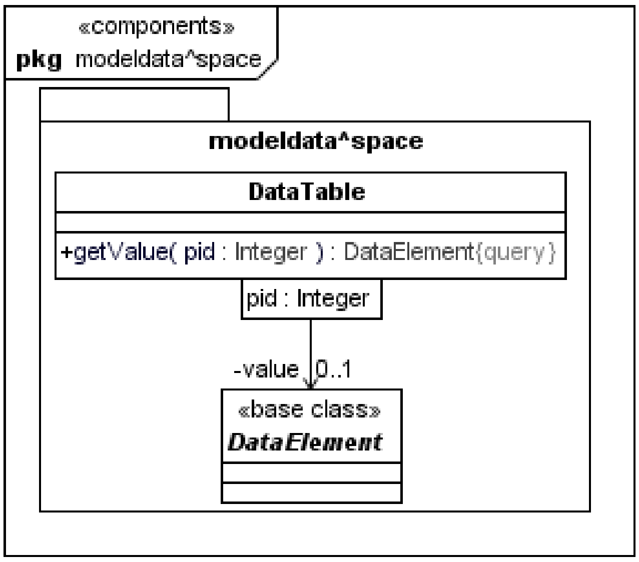

«components»
cmp Simulation^space
«component»
Simulation
SimulationAccess
UserInterface
ExceptionHandler
ModelAccess
BasedataAccess
ui
models
bdAdmin
SimulationAdmin
+start()
...
-sc
1
1
«data type»
SimulationConfiguration
{from metadata^space}


«components»
cmp Basedata^space
«component»
Basedata
«base class»
ResourceHandler
+ loadResource(rmd : ResourceMetadata) : DataTable
DataTable
{from modeldata^space}
-property
0..1
-handler
1
BasedataAccess
ExceptionHandler
BasedataQuery
sim
BasedataAdmin
+BasedataAdmin(sc : SimulationConfiguration)
+getBasedata(property : AreaProperty) : DataTable{query}
name : String
-sc
1
«data type»
SimulationConfiguration
{from metadata^space}


«components»
cmp Model^space
«component»
Model
ModelCore
+start()
...
ModelAccess
sim
ExceptionHandler
-sc
1
«data type»
SimulationConfiguration
{from metadata^space}
-sc
0..1
-participatingModels
*
-mmd
1
«data type»
ModelMetadata
{from metadata^space}
proxelTable
ProxelTableAccess
-base
1
-mmd
0..1
ExceptionHandler
«base class»
AbstractModel
~setMmd(mmd : ModelMetadata)
~setProxelTable(pt)
~setSc(sc : SimulationConfiguration)
«plug-points»
#compute()
#finalize()
#init()
«queries»
+getMmd() : ModelMetadata
+getProxel(pid : Integer)
+getSc() : SimulationConfiguration
BasedataQuery
«component»
Proxel
proxelTable
ProxelHandler
ProxelHandler
+getProxel(pid : Integer)
ProxelTableAccess
+setSimulationConfiguration(sc : SimulationConfiguration)
ExceptionHandler
+exception()

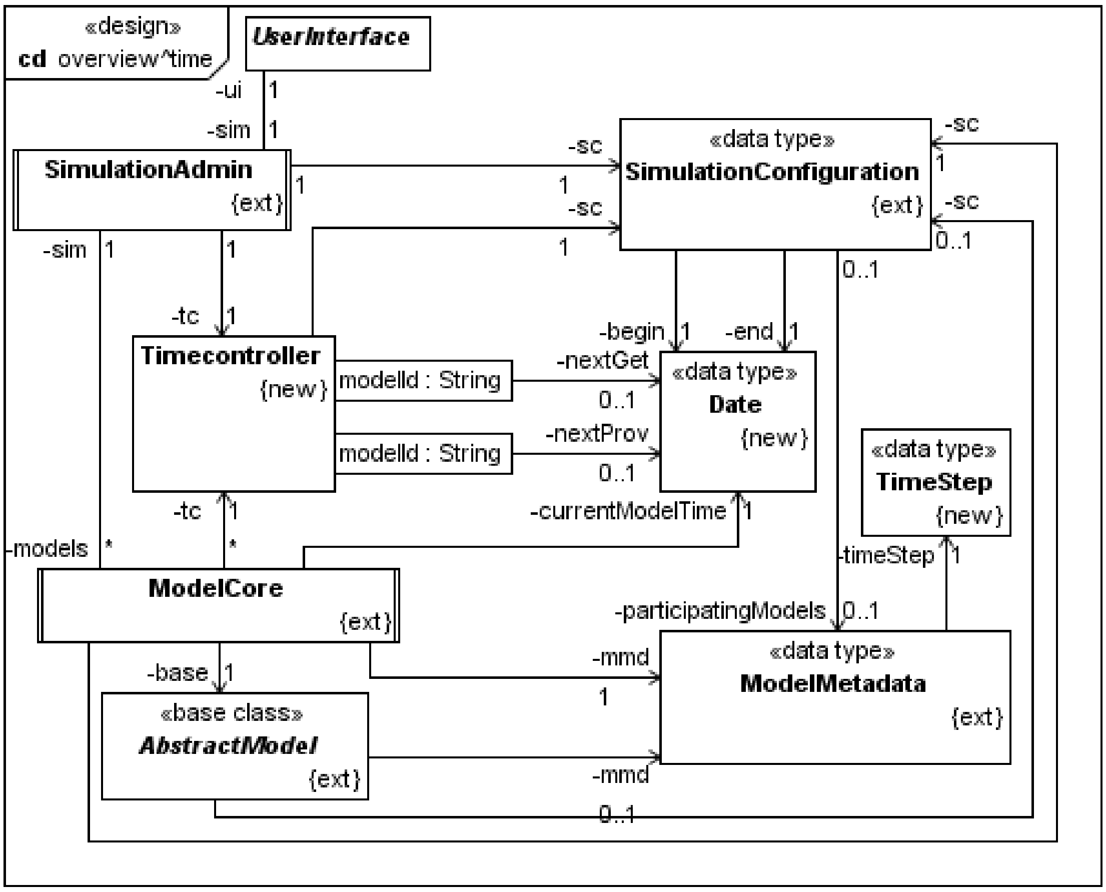

«components»
cmp Proxel^space
«component»
Proxel
ProxelTableAccess
ExceptionHandler
BasedataQuery
ProxelHandler
core
bdAdmin
PoxelTable
+getProxel(pid : Integer)
+ProxelTable(sc : SimulationConfiguration, bdAdmin : BasedataQuery)
pid : Integer
-proxelTable
1
-sc
1
-area
1
«data type»
SimulationConfiguration
{from metadata^space}
«data type»
AreaMetadata
{from metadata^space}
-proxels
0..1
«base class»
AbstractProxel
-pid : Integer
~setPid(pid : Integer)
~setProperty(p : AreaProperty, d : DataElement)
«plug-points»
# computeProxel()
«queries»
+getPid() : Integer
+getProperty(p : AreaProperty) : DataElement
p : AreaProperty
-value
1
«base class»
DataElement
{from modeldata^space}


«components»
pkg metadata^space
metadata^space
«data type»
SimulationConfiguration
-simulationId : String
+getParticipatingModels() : ModelMetadata [*]{query}
+getModelMetadata(modelId : String) : ModelMetadata{query}
+getResources() : ResourceMetadata [*]{query}
+getSimulationId() : String{query}
+getArea() : AreaMetadata{query}
+isValid() : Boolean{query}
...
-area
1
«data type»
AreaMetadata
-areaId : String
-description : String
-nrProxels : Integer
+getAreaId() : String{query}
+getDescription() : String{query}
+getNrProxels() : Integer{query}
+getProperties() : AreaProperty [*]{query}
+isValid() : Boolean{query}
0..1
-participatingModels
*
-resources
*
-properties
*
«data type»
ModelMetadata
-modelId : String
-modelClass : String
-proxelClass : String
+getModelId() : String{query}
+getProxelClass() : String{query}
+getModelClass() : String{query}
+isValid() : Boolean{query}
...
«data type»
ResourceMetadata
-resourceId : String
-description : String
-resourceType : String
-resourceLocation : String
+getResourceId() : String{query}
+getDescription() : String{query}
+getresourceType() : String{query}
+getResourceLocation() : String{query}
+isValid() : Boolean{query}
-property
1
1
«data type»
AreaProperty
-name : String
-type : String
+getName() : String{query}
+getType() : String{query}
+isValid() : Boolean{query}

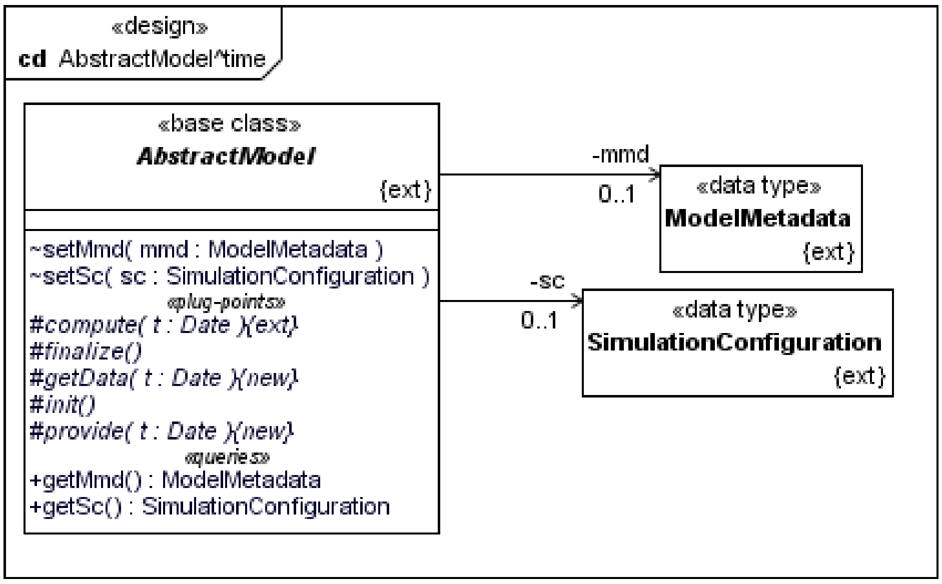

«components»
pkg modeldata^space
modeldata^space
DataTable
+getValue(pid : Integer) : DataElement{query}
pid : Integer
-value
0..1
«base class»
DataElement

A.4 View “Time Coordination”

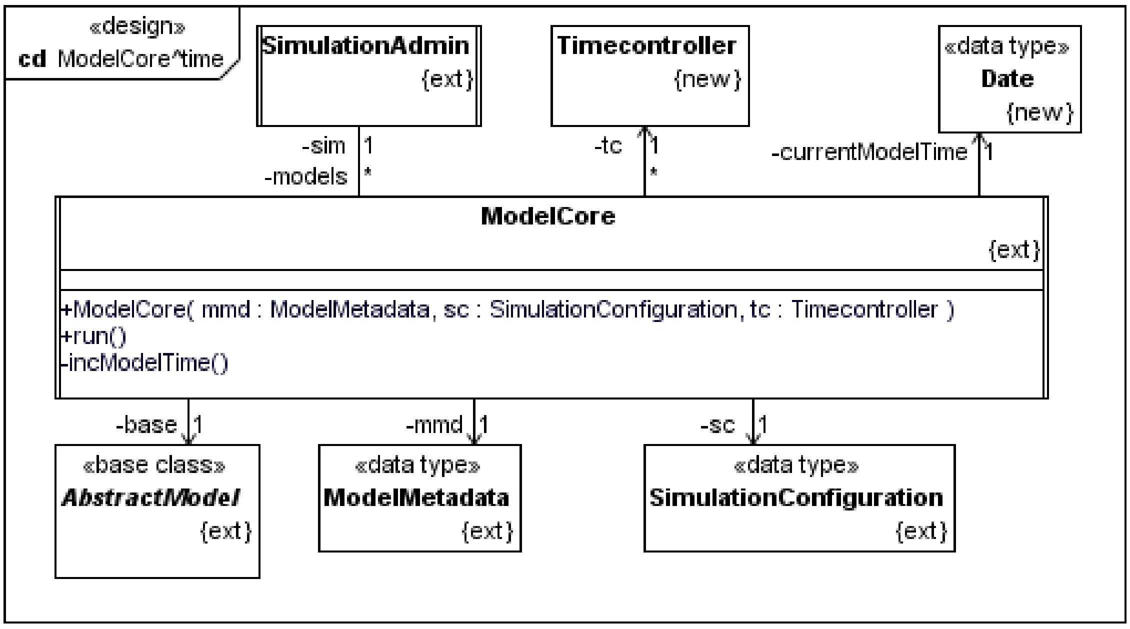
A.4.1 Structural Design



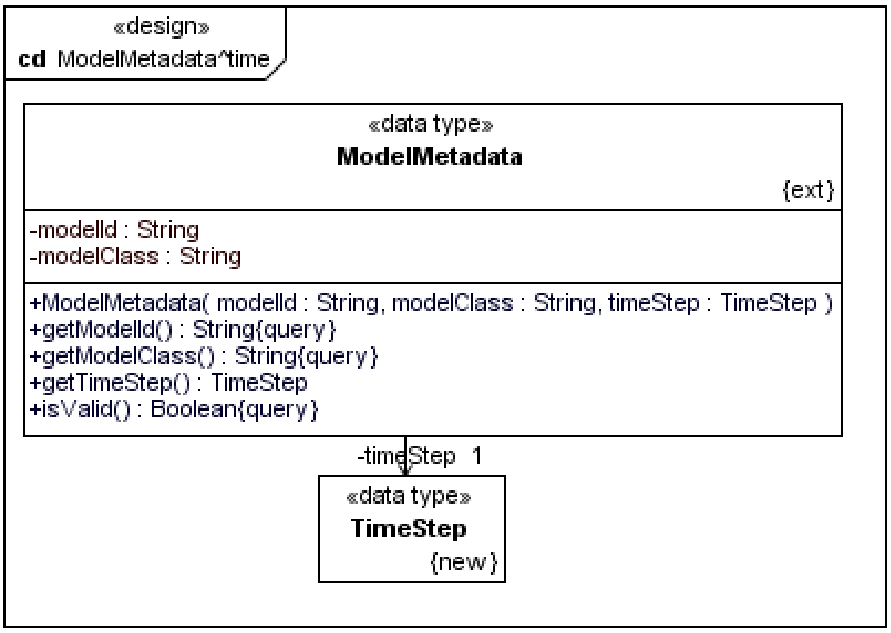
AbstractModel


«design»
cd AbstractModel^time
«base class»
AbstractModel
{ext}
~setMmd(mmd : ModelMetadata)
~setSc(sc : SimulationConfiguration)
«plug-points»
#compute(t : Date){ext}
#finalize()
#getData(t : Date){new}
#init()
#provide(t : Date){new}
«queries»
+getMmd() : ModelMetadata
+getSc() : SimulationConfiguration
-mmd
0..1
«data type»
ModelMetadata
{ext}
-sc
0..1
«data type»
SimulationConfiguration
{ext}

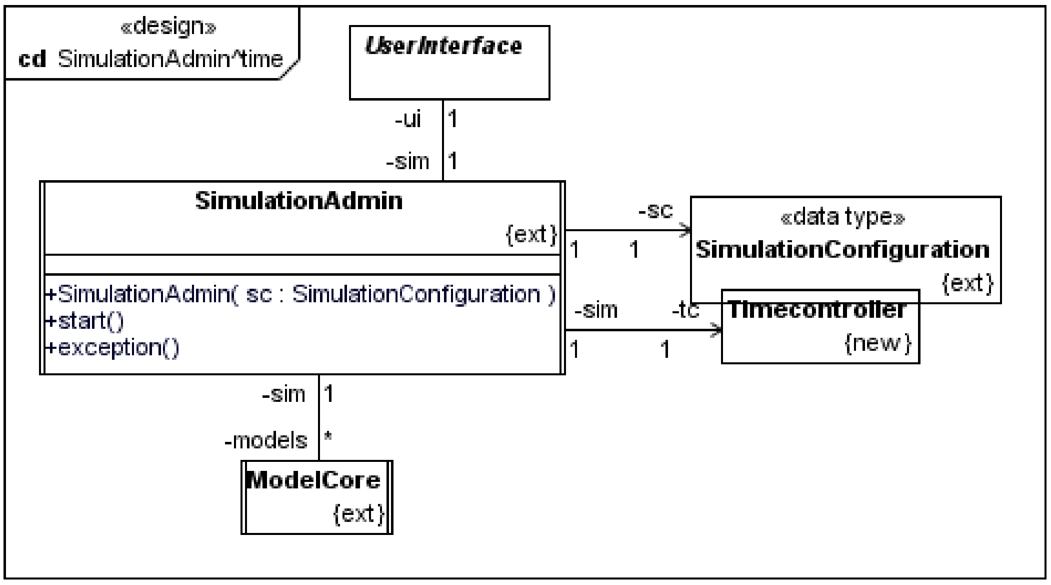
ModelCore



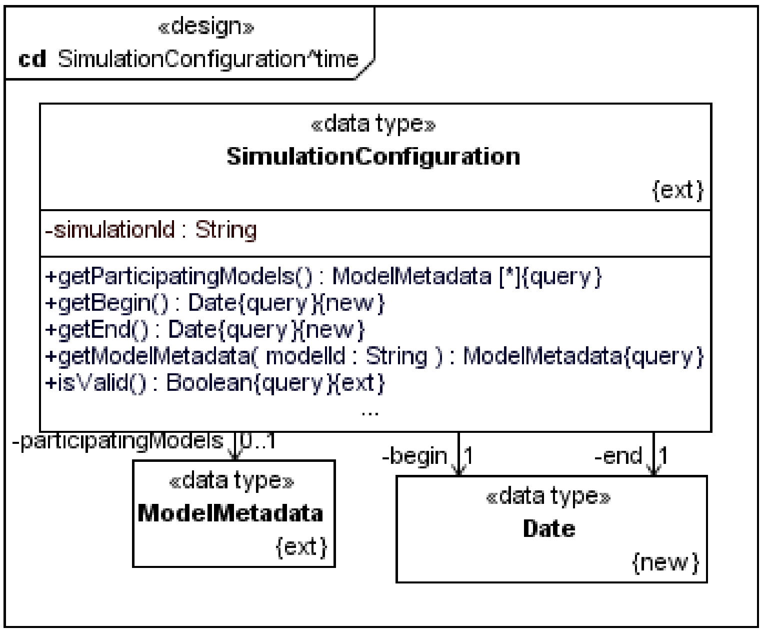
ModelMetadata


«design»
cd ModelMetadata^time
«data type»
ModelMetadata
{ext}
-modelId : String
-modelClass : String
+ModelMetadata(modelId : String, modelClass : String, timeStep : TimeStep)
+getModelId() : String{query}
+getModelClass() : String{query}
+getTimeStep() : TimeStep
+isValid() : Boolean{query}
-timeStep 1
«data type»
TimeStep
{new}

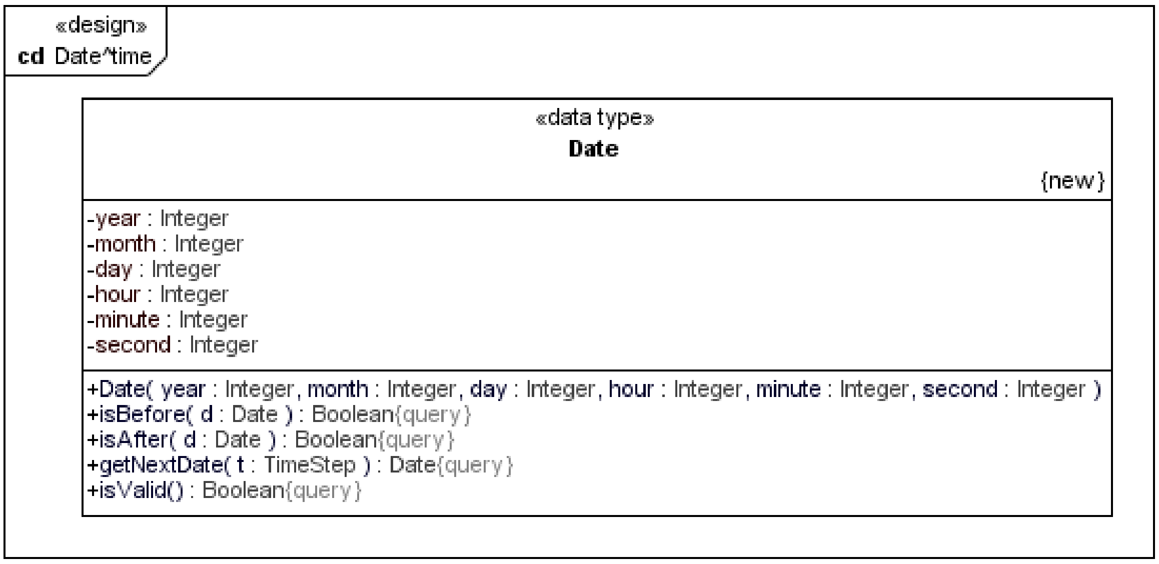
SimulationAdmin



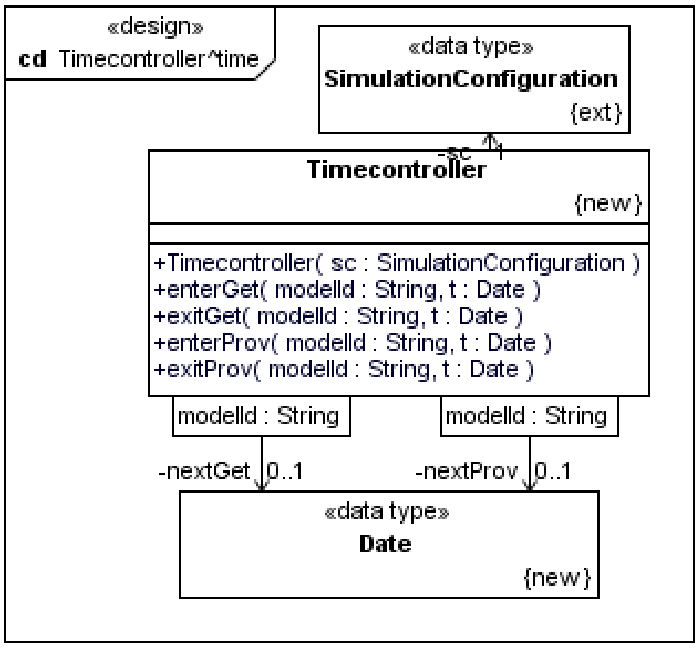
SimulationConfiguration


«design»
cd SimulationConfiguration^time
«data type»
SimulationConfiguration
{ext}
-simulationId : String
+getParticipatingModels() : ModelMetadata [*]{query}
+getBegin() : Date{query}{new}
+getEnd() : Date{query}{new}
+getModelMetadata(modelId : String) : ModelMetadata{query}
+isValid() : Boolean{query}{ext}
...
-participatingModels
0..1
«data type»
ModelMetadata
{ext}
-begin
1
-end
1
«data type»
Date
{new}

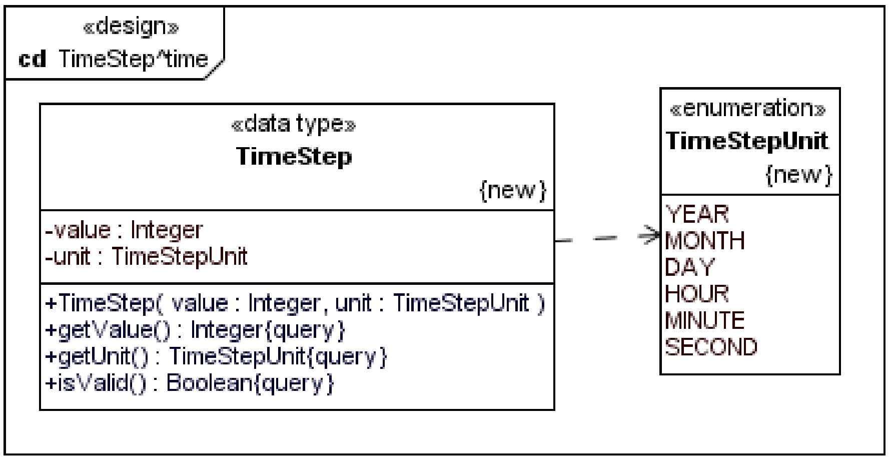
Date


«design»
cd Date^time
«data type»
Date
{new}
-year : Integer
-month : Integer
-day : Integer
-hour : Integer
-minute : Integer
-second : Integer
+Date(year : Integer, month : Integer, day : Integer, hour : Integer, minute : Integer, second : Integer)
+isBefore(d : Date) : Boolean{query}
+isAfter(d : Date) : Boolean{query}
+getNextDate(t : TimeStep) : Date{query}
+isValid() : Boolean{query}

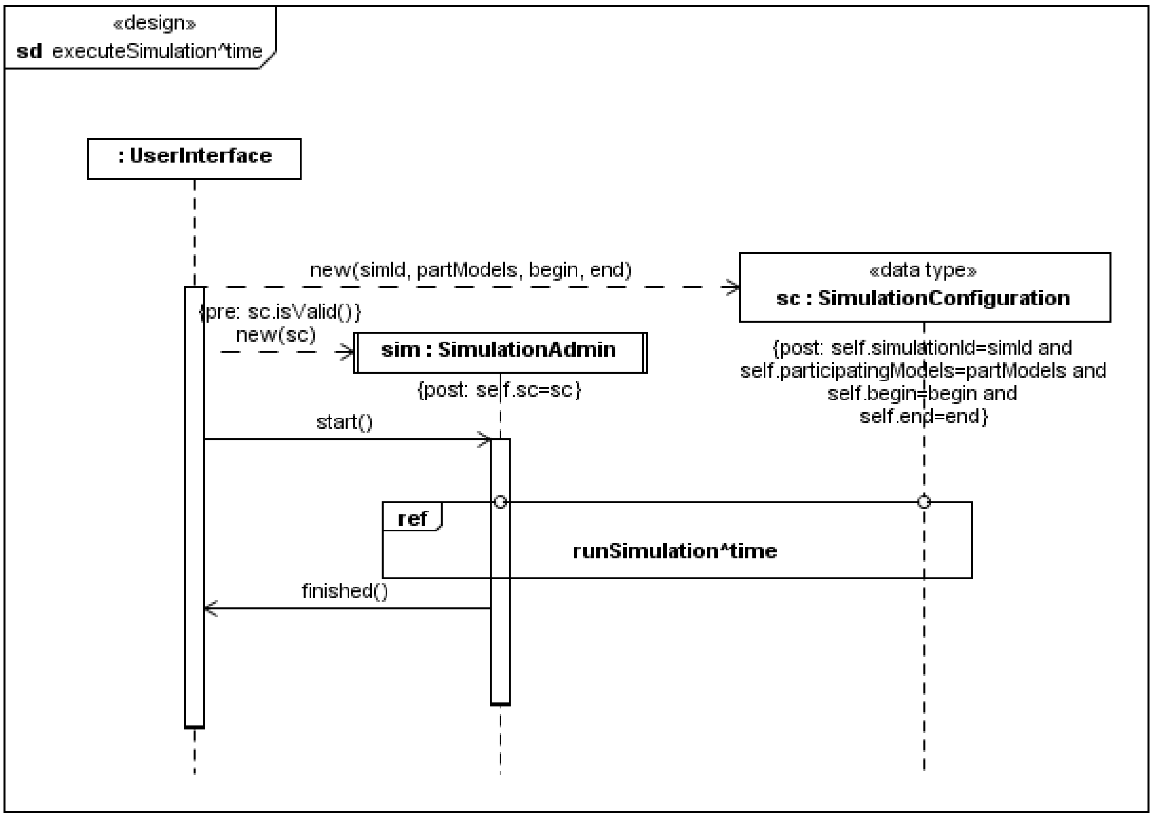
Timecontroller

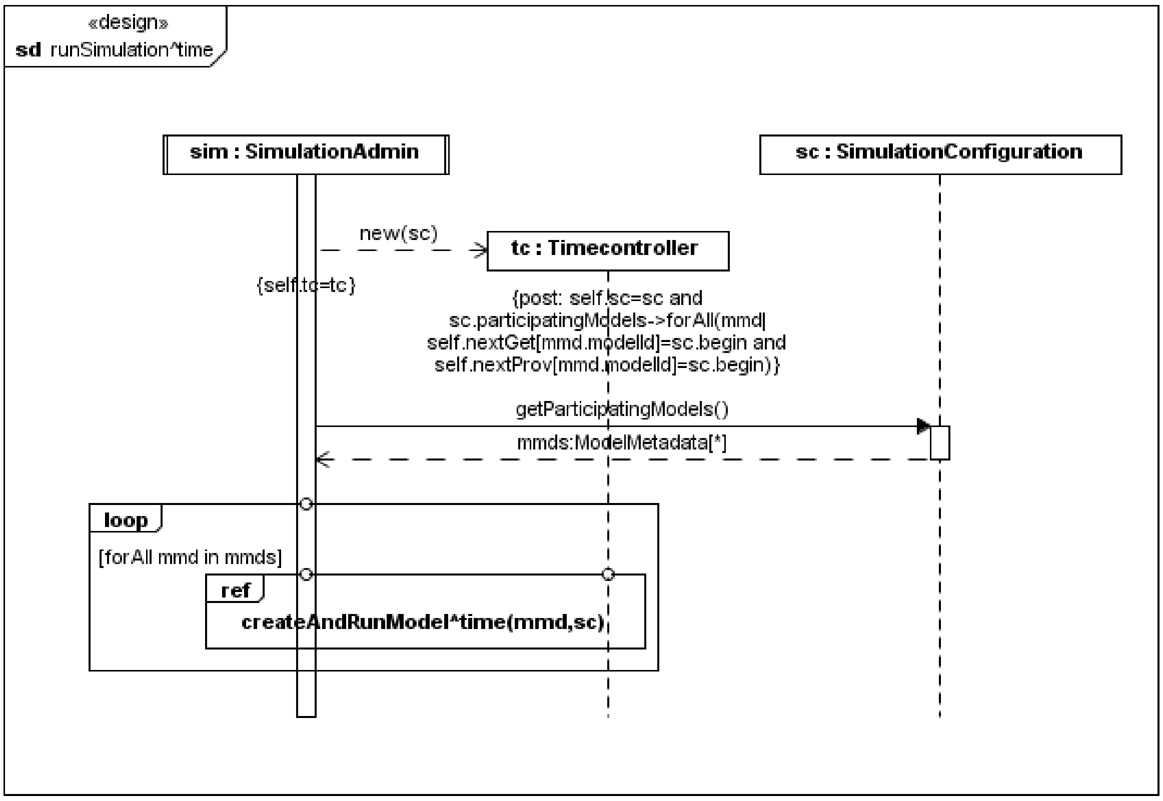

«design»
cd Timecontroller^time
«data type»
SimulationConfiguration
{ext}
-sc
1
Timecontroller
{new}
+Timecontroller(sc : SimulationConfiguration)
+enterGet(modelId : String, t : Date)
+exitGet(modelId : String, t : Date)
+enterProv(modelId : String, t : Date)
+exitProv(modelId : String, t : Date)
modelId : String
modelId : String
-nextGet
0..1
-nextProv
0..1
«data type»
Date
{new}

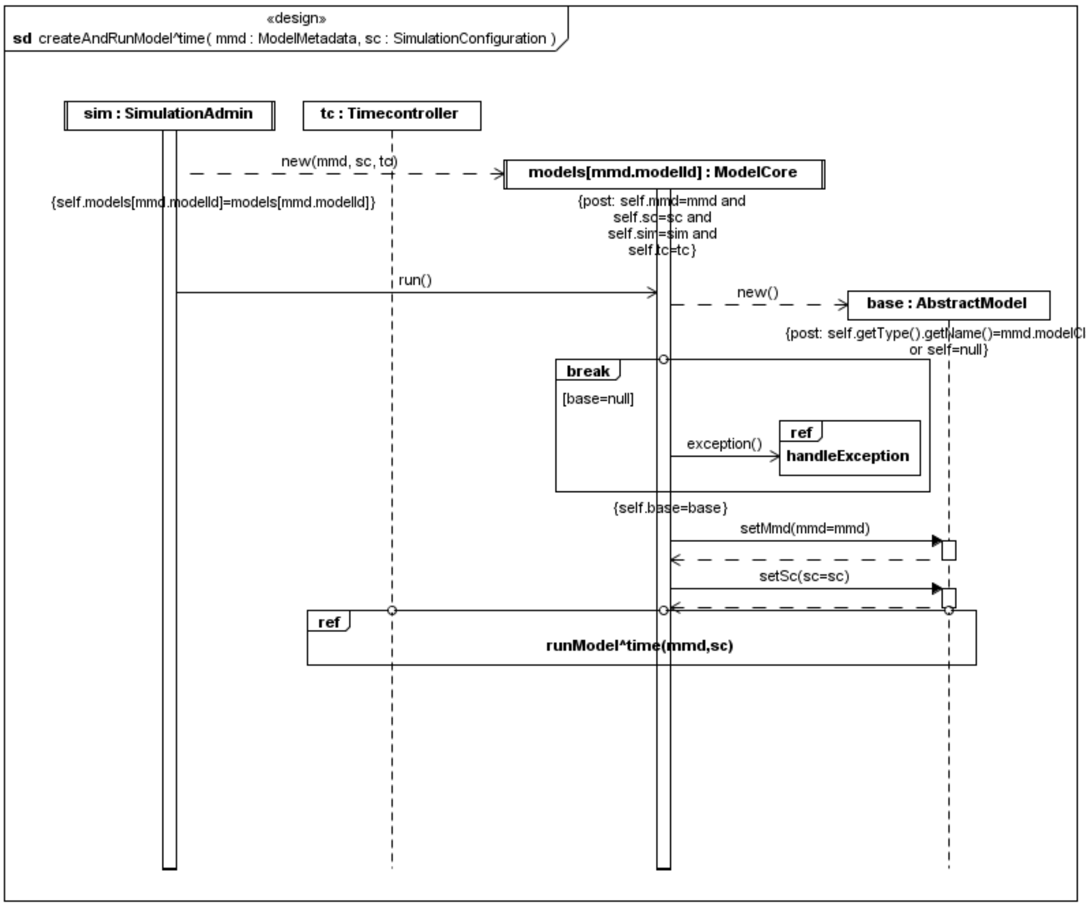
TimeStep

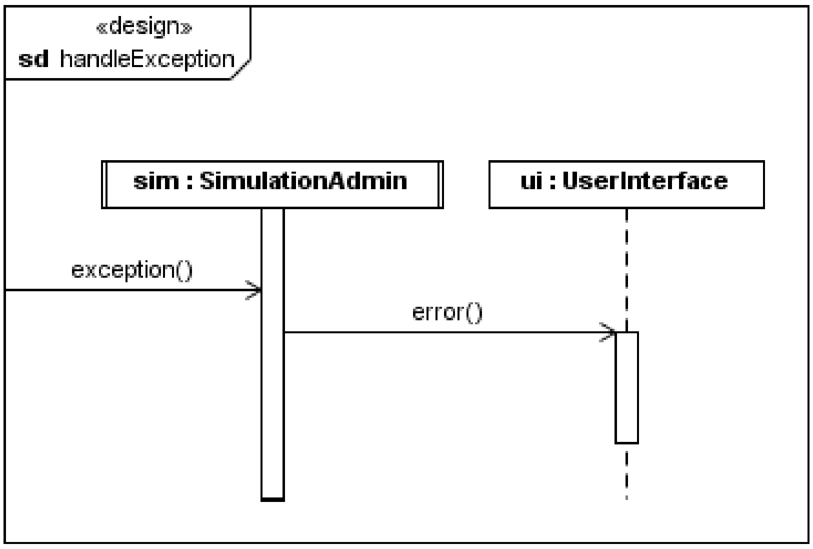


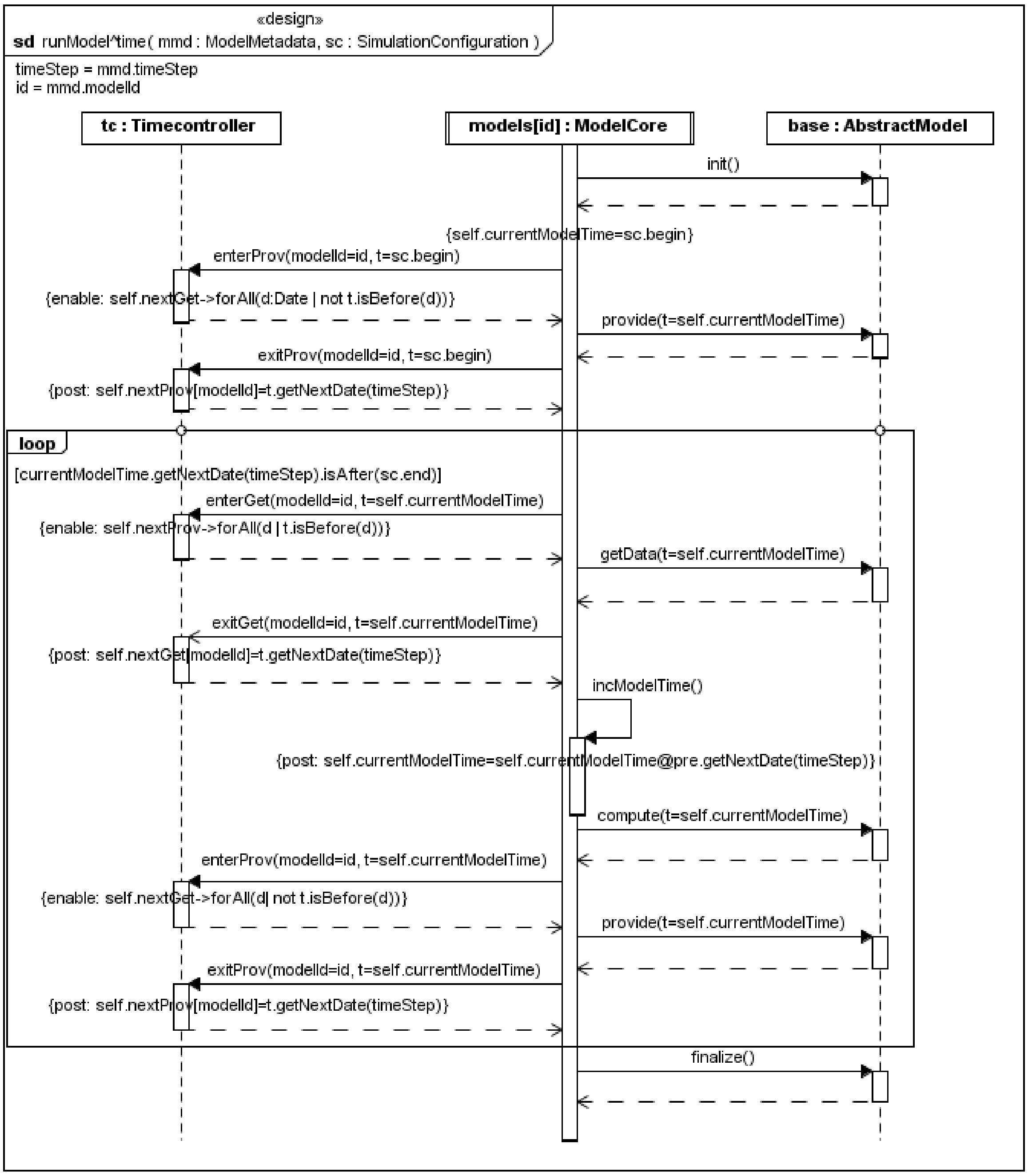
A.4.2 Behavioural Design



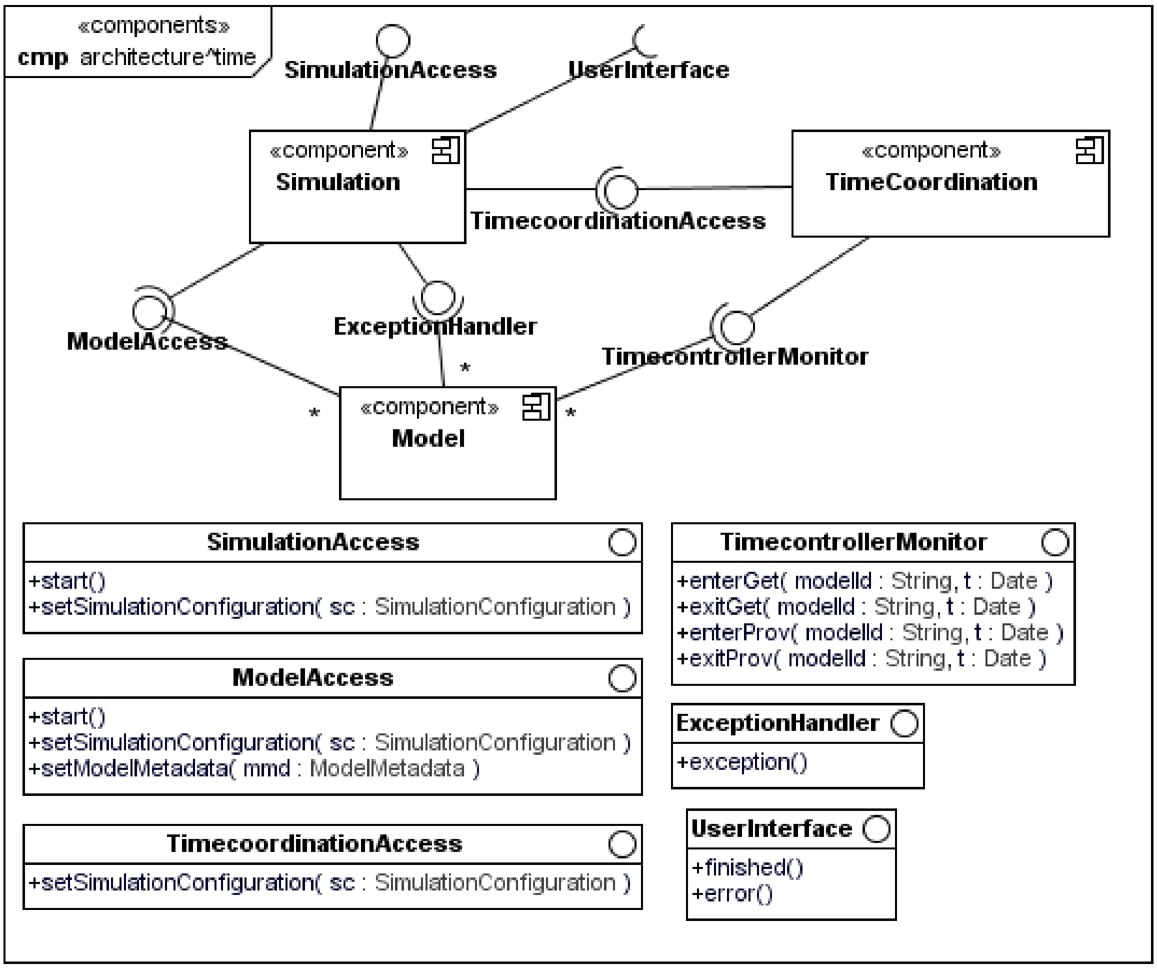

«design»
sd runSimulation^time
sim : SimulationAdmin
sc : SimulationConfiguration
new(sc)
tc : Timecontroller
{self.tc=tc}
{post: self.sc=sc and
sc.participatingModels->forAll(mmd|
self.nextGet[mmd.modelId]=sc.begin and
self.nextProv[mmd.modelId]=sc.begin)}
getParticipatingModels()
mmds:ModelMetadata[*]
loop
[forAll mmd in mmds]
ref
createAndRunModel^time(mmd,sc)

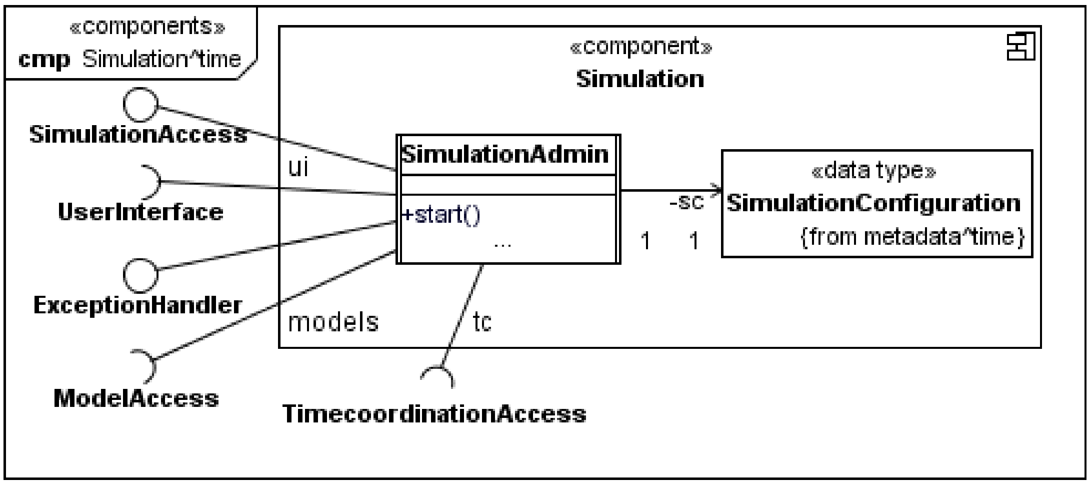

«design»
sd createAndRunModel^time(mmd : ModelMetadata, sc : SimulationConfiguration)
sim : SimulationAdmin
tc : Timecontroller
new(mmd, sc, tc)
models[mmd.modelId] : ModelCore
{self.models[mmd.modelId]=models[mmd.modelId]}
{post: self.mmd=mmd and self.sc=sc and self.sim=sim and self.tc=tc}
run()
new()
base : AbstractModel
{post: self.getType().getName()=mmd.modelClass or self=null}
break
[base=null]
exception()
ref
handleException
{self.base=base}
setMmd(mmd=mmd)
setSc(sc=sc)
ref
runModel^time(mmd,sc)

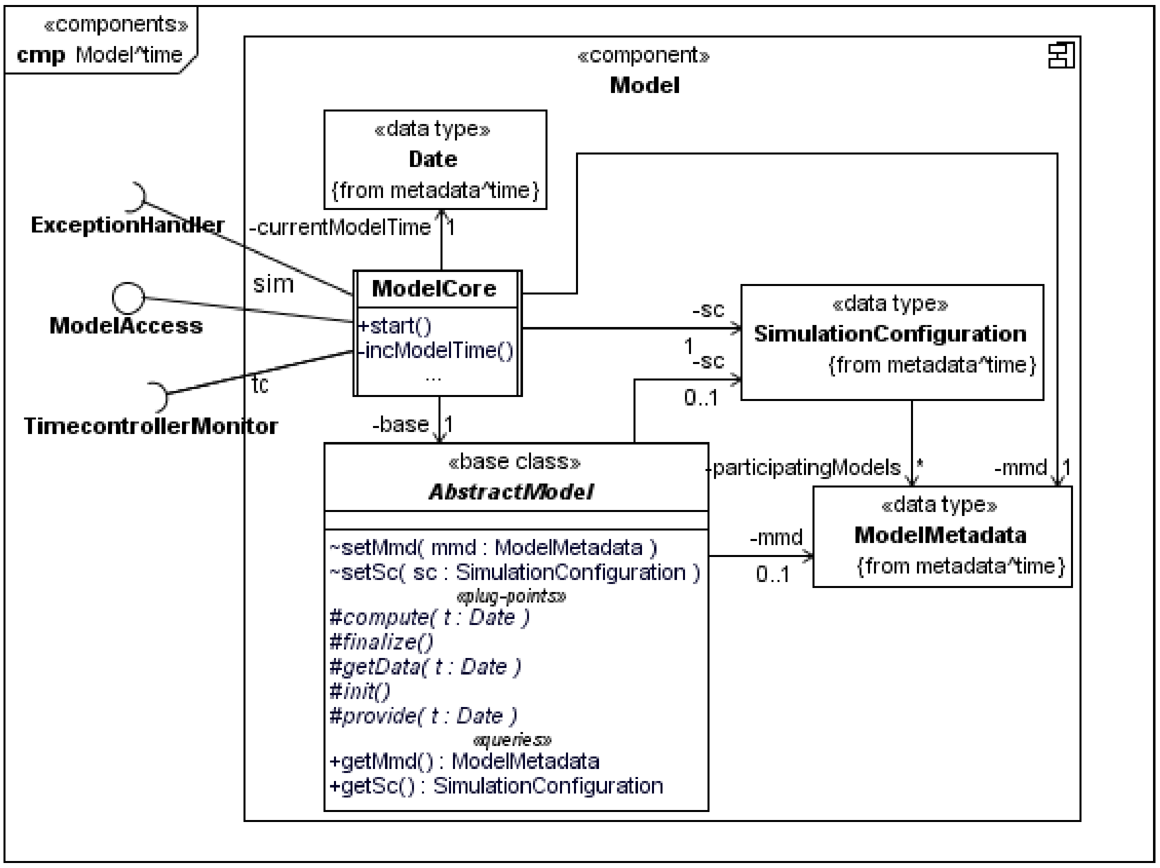

«design»
sd handleException
sim : SimulationAdmin
ui : UserInterface
exception()
error()

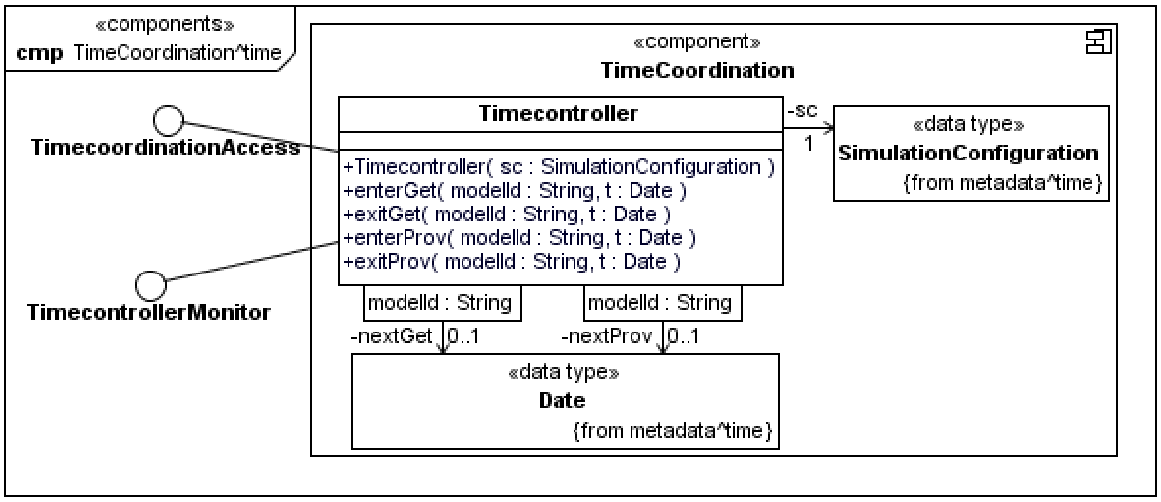

«design»
sd runModel^time(mmd : ModelMetadata, sc : SimulationConfiguration)
timeStep = mmd.timeStep
id = mmd.modelId
tc : Timecontroller
models[id] : ModelCore
base : AbstractModel
init()
{self.currentModelTime=sc.begin}
enterProv(modelId=id, t=sc.begin)
{enable: self.nextGet->forAll(d:Date | not t.isBefore(d))}
provide(t=self.currentModelTime)
exitProv(modelId=id, t=sc.begin)
{post: self.nextProv[modelId]=t.getNextDate(timeStep)}
loop
[currentModelTime.getNextDate(timeStep).isAfter(sc.end)]
enterGet(modelId=id, t=self.currentModelTime)
{enable: self.nextProv->forAll(d | t.isBefore(d))}
getData(t=self.currentModelTime)
exitGet(modelId=id, t=self.currentModelTime)
{post: self.nextGet[modelId]=t.getNextDate(timeStep)}
incModelTime()
{post: self.currentModelTime=self.currentModelTime@pre.getNextDate(timeStep)}
compute(t=self.currentModelTime)
enterProv(modelId=id, t=self.currentModelTime)
{enable: self.nextGet->forAll(d| not t.isBefore(d))}
provide(t=self.currentModelTime)
exitProv(modelId=id, t=self.currentModelTime)
{post: self.nextProv[modelId]=t.getNextDate(timeStep)}
finalize()

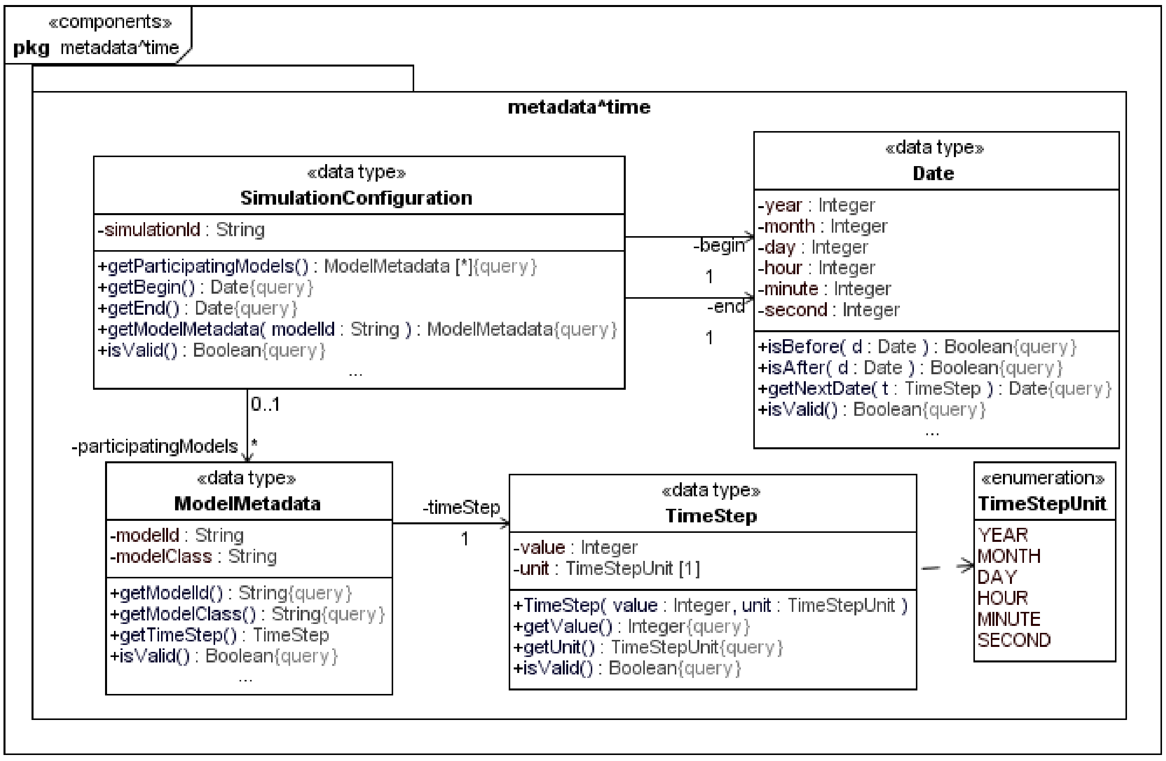
A.4.3 Components


«components»
cmp architecture^time
SimulationAccess
UserInterface
«component»
Simulation
«component»
TimeCoordination
TimecoordinationAccess
ModelAccess
ExceptionHandler
TimecontrollerMonitor
*
*
*
«component»
Model
SimulationAccess
+start()
+setSimulationConfiguration(sc : SimulationConfiguration)
ModelAccess
+start()
+setSimulationConfiguration(sc : SimulationConfiguration)
+setModelMetadata(mmd : ModelMetadata)
TimecoordinationAccess
+setSimulationConfiguration(sc : SimulationConfiguration)
TimecontrollerMonitor
+enterGet(modelId : String, t : Date)
+exitGet(modelId : String, t : Date)
+enterProv(modelId : String, t : Date)
+exitProv(modelId : String, t : Date)
ExceptionHandler
+exception()
UserInterface
+finished()
+error()



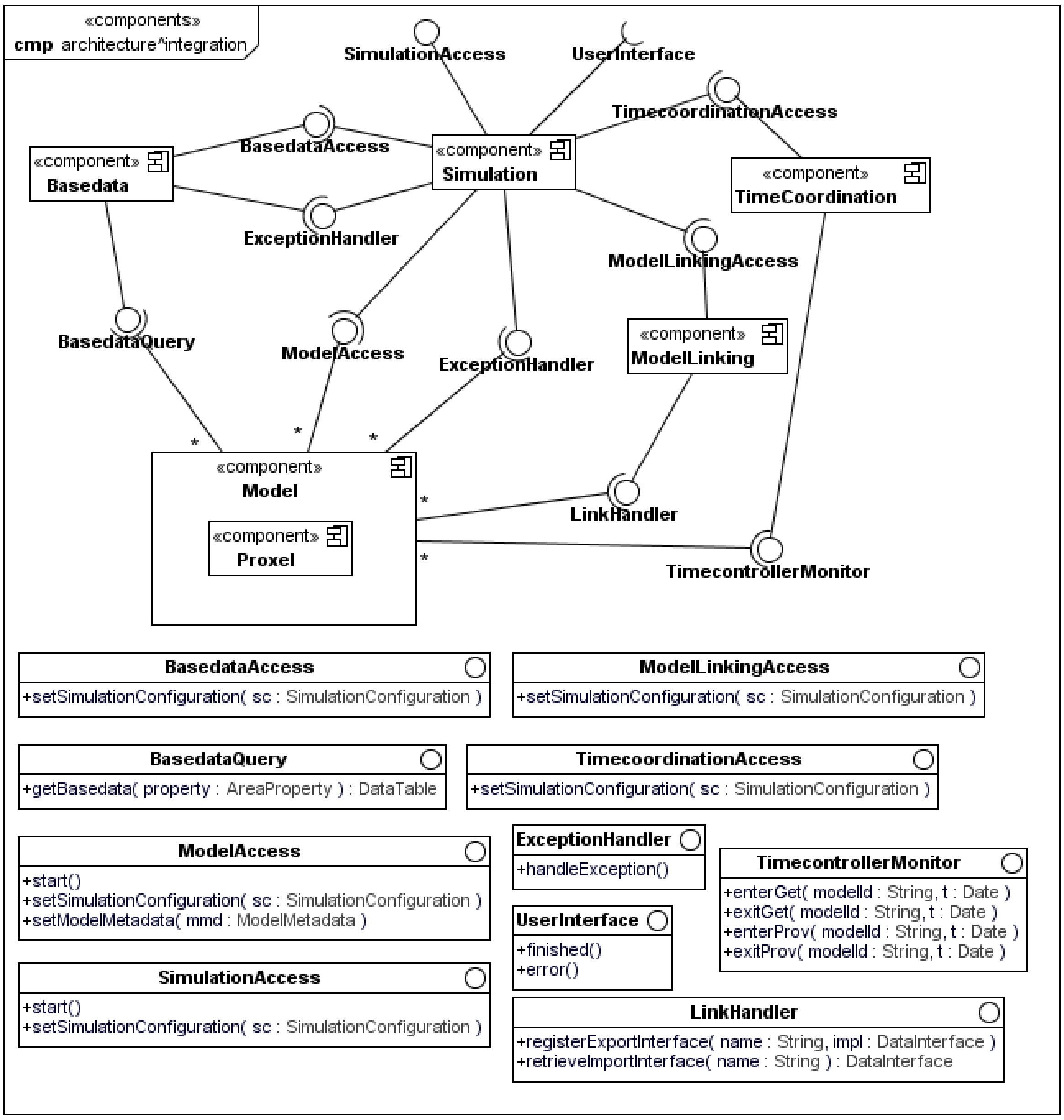

«components»
cmp Model^time
«component»
Model
«data type»
Date
{from metadata^time}
-currentModelTime
1
ExceptionHandler
sim
ModelAccess
TimecontrollerMonitor
tc
ModelCore
+start()
-incModelTime()
...
-sc
1
-sc
0..1
«data type»
SimulationConfiguration
{from metadata^time}
-base
1
«base class»
AbstractModel
~setMmd(mmd : ModelMetadata)
~setSc(sc : SimulationConfiguration)
«plug-points»
#compute(t : Date)
#finalize()
#getData(t : Date)
#init()
#provide(t : Date)
«queries»
+getMmd() : ModelMetadata
+getSc() : SimulationConfiguration
-participatingModels
*
-mmd
1
-mmd
0..1
«data type»
ModelMetadata
{from metadata^time}

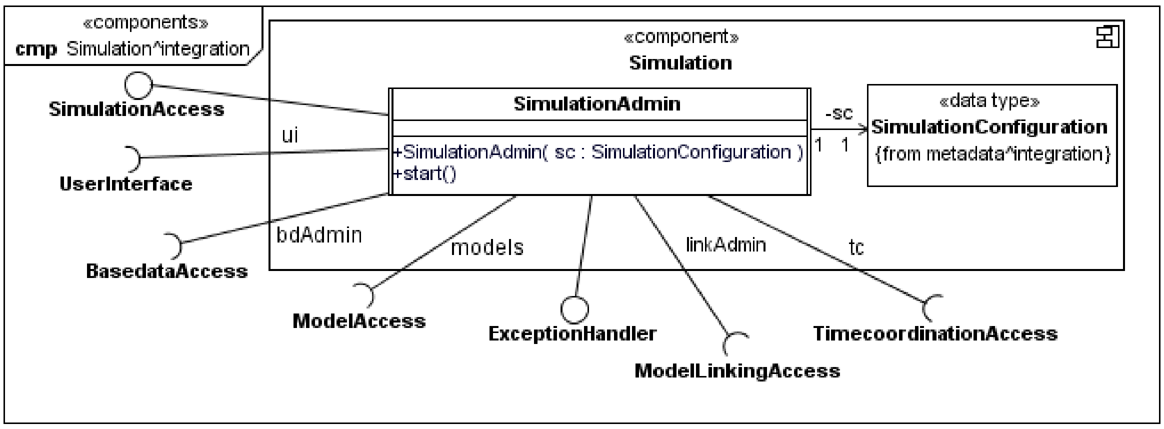

«components»
cmp TimeCoordination^time
«component»
TimeCoordination
TimecoordinationAccess
TimecontrollerMonitor
Timecontroller
+Timecontroller(sc : SimulationConfiguration)
+enterGet(modelId : String, t : Date)
+exitGet(modelId : String, t : Date)
+enterProv(modelId : String, t : Date)
+exitProv(modelId : String, t : Date)
-sc
1
«data type»
SimulationConfiguration
{from metadata^time}
modelId : String
modelId : String
-nextGet
0..1
-nextProv
0..1
«data type»
Date
{from metadata^time}

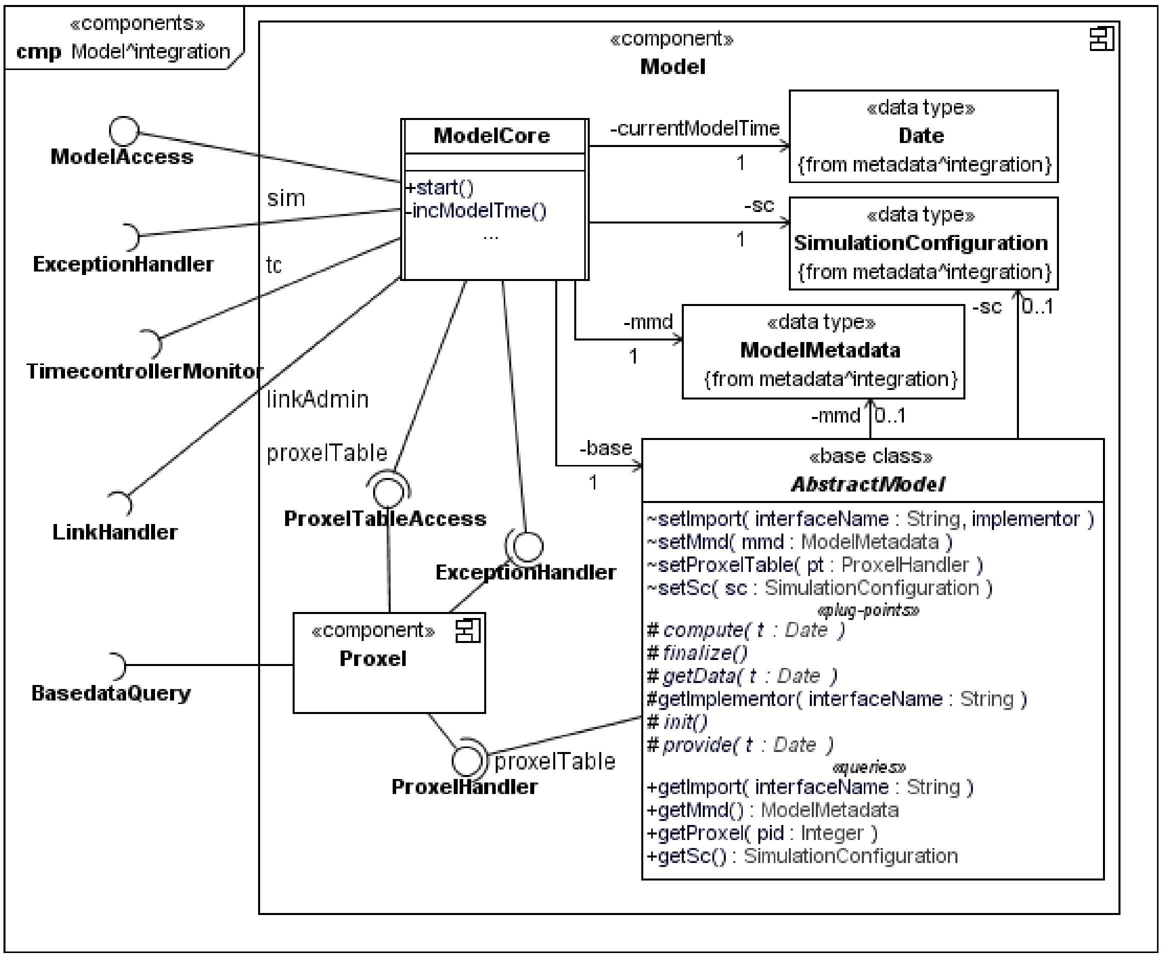

«components»
pkg metadata^time
metadata^time
«data type»
SimulationConfiguration
-simulationId : String
+getParticipatingModels() : ModelMetadata [*]{query}
+getBegin() : Date{query}
+getEnd() : Date{query}
+getModelMetadata(modelId : String) : ModelMetadata{query}
+isValid() : Boolean{query}
...
-begin
1
-end
1
«data type»
Date
-year : Integer
-month : Integer
-day : Integer
-hour : Integer
-minute : Integer
-second : Integer
+isBefore(d : Date) : Boolean{query}
+isAfter(d : Date) : Boolean{query}
+getNextDate(t : TimeStep) : Date{query}
+isValid() : Boolean{query}
...
0..1
-participatingModels
*
«data type»
ModelMetadata
-modelId : String
-modelClass : String
+getModelId() : String{query}
+getModelClass() : String{query}
+getTimeStep() : TimeStep
+isValid() : Boolean{query}
...
-timeStep
1
«data type»
TimeStep
-value : Integer
-unit : TimeStepUnit [1]
+TimeStep(value : Integer, unit : TimeStepUnit)
+getValue() : Integer{query}
+getUnit() : TimeStepUnit{query}
+isValid() : Boolean{query}
«enumeration»
TimeStepUnit
YEAR
MONTH
DAY
HOUR
MINUTE
SECOND

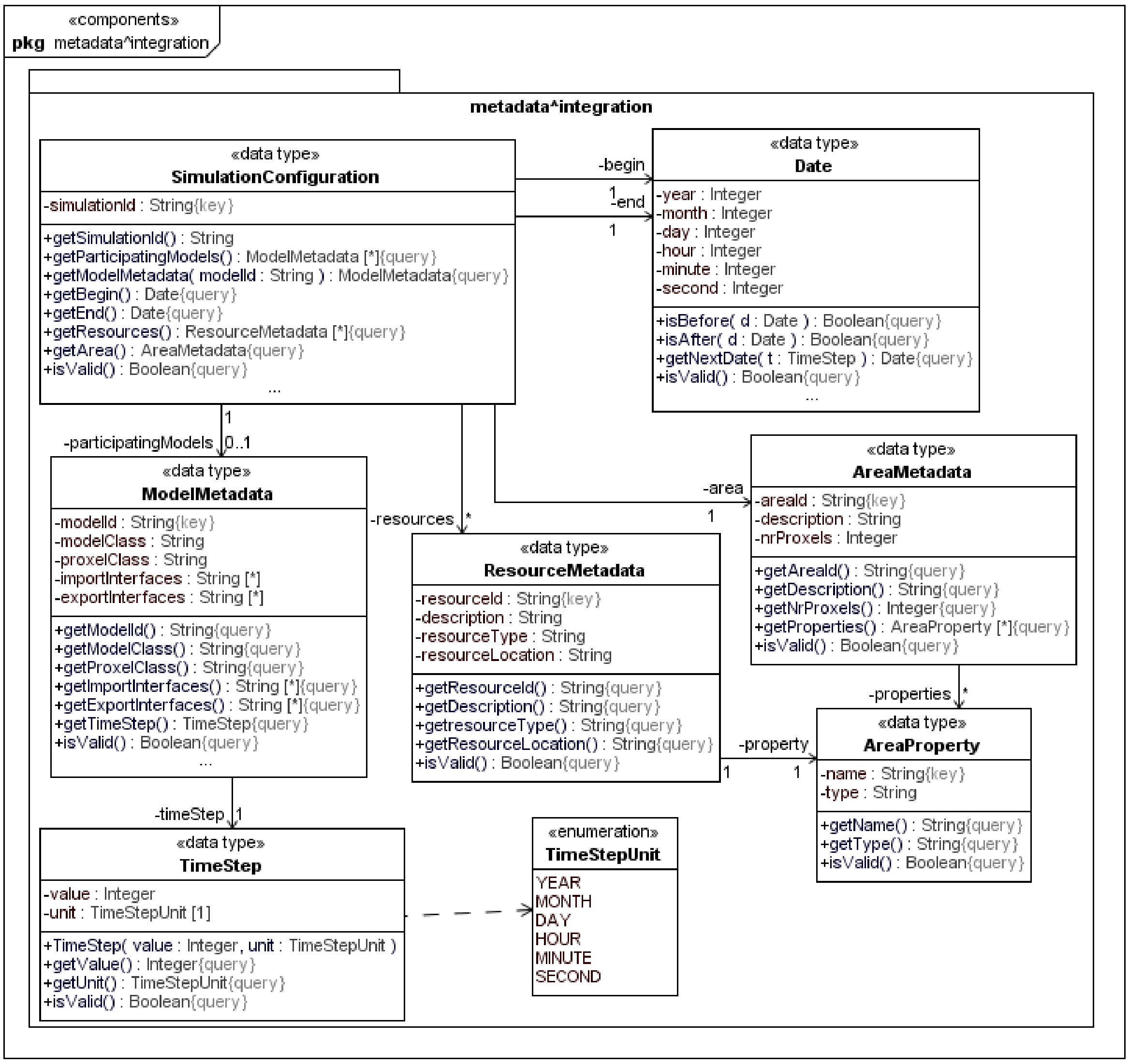
A.5 Integration

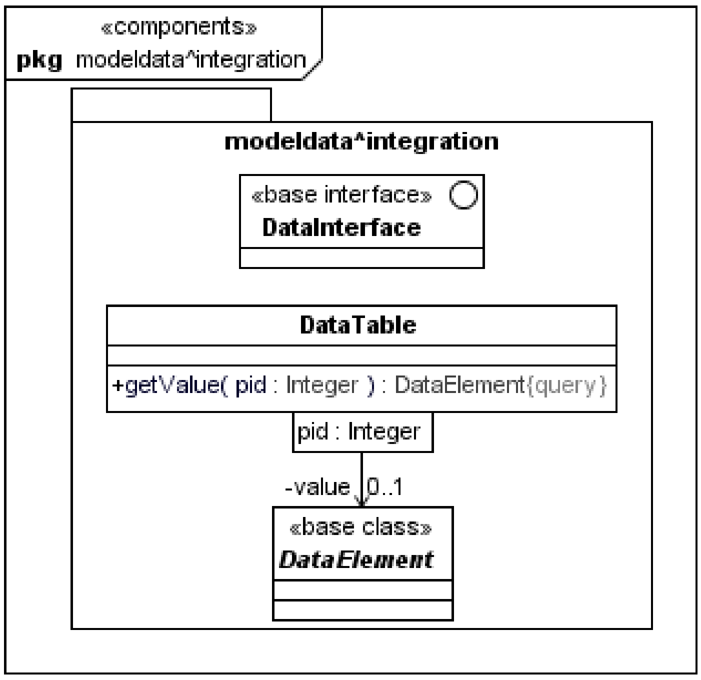
A.5.1 Component Integration


«components»
cmp architecture^integration
SimulationAccess
UserInterface
TimecoordinationAccess
BasedataAccess
«component» Basedata
«component» Simulation
«component» TimeCoordination
ExceptionHandler
ModellLinkingAccess
BasedataQuery
ModelAccess
ExceptionHandler
«component» ModellLinking
*
*
*
«component» Model
«component» Proxel
*
*
LinkHandler
TimecontrollerMonitor
BasedataAccess
+setSimulationConfiguration(sc : SimulationConfiguration)
ModellLinkingAccess
+setSimulationConfiguration(sc : SimulationConfiguration)
BasedataQuery
+getBasedata(property : AreaProperty) : DataTable
TimecoordinationAccess
+setSimulationConfiguration(sc : SimulationConfiguration)
ModelAccess
+start()
+setSimulationConfiguration(sc : SimulationConfiguration)
+setModelMetadata(mmd : ModelMetadata)
ExceptionHandler
+handleException()
TimecontrollerMonitor
+enterGet(modelId : String, t : Date)
+exitGet(modelId : String, t : Date)
+enterProv(modelId : String, t : Date)
+exitProv(modelId : String, t : Date)
UserInterface
+finished()
+error()
SimulationAccess
+start()
+setSimulationConfiguration(sc : SimulationConfiguration)
LinkHandler
+registerExportInterface(name : String, impl : DataInterface)
+retrieveImportInterface(name : String) : DataInterface

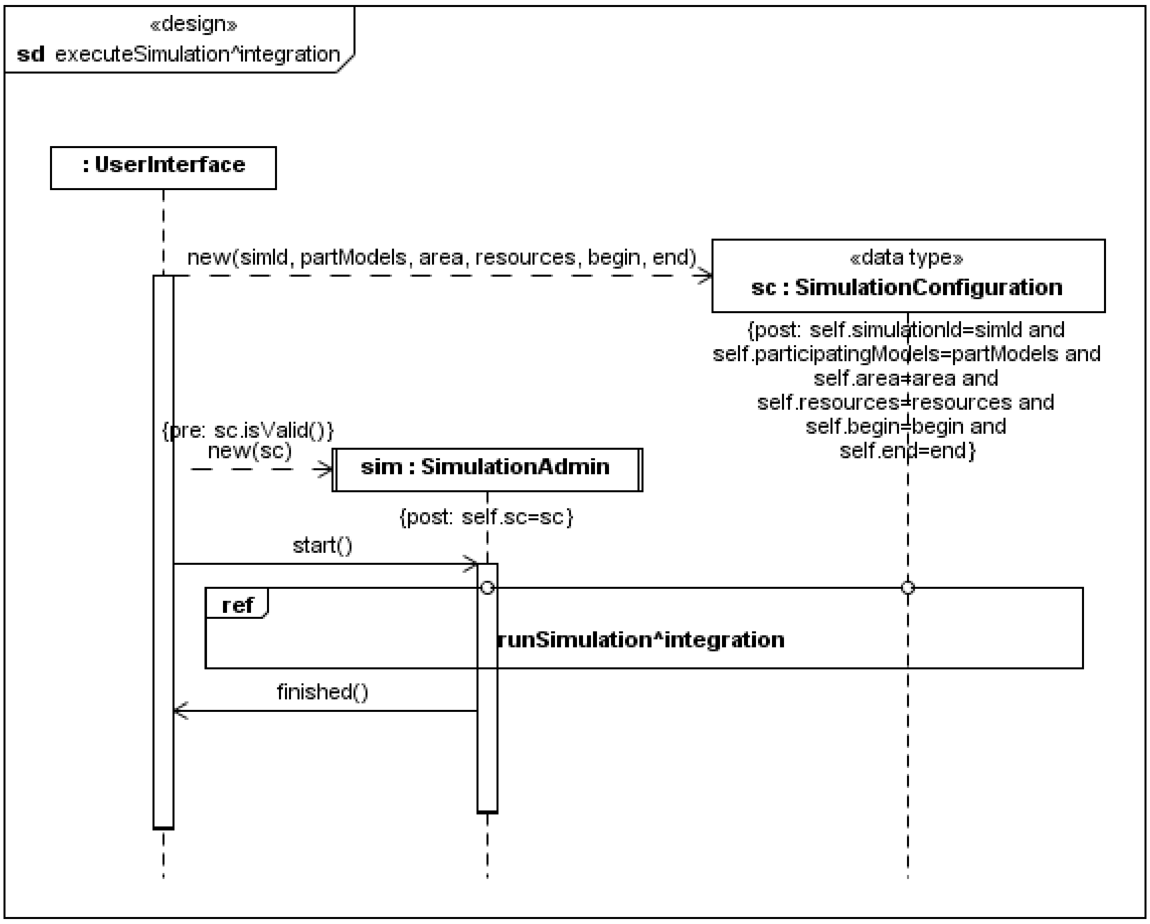

«components»
cmp Simulation^integration
«component»
Simulation
SimulationAdmin
+SimulationAdmin(sc : SimulationConfiguration)
+start()
-sc
1
1
«data type»
SimulationConfiguration
{from metadata^integration}
SimulationAccess
ui
UserInterface
bdAdmin
BasedataAccess
models
linkAdmin
tc
ModelAccess
ExceptionHandler
ModelLinkingAccess
TimecoordinationAccess

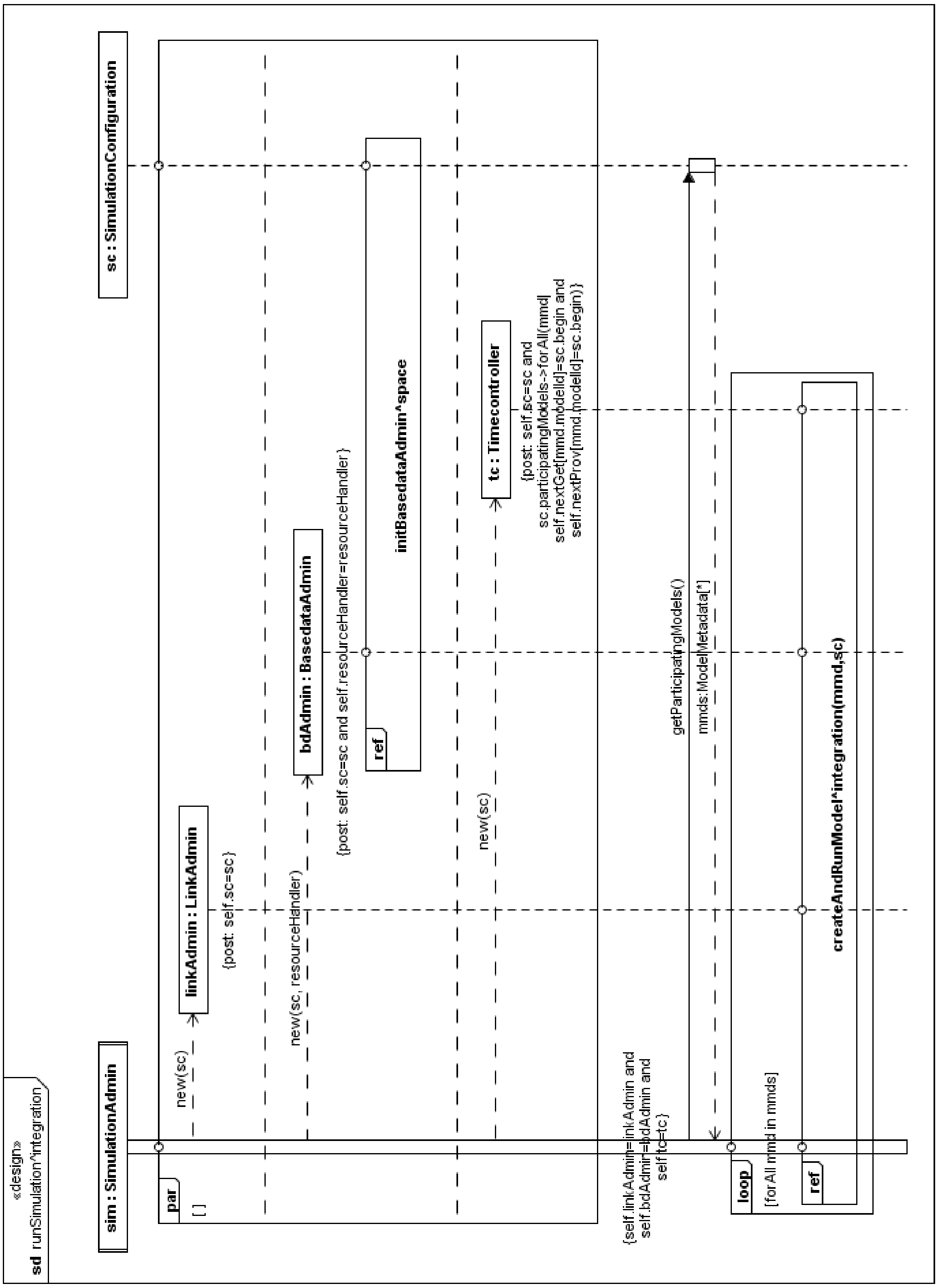

«components»
cmp Model^integration
«component»
Model
ModelCore
+start()
-incModelTme()
...
-currentModelTime
1
«data type»
Date
{from metadata^integration}
-sc
1
«data type»
SimulationConfiguration
{from metadata^integration}
-mmd
1
«data type»
ModelMetadata
{from metadata^integration}
-sc
0..1
-mmd
0..1
-base
1
«base class»
AbstractModel
~setImport(interfaceName : String, implementor)
~setMmd(mmd : ModelMetadata)
~setProxelTable(pt : ProxelHandler)
~setSc(sc : SimulationConfiguration)
«plug-points»
# compute(t : Date)
# finalize()
# getData(t : Date)
#getImplementor(interfaceName : String)
# init()
# provide(t : Date)
«queries»
+getImport(interfaceName : String)
+getMmd() : ModelMetadata
+getProxel(pid : Integer)
+getSc() : SimulationConfiguration
ModelAccess
sim
ExceptionHandler
tc
TimecontrollerMonitor
linkAdmin
LinkHandler
proxelTable
ProxelTableAccess
ExceptionHandler
«component»
Proxel
BasedataQuery
proxelTable
ProxelHandler

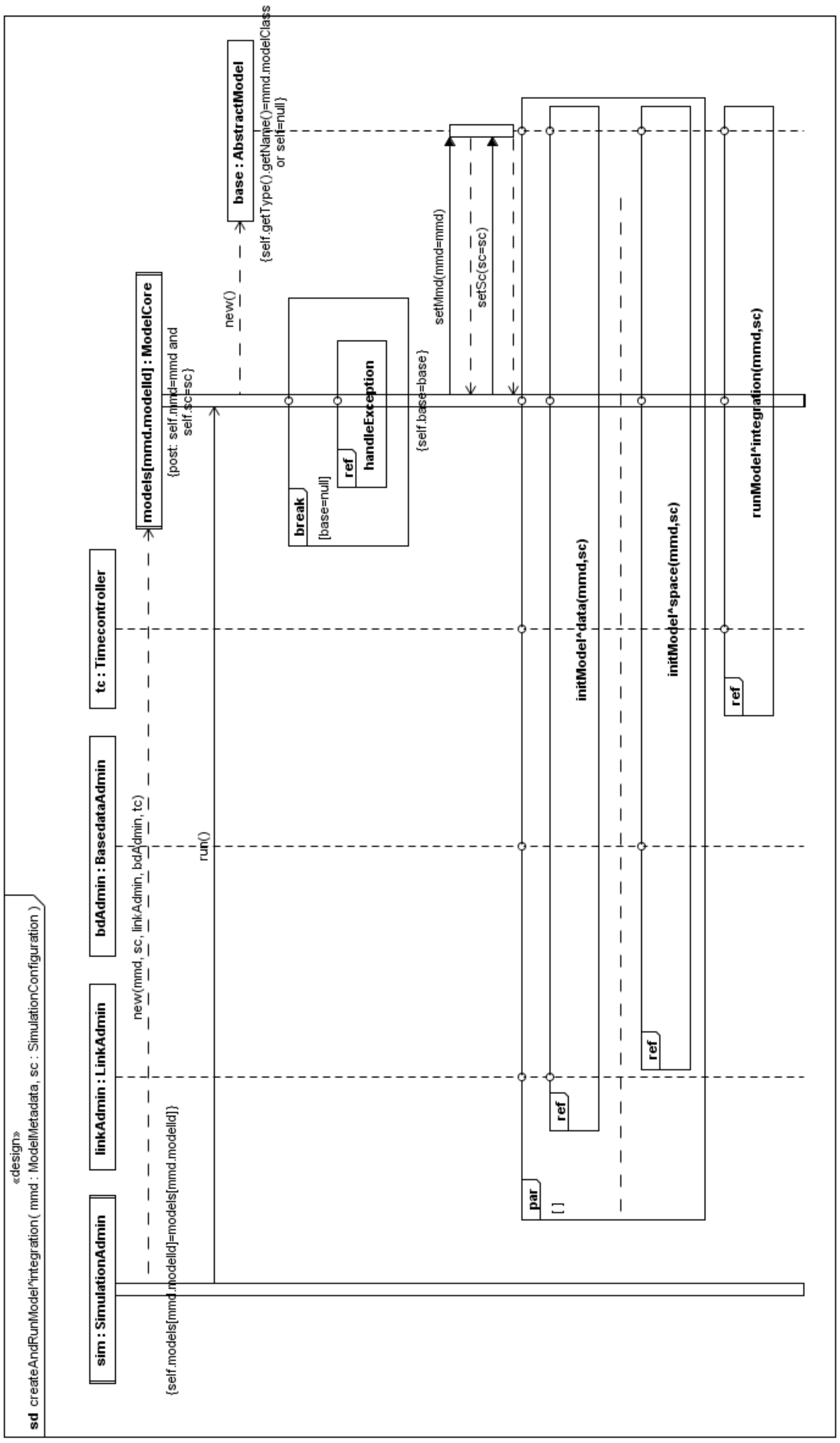

«components»
pkg metadata^integration
metadata^integration
«data type»
SimulationConfiguration
-simulationId : String{key}
+getSimulationId() : String
+getParticipatingModels() : ModelMetadata [*]{query}
+getModelMetadata(modelId : String) : ModelMetadata{query}
+getBegin() : Date{query}
+getEnd() : Date{query}
+getResources() : ResourceMetadata [*]{query}
+getArea() : AreaMetadata{query}
+isValid() : Boolean{query}
...
-begin
1
-end
1
«data type»
Date
-year : Integer
-month : Integer
-day : Integer
-hour : Integer
-minute : Integer
-second : Integer
+isBefore(d : Date) : Boolean{query}
+isAfter(d : Date) : Boolean{query}
+getNextDate(t : TimeStep) : Date{query}
+isValid() : Boolean{query}
...
1
-participatingModels
0..1
«data type»
ModelMetadata
-modelId : String{key}
-modelClass : String
-proxelClass : String
-importInterfaces : String [*]
-exportInterfaces : String [*]
+getModelId() : String{query}
+getModelClass() : String{query}
+getProxelClass() : String{query}
+getImportInterfaces() : String [*]{query}
+getExportInterfaces() : String [*]{query}
+getTimeStep() : TimeStep{query}
+isValid() : Boolean{query}
...
-resources
*
-area
1
«data type»
AreaMetadata
-areaId : String{key}
-description : String
-nrProxels : Integer
+getAreaId() : String{query}
+getDescription() : String{query}
+getNrProxels() : Integer{query}
+getProperties() : AreaProperty [*]{query}
+isValid() : Boolean{query}
«data type»
ResourceMetadata
-resourceId : String{key}
-description : String
-resourceType : String
-resourceLocation : String
+getResourceId() : String{query}
+getDescription() : String{query}
+getresourceType() : String{query}
+getResourceLocation() : String{query}
+isValid() : Boolean{query}
-properties
*
-property
1
1
«data type»
AreaProperty
-name : String{key}
-type : String
+getName() : String{query}
+getType() : String{query}
+isValid() : Boolean{query}
-timeStep
1
«data type»
TimeStep
-value : Integer
-unit : TimeStepUnit [1]
+TimeStep(value : Integer, unit : TimeStepUnit)
+getValue() : Integer{query}
+getUnit() : TimeStepUnit{query}
+isValid() : Boolean{query}
«enumeration»
TimeStepUnit
YEAR
MONTH
DAY
HOUR
MINUTE
SECOND

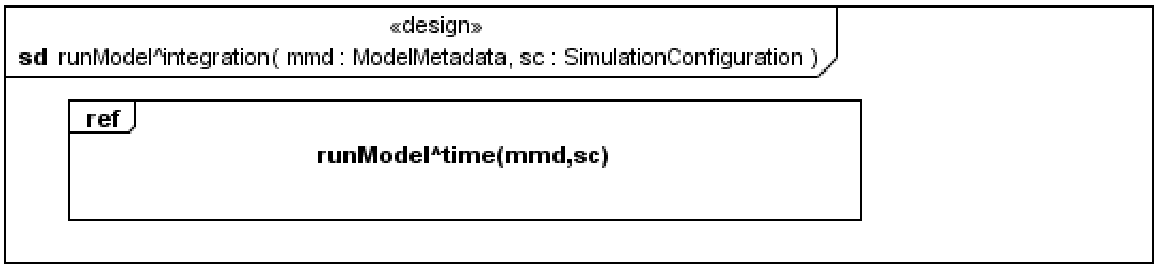

«components»
pkg modeldata^integration
modeldata^integration
«base interface»
DataInterface
DataTable
+getValue(pid : Integer) : DataElement{query}
pid : Integer
-value
0..1
«base class»
DataElement

A.5.2 Behavioural Integration


«design»
sd executeSimulation^integration
: UserInterface
new(simId, partModels, area, resources, begin, end)
«data type»
sc : SimulationConfiguration
{post: self.simulationId=simId and
self.participatingModels=partModels and
self.area=area and
self.resources=resources and
self.begin=begin and
self.end=end}
{pre: sc.isValid()}
new(sc)
sim : SimulationAdmin
{post: self.sc=sc}
start()
ref
runSimulation^integration
finished()


«design»
sd runSimulation^integration
sim : SimulationAdmin
sc : SimulationConfiguration
par
[ ]
new(sc)
linkAdmin : LinkAdmin
{post: self.sc=sc}
new(sc, resourceHandler)
bdAdmin : BasedataAdmin
{post: self.sc=sc and self.resourceHandler=resourceHandler}
ref
initBasedataAdmin^space
new(sc)
tc : Timecontroller
{post: self.sc=sc and
sc.participatingModels->forAll(mmd|
self.nextGet[mmd.modelId]=sc.begin and
self.nextProv[mmd.modelId]=sc.begin)}
getParticipatingModels()
mmds:ModelMetadata[*]
{self.linkAdmin=linkAdmin and
self.bdAdmin=bdAdmin and
self.tc=tc}
loop
[forAll mmd in mmds]
ref
createAndRunModel^integration(mmd,sc)


«design»
sd createAndRunModel^integration(mmd : ModelMetadata, sc : SimulationConfiguration)
sim : SimulationAdmin
linkAdmin : LinkAdmin
bdAdmin : BasedataAdmin
tc : Timecontroller
models[mmd.modelId] : ModelCore
base : AbstractModel
new(mmd, sc, linkAdmin, bdAdmin, tc)
{self.models[mmd.modelId]=models[mmd.modelId]}
{post: self.mmd=mmd and self.sc=sc}
run()
new()
{self.getType().getName()=mmd.modelClass or self=null}
break
[base=null]
ref
handleException
{self.base=base}
setMmd(mmd=mmd)
setSc(sc=sc)
par
[ ]
ref
initModel^data(mmd,sc)
ref
initModel^space(mmd,sc)
ref
runModel^integration(mmd,sc)


«design»
sd runModel^integration(mmd : ModelMetadata, sc : SimulationConfiguration)
ref
runModel^time(mmd,sc)