View-Based Development of a Simulation Framework for Multi-disciplinary Environmental Modelling*

Rolf Hennicker and Matthias Ludwig

Institut für Informatik, Ludwig-Maximilians-Universität München
{hennicker,mludwig}@pst.ifi.lmu.de

Abstract. We report on the development of a large-scale simulation framework for environmental modelling. The framework allows to couple various simulation models from natural and social science disciplines to perform integrative simulations. It has been constructed following a development methodology based on the identification of different functional views, which are concerned with data exchange, simulation space and coordination of distributed simulation models with respect to (logical) simulation time. On all levels of the development we have rigorously applied modelling and specification techniques including the last step, in which the different views are integrated into a component model of the full framework. The requirements for the correct coordination of simulation models have been formally specified in terms of the process algebra FSP and the design model has been model checked against the coordination requirements. Within the GLOWA-Danube project the framework has been successfully instantiated to construct the distributed simulation system Danubia which integrates up to 15 simulation models from various disciplines to model the consequences of global climate change for the water household on regional scales.

1 Introduction

Global climate change has an increasing impact on our natural and social environment. Therefore it is important to understand better the complex, mutually dependent processes occurring in nature and in socio-economic systems which calls for interdisciplinary research. Computer-based simulations have emerged as an appropriate means for studying possible scenarios for the future and to support the management of adaptation and/or prevention strategies. While in the past simulation models often were developed as monolithic applications by a particular discipline to provide specialised answers, nowadays the need for interdisciplinary modellling and integrative simulation has been recognized.

* This research has been partially supported by the GLOWA-Danube project 01LW0602A2 sponsored by the German Federal Ministry of Education and Research and by the EU project ASCENS, 257414.

R. Calinescu and D. Garlan (Eds.): Monterey Workshop 2012, LNCS 7539, pp. 224–250, 2012.
© Springer-Verlag Berlin Heidelberg 2012

In an integrative simulation system several simulation models are coupled in order to analyse dependencies and transdisciplinary effects of the simulated processes. Following [20] and [8] 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) and treatment of simulation time (discrete or continuous)[1]. Moreover, in a network of coupled simulation models one can distinguish whether the models are sequentially executed one after the other (possibly with iterations), whether they are concurrently executed and whether they are dependent from each other.

Coupling of simulation models from various disciplines is a non-trivial task, both conceptually and also from the implementation point of view. One has to cope, among others, with different simulation paradigms, different resolutions of space, and different local time scales to represent simulation time. For instance, in natural sciences often a process-based simulation approach is preferred, models typically use grid-based resolutions of space, and the time scale typically ranges from minutes to hours. In social sciences, however, an agent-based approach is most likely, space is often distributed into political units, and the time scale is usually more coarse ranging from months to years.

In this paper, we focus on process-based models which simulate spatially distributed processes and work on discrete time scales. We consider concurrently running simulation models which are dependent and exchange data at runtime. In this context, we report on the development of a generic framework for computer-based environmental modelling which has been constructed within the project GLOWA-Danube, cf. [22,26], which is part of a program on the consequences of climate change set up by the German Ministry of Education and Research. The framework is generic in the sense that it is, in principle, applicable to any kind of model which supports distributed geographical units of arbitrary size and arbitrary discrete time scales.

The development of the simulation framework has been guided by conceptual and architectural requirements. Conceptually, we have identified three major issues. The framework should support:

1. Data exchange between concurrently running simulation models.
2. Consistent treatment of simulation space for all models.
3. Coordination of simulation models with respect to simulation time.

From the architectural perspective, two logical layers are required, a *framework core* and a *developer interface* as indicated in Fig. 1. The framework core comprises all features that can be handled by the framework itself like, e.g., building simulation configurations and coordination of simulation models. Hence, it serves as a runtime environment for coupled simulations. The developer interface is intended to facilitate the implementation of single simulation models. It provides a programming interface, where particular elements exhibit so-called *plug-points*

[1] Our notion of simulation time does not refer to real time but to the specific date for which a simulation model actually computes data; e.g. the simulated temperature at 5 p.m. on July 5th, 2035.

(in the sense of [7]), which have to be filled with appropriate *plug-ins* in order to obtain an executable system. The plug-ins are provided by concrete simulation models, say M1, ..., M4, as indicated in Fig. 1. Hence, the simulation models instantiate the generic framework to a complete, coupled simulation system. The framework core is transparent for the model developer. Thus the developer of a simulation model is not concerned with administrative issues, like, e.g. model linking. On the other hand, all simulation models must adjust to general rules for common structure and behaviour which are implemented in the framework.

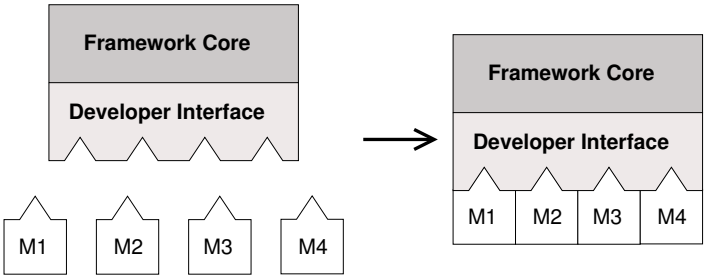

Fig. 1. Framework layers

For the development of the framework we have applied a rigorous methodology based on different functional views (or aspects) and on different abstraction levels. The view-based approach supports separation of concerns which is mandatory to understand the various tasks, in which an integrative simulation framework is involved. In our context, we have identified three views related to the three requirements from above: *data exchange*, *simulation space* and *simulation time*. These views are founded on a common *base view* which deals with basic properties of integrative simulations. We propose three abstraction levels for each view, dealing with *requirements*, with *design* and with the construction of a *component architecture*. Finally, on the component level, the single system views are integrated into an overall component model of the simulation framework. Fig. 2 gives an overview of our methodology which shows that the base view is extended on each abstraction level. As indicated in the picture, the diagrams must commute, i.e. extensions (denoted by $\hookrightarrow$) and refinements (denoted by $\rightsquigarrow$) must be compatible with each other.

For the representation of each view we use the Unified Modeling Language UML [15] as a graphical notation and the Object Constraint Language OCL [27] for specifying constraints. We have restricted the use of UML to an excerpt for which we have defined refinement relations between models on different abstraction levels, extension relations between models on the same abstraction level as well as a construction for model integration. We use structural models in the form of class and component diagrams and behavioural models in the form of sequence diagrams. For the most critical part of the framework, concerning the

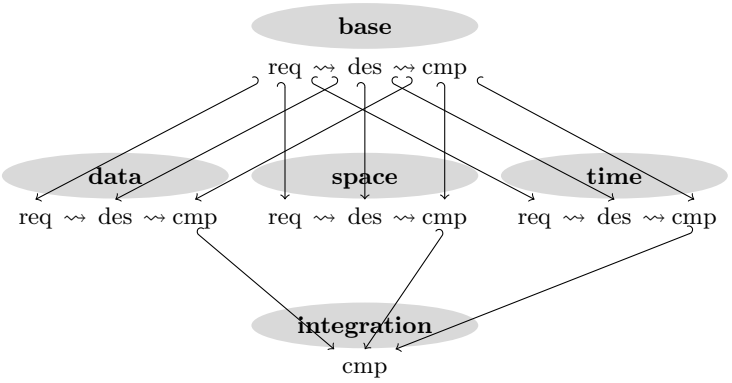

Fig. 2. View-based development

coordination of the simulation models w.r.t. simulation time, formal specifications in terms of the process algebra FSP [23] have been provided and the design model (using a timecontroller) has been model checked against the coordination requirements.

The framework implementation is systematically derived from the integrated component model by a pattern transforming components into Java packages which contain component managers to instantiate interfaces between components. The framework has been implemented as a distributed system relying on Java's Remote Method Invocation interface. Within the GLOWA-Danube project, the framework has been successfully applied to construct the distributed simulation system DANUBIA which integrates up to 15 simulation models from various disciplines, like meteorology, hydrology, plant physiology, glaciology, economy, agriculture, tourism, and environmental psychology. Actually, DANUBIA is already in use as a tool for decision makers to support the sustainable planning of the future of water resources in the Upper Danube basin.

A number of other frameworks and interfaces supporting integrated environmental modelling emerged since the GLOWA-Danube project started in 2001; for an overview see [16]. There are, e.g., the Object Modelling System OMS [18], ModCom [13], The Invisible Modelling Environment TIME [24], and the Open Modelling Interface OpenMI [9]. 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 data exchange. In contrast to our approach, OpenMI allows only for sequential execution of dependent models. OMS supports also parallel execution, as long as models are independent from each other. ModCom and TIME are both not designed for distributed execution. Distributed simulations of dependent models are supported by the High Level Architecture HLA [6] which was set up in the nineties to define a structural basis for simulation interoperability. A formal model for the architecture of HLA

has been provided in [2] on the basis of the architectural description language Wright [1]. HLA provides a general purpose architecture while our approach is tailored to environmental simulations fixing particular rules for this kind of application. For instance, the life cycle and the coordination of simulation models are already implemented in the framework core. As a consequence, the developer of a simulation model has only to implement the plug points provided by our developer interface while in HLA a simulation model (called federate there) must take care of calling the services of the HLA runtime infrastructure, e.g. to publish state updates or to request advance of logical time, in accordance with the type of simulation. Thus even real-time players can be integrated in HLA architectures which was not the intention of our environmental modelling approach.

After this introduction we proceed by illustrating in more detail the application of our development methodology for (parts of) the base and the time view of the simulation framework in Sects. 2 and 3. In Sect. 4 we describe briefly the requirements models for the data exchange and the simulation space view. We do not consider in detail the component models of the single views, but we give an overview of the final result of their integration in Sect. 5. Then we discuss the application of our framework to obtain the DANUBIA simulation system in Sect. 6 and we finish with some concluding remarks in Sect. 7.

2 Base View Development

2.1 Base View Requirements

Requirements analysis concerns the identification and modelling of concepts which are crucial for the envisaged system. To model the concept of an *integrative simulation* we use the class Simulation shown in Fig. 3. Any (integrative) simulation has a (non-empty) set of participating simulation models represented by instances of the class Model. We require that simulations and models can be identified by a unique simulationId and modelId, resp., expressed by the property {key}, which is a shorthand notation for a corresponding OCL invariant defined in an obvious way.

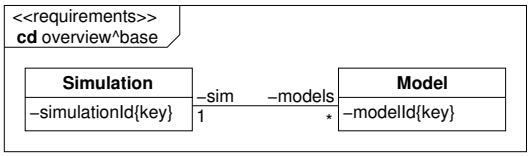

Fig. 3. Base view requirements: static model

Concerning basic dynamic behaviour of integrative simulations, the sequence diagram in Fig. 4 shows a minimal set of actions that are expected, when an integrative simulation is performed. By means of an appropriate user interface,

which is not in the scope of the simulation framework and therefore is modelled as an actor, a Simulation object is created and started. Then instances of all participating models must be created and executed (as indicated in the loop fragment). When a model has finished its simulation run the Simulation object is notified by the message finished. If this notification has arrived from all participating models the end of the simulation is signalled to the UserInterface. All messages in the sequence diagram are asynchronous (indicated by an open arrowhead). Hence the single simulation models are executed in parallel after they have been started within the loop. Obviously, the static and the dynamic model are consistent, since all lifelines in the sequence diagram correspond to roles and types of the static model.

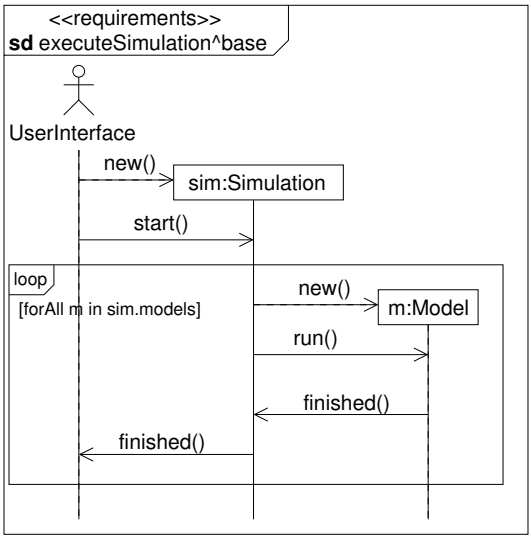

Fig. 4. Base view requirements: dynamic model

2.2 Base View Design

Design modelling concerns the development of solutions in order to realise the abstract concepts. In our case we discriminate active entities for controlling and descriptive objects that carry information. An overview of the structural design model of the base view is depicted in Fig. 5.

The class Simulation of the requirements model is split into the two classes SimulationAdmin and 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 (indicated by a vertical double line on the border of the UML class box), the class SimulationConfiguration holds descriptive information about the simulation (indicated by the stereotype «data type»).

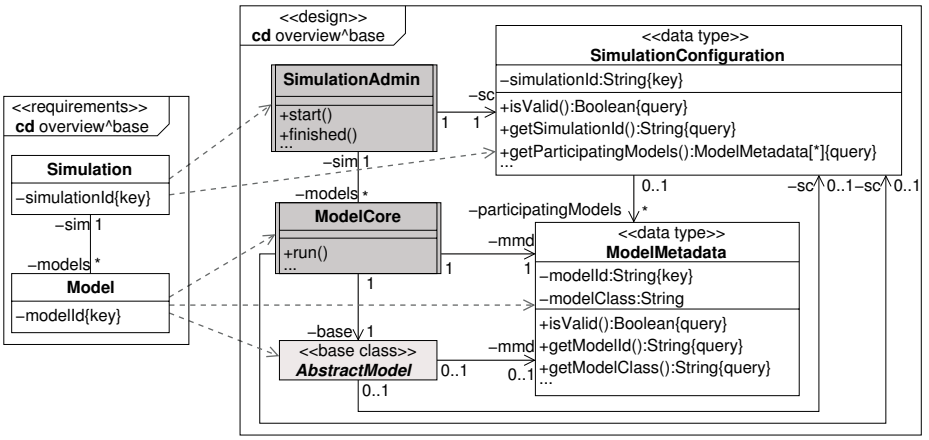

Fig. 5. Base view design: static model

The requirement class Model has been split into the three design classes ModelCore, AbstractModel and ModelMetadata. While ModelMetadata is a class for storing meta data of a simulation model, the classes ModelCore and AbstractModel represent a simulation model itself. This partition follows the framework principle explained in Sect. 1: while ModelCore belongs to the framework core (indicated by the dark colour) to implement the general life cycle of a simulation model within the method run (and therefore is again 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) extension. In contrast to the requirements model, the static design model shows operations which are either derived from the messages of the dynamic requirements model (see Fig. 4) or identified during the design phase, like isValid. The latter operation occurs in the classes SimulationConfiguration and ModelMetadata, to determine the validity of simulation configurations and the validity of model meta data resp. In each case it is a query specified by OCL pre- and postconditions as expected.

```
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 <> ""
```

There is also a refinement of the dynamic requirements model of Fig. 4 taking into account the new classes, see Fig. 6. The sequence diagram depicts preconditions that must be satisfied before an object is created and postconditions that must be valid after creation. It also contains a reference to a nested sequence diagram not shown here.

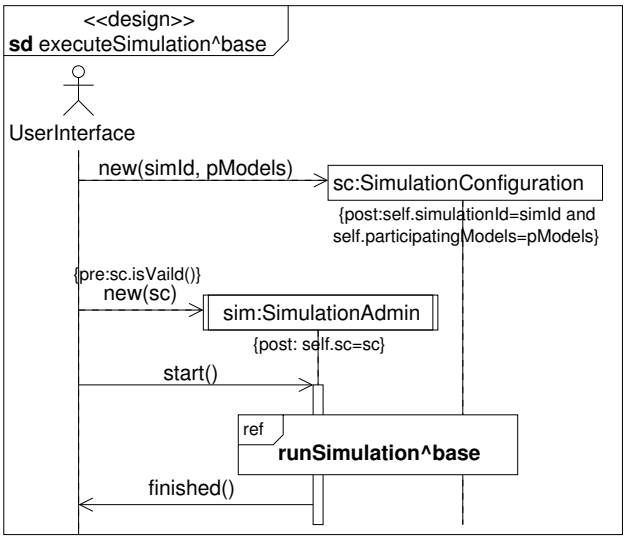

Fig. 6. Base view design: dynamic model

Refinement Rules. We have defined general rules for refinement which allow us to split requirement classes into sets of design classes and to rename and add model elements and behaviours. The rules are defined for class diagrams and sequence diagrams by taking into account their syntactic structure. Semantic refinement relations between sequence diagrams can be found, e.g., in [5], or in the STAIRS approach [11,10].

Structural Model. 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 refining 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 resp. such that multiplicities are respected, and
- for each invariant Inv occurring in SM_1 there exists an invariant Inv' in SM_2 such that $Inv' \Rightarrow Inv$.

Dynamic Model. Let SD_1 and SD_2 be sequence diagrams with corresponding structural models SM_1 and SM_2 resp. such that $SM_1 \rightsquigarrow SM_2$. Then SD_2 is a refinement of SD_1, denoted by $SD_1 \rightsquigarrow SD_2$, if

- 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 there is a corresponding interaction fragment in SD_2 such that the (partial) order of the interaction fragments in SD_1 is reflected by the (partial) order of the corresponding interaction fragments in SD_2.

2.3 Base View Components

The goal of our component model is to group the classes, identified in the structural design model, into components following the general principles of high cohesion and low coupling. Components themselves are connected via (provided and required) interfaces and they can be organised hierarchically. We say that a component model is a refinement of a design model if each class of the design model occurs in one of the components and if each association of the design model is either preserved, if the associated classes belong to the same component, or otherwise, it is resolved by connections via interfaces. We use components solely for structuring purposes; they are not instantiable and hence behaviours are implemented by the classes inside a component. (In particular, this allows a straightforward implementation in object-oriented languages.) Hence the transition from design to components concerns only the static aspects while the dynamic model of the design remains still valid on the component level.

The UML component diagram in Figure 7 shows the component model for the base view. We use two components, Simulation and Model, containing the respective classes for simulations and for models occuring in the static design model in Fig. 5. The associations between simulation and model classes and the interactions between their instances have lead to the interfaces ModelAccess and ExceptionHandler which are implemented (depicted by the ball notation) and used (depicted by the socket notation) by the appropriate classes of the components. (The multiplicity * indicates that at runtime arbitrary many instances of model classes can interact with a simulation.) The interfaces SimulationAccess and UserInterface show the open connections to the user interface not being part of the framework.

3 Simulation Time and Coordination

A central role in integrative environmental simulations is played by the notion of time and by the coordination of the simulation models. As already mentioned in the introduction, our notion of time expresses logical simulation time and does not refer to execution time. In this section we show how the models of

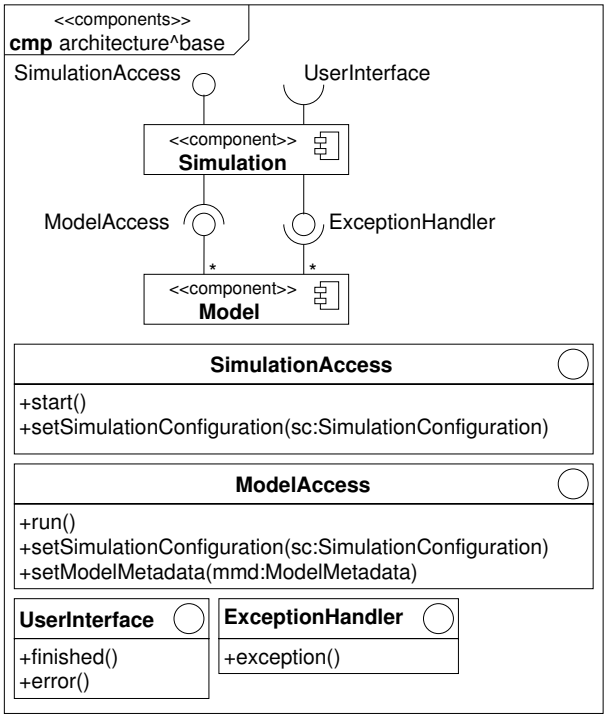

Fig. 7. Base view: component model

the time view are constructed by extensions of the corresponding levels of the base view. Fig. 8 gives an overview of the single extensions and refinements to be considered. The single steps are performed in the following order: Steps 1 and 2 concern the refinement from requirements to design and from design to components in the base case which have already been carried out in Sect. 2. In step 3, the requirements model of the time view is constructed as an extension of the requirements model of the base view. This requirements model is then refined, in step 4, into a design model of the time view. This leads to the proof obligation (*) that the resulting design model of the time view is an extension of the design model of the base view, i.e. the lefthand diagram commutes. Finally, in step 5, the design model of the time view is refined into a component model. This leads to the proof obligation (**) that the resulting component model of the time view is an extension of the component model of the base view, i.e. the righthand diagram commutes.

In our approach extension relations are defined by precise (syntactic) rules on the basis of an excerpt of the UML metamodel for class and sequence diagrams. In principle, an extension relation is a particular case of a refinement relation such that renaming of model elements and splitting of classes and lifelines is not allowed. For details see [21].

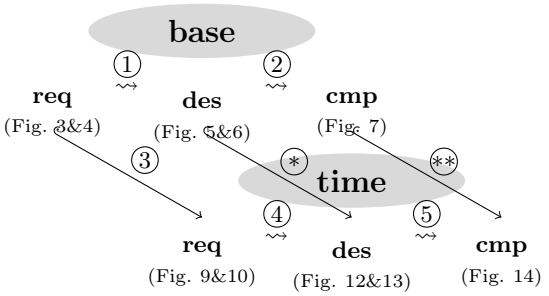

Fig. 8. Base and time view development

3.1 Time View Requirements

A simulation model simulates a (physical or social) process for a certain period of time. In an integrative simulation all models must agree on a common time period which is determined by the overall simulation[2]. Hence, the class Simulation of the base view requirements in Fig. 3 is extended by two attributes storing the *begin* and the *end* of a simulation. On the other hand, since we are considering discrete time, each model has an individual *time step*, which is represented by an additional attribute of the class Model shown in Fig. 9.

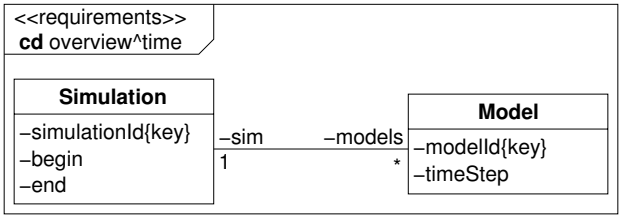

Fig. 9. Time view requirements: static model

Much more involved are the *behavioural* requirements concerning simulation time and coordination. In contrast to stand-alone simulation models, a coupled simulation model not only computes data, but has to perform activities concerning data exchange in accordance with the simulation time. The general life cycle a coupled simulation model must follow is described as follows.

- *initialise* model with basic data (e.g. about the simulation area)
- *provide* exported data at the model's export interfaces[3]
- while not at simulation end
  - *get data* from the model's import interfaces

[2] For instance, in the GLOWA-Danube project the common simulation time spans typically 50 years starting from the actual date.

[3] Which is necessary for other models to start their computation.

- *compute* new data for the next time step
- *provide* newly computed data at the model's export interfaces

– *finalise* the simulation (e.g. closing of open files or database connections)

This general life cycle of any simulation model is integrated into the sequence diagram of the base view (Fig. 4) resulting in the sequence diagram for the dynamic time view requirements shown in Fig. 10, which is an obvious behavioural extension of the former one.

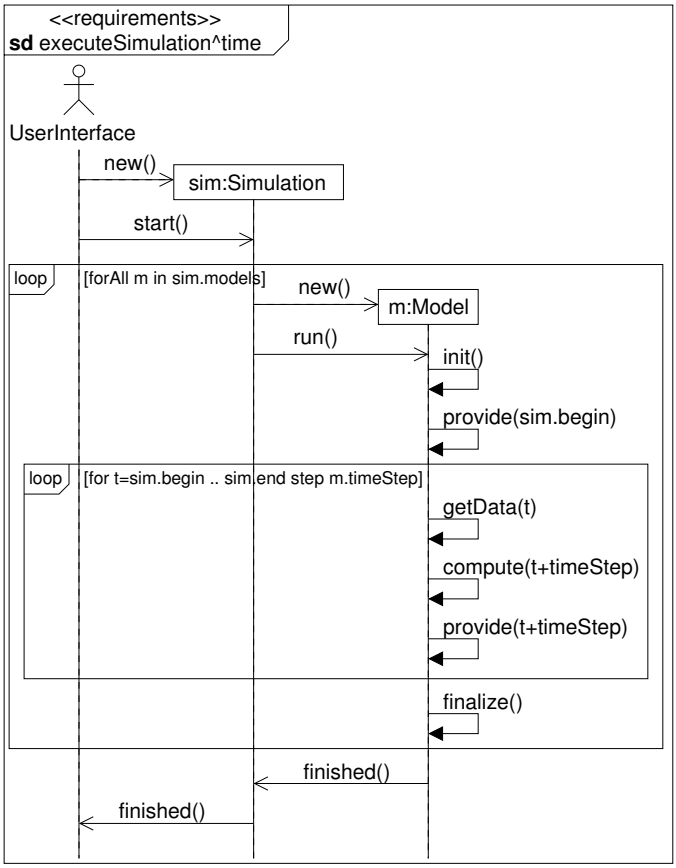

Fig. 10. Time view requirements: dynamic model

The sequence diagram in Fig. 10 models the parallel execution of all simulation models participating in an integrative simulation. But it allows much more (parallel) executions than desired since the single models are by no means coordinated w.r.t. simulation time yet. For instance, Fig. 10 would allow an execution where the first simulation model has already finished its getData - compute - provide loop, while some other model, whose exported data is needed by the

first one, has not even provided data yet or has only provided data which is obsolete for the first model. Hence, we are faced with a non-trivial coordination problem which cannot be specified in UML. Our solution is to switch from the UML requirements model to a formal specification of the coordination problem. For this purpose we use the process algebra FSP (Finite State Processes FSP) introduced by Magee and Kramer [23] which allows us to formalise the coordination requirements in terms of so-called property processes[4]. Then we develop an FSP design model and check that the design model satisfies the coordination constraints. Finally, we move back to UML and obtain a UML design model which is a refinement of the original UML requirements model for the time view. Our procedure is depicted in Fig. 11. We start with the specification of the coordination problem.

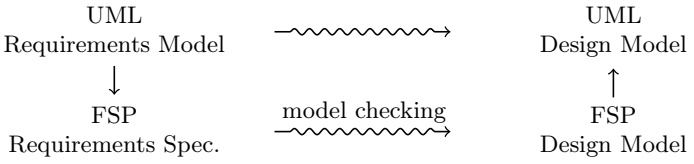

Fig. 11. From UML to FSP and back

The Coordination Problem. 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.

To specify the coordination conditions, we first formalise the general life cycle of a simulation model in terms of the following FSP process `MODEL`, which is

[4] An alternative formalisation of the coordination problem using purely mathematical notations is given in [3].

parameterised with respect to the model's time step. The actual simulation time, when a certain action happens, is modelled by an action index. 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 inner loop of the sequence diagram in Fig. 10. Let us remark 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 resolved by using smaller time steps.

```
1  range SimTime = SimStart..SimEnd
2  MODEL(Step) = (run -> init -> 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 (finalize -> finished -> STOP).
```

A particular simulation model with modelId `m` and time step `sm` is then formalised by the labelled FSP process `[m]:MODEL(sm)`. In this process all actions are prefixed by the model identificator `m`, i.e. the actions are of the form `[m].run`, `[m].init`, `[m].provide[t]`, `m.get[t]` etc.

On this basis we can formalise the coordination conditions in terms of the following FSP property process `VALIDDATA`. The first alternative of the process `VALIDDATA` formalises condition (U) from above such that the index variable `nextUser` corresponds to t_U, `nextProv` corresponds to t_P and therefore `nextProv-StepProv` corresponds to $last_P$. The second alternative formalises condition (P) from above.

```
1  property VALIDDATA(User, StepUser, Prov, StepProv) =
2    VD[SimStart][SimStart],
3  VD[nextGet:Time][nextProv:Time] =
4   (when (nextGet<nextProv)
5       [User].getData[nextGet] -> VD[nextGet+StepUser][nextProv]
6   |when (nextGet>=nextProv)
7       [Prov].provide[nextProv] -> VD[nextGet][nextProv+StepProv]).
```

For a system of coupled simulation models all requirements concerning the validity of data are now obtained by pairwise instantiations of the generic property process `VALIDDATA` such that, in different instantiations, the same simulation model occurs once in the role of a user and once in the role of a provider of data. To validate the property processes we have used the FSP-tool LTSA (Labelled Transition System Analyser) which translates FSP processes into labelled transition systems and visualises the transition systems if the property process is instantiated (by small parameters).

3.2 Time View Design

The formal specification of the coordination requirements is highly non-constructive. The basic idea of the formal design model is to introduce a global control process that coordinates appropriately all simulation models participating in an integrative simulation. In [12] we have constructed an explicit coordination process with FSP, called `TIMECONTROLLER`, which has actions of the form `m.enterGet[t], m.exitGet[t], m.enterProv[t], m.exitProv[t]` for all model identificators `m` and time steps `t` within the range of the simulation time. The `enter` actions are guarded by appropriate coordination conditions like, in the case of three simulation models to be coordinated,

```
  when (t<nextProv1 & t<nextProv2 & t<nextProv3)
       [1..3].enterGet[t] -> ...
| when (nextGet1>=t & nextGet2>=t & nextGet3>=t)
       [1..3].enterProv[t] -> ...
```

The `exit` actions are not guarded but change the value of the `nextGet` and `nextProv` variables accordingly.

Moreover, the FSP process `MODEL` of the requirements model is extended such that any `provide` and `get` action is surrounded by appropriate `enter` and `exit` actions which are shared with the timecontroller. Since shared actions can only be executed together, the timecontroller process now monitors when a simulation model can execute its `get` and `provide` actions in the parallel composition

```
([1]:MODEL(s1)||...||[n]:MODEL(sn)||TIMECONTROLLER)
```

We have verified with LTSA that the FSP design model indeed satisfies the coordination conditions formalised by (instantiations of) the property processes of the FSP requirements model; see [12] for more details.

The formal FSP design model suggests a particular architecture of a design model on the UML level which introduces the class Timecontroller shown in the static UML design model in Fig. 12. Obviously, this model is a refinement of the static time view requirements model in Fig. 9 and also an extension of the static base view design model in Fig. 5, as required by the proof obligation (*) in Fig. 8.

During an integrative simulation run there is exactly one instance of the class Timecontroller which acts as a monitor that must be called by the simulation models (more precisely, by the ModelCore instances) before data delivery and data access can be performed. This is pointed out in Fig. 13 which shows an excerpt of the dynamic UML design model for the time view. In particular one can see in Fig. 13 that any enter message called on the timecontroller is equipped with an "enable" constraint which expresses a coordination condition derived from the FSP guards in the timecontroller process. We have introduced enable constraints, though not part of the OCL standard, to model situations in which

a calling object will be blocked if the condition is not valid and then waits until the constraint becomes true. Let us remark that enable conditions are methodologically (and also from the implementation point of view) quite different from OCL preconditions, since preconditions are expected to hold when an operation is called. Indeed, when we use preconditions in our models, we express a requirement for the caller and our reference implementation will raise an exception if the precondition is not satisfied upon operation call. In contrast, if an operation call is constrained by an enable condition, say `cond`, then the operation, say `op`, will be implemented in Java by a synchronized method applying the following general pattern proposed in [23]:

```
public synchronized void op() throws InterruptedException {
    while (!cond) wait();
    ... // monitor state = nextState
    notifyAll();
 }
```

If the condition `cond` is not satisfied, the calling thread will be blocked by `wait`. If the condition is satisfied the thread may enter the critical region and change the monitor state. After that it releases, if necessary, all waiting threads by `notifyAll`. The `while` loop ensures that the condition is checked again after a thread has been released which is necessary since Java follows the "signal and continue" principle.

The sequence diagram in Fig. 13 shows also that, after a simulation model has entered the monitor, the concrete execution of getting data and providing data is delegated to an instance of the class AbstractModel and similarly for computing new data. How the operations getData, provide and compute will finally be implemented is due to the developer of a concrete simulation model who has to extend the abstract model class. Therefore getData, provide and compute are declared as plug points in the class AbstractModel as indicated in Fig. 12.

As already mentioned, Fig. 13 shows only an excerpt of the dynamic design model for the time view. The full model is a hierarchically organised sequence diagram presented in all details in [21]. It is a refinement of the dynamic time view requirements model in Fig. 10 and also an extension of the dynamic base view design model in Fig. 6 (as required by the proof obligation (*) in Fig. 8).

3.3 Time View Components

The component model for the time view encapsulates the Timecontroller class in the component TimeCoordination, which is connected to the two components of the base view by appropriate interfaces, one to access the timecontroller monitor from a model and the other one to pass a simulation configuration from the simulation administrator. This corresponds to the refinement step 5 in Fig. 8.

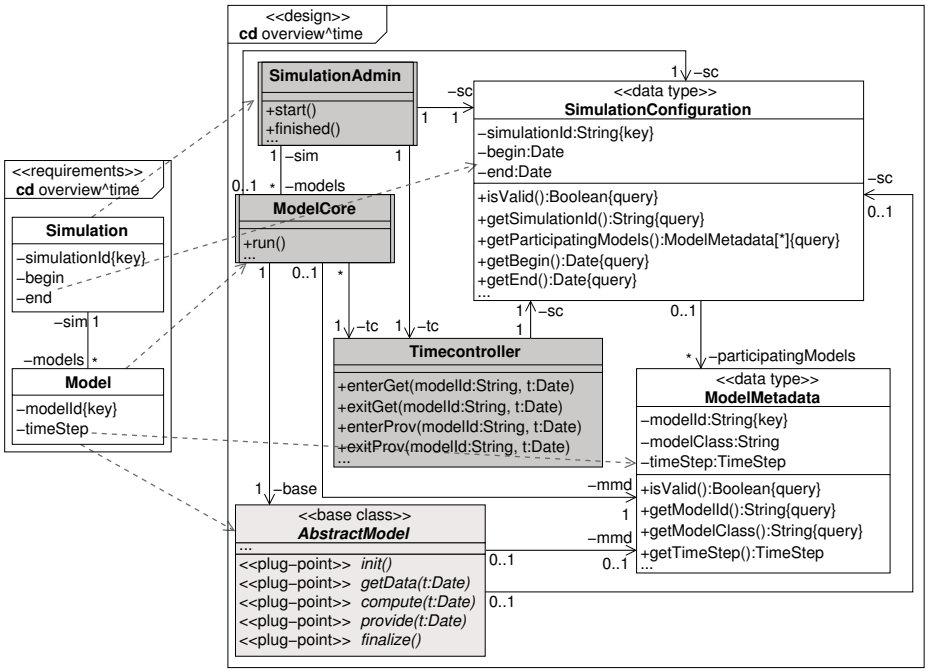

Fig. 12. Time view design: static model

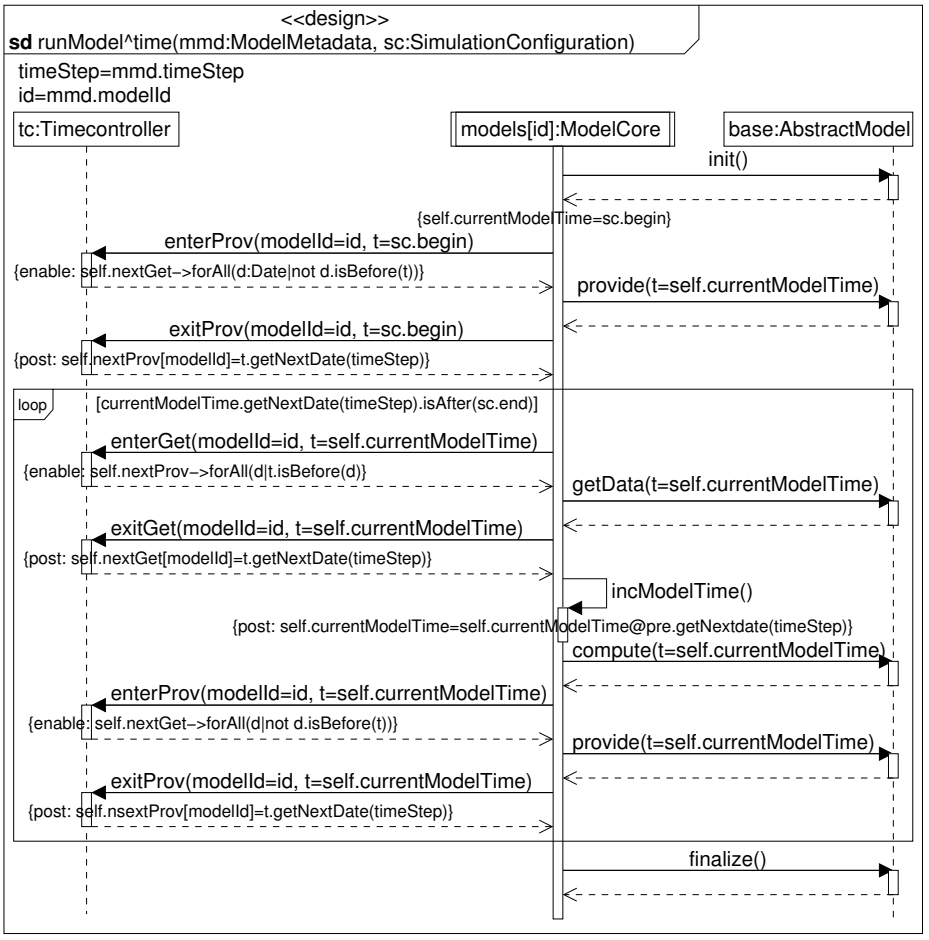

Fig. 13. Time view design: Excerpt of the dynamic model

Fig. 14 shows the component model of the time view. It extends the base view component model in Fig. 7 by the component TimeCoordination and by the two interfaces TimecontrollerMonitor and TimeCoordinationAccess together with their associated relationships for usage and implementation. Hence, the proof obligation (**) of Fig. 8 is satisfied.

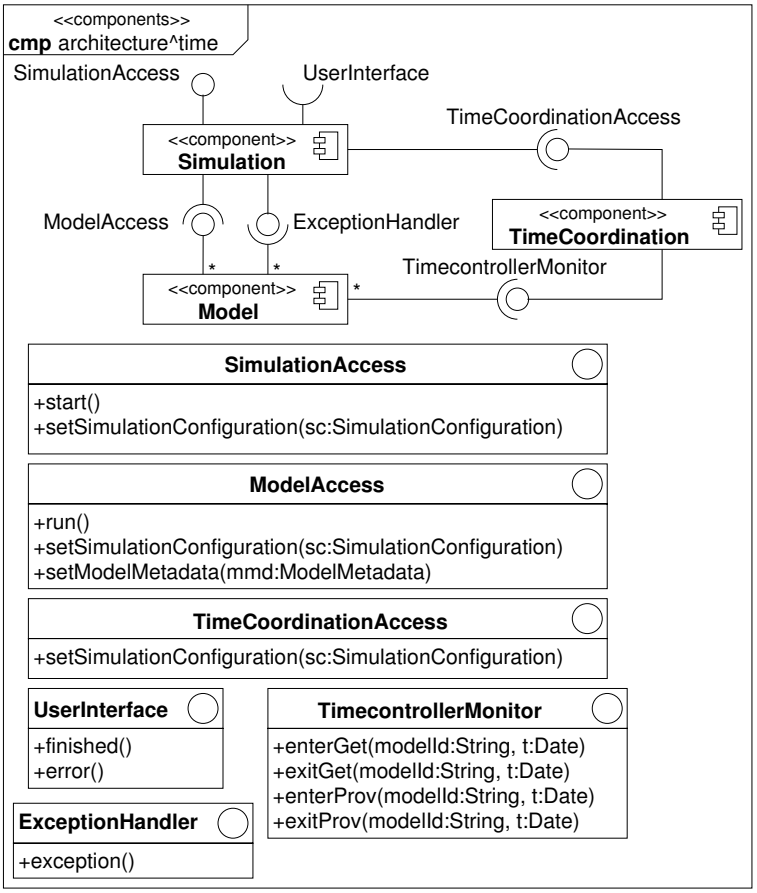

Fig. 14. Time view: component model

4 Data Exchange and Simulation Space

This section gives a short overview on the remaining system views concerning data exchange and simulation space. We only present the static requirements models as extensions of the base view to get an idea of the relevant concepts in these cases. For the complete development of the data exchange and simulation space aspects we refer to [21].

4.1 Data Exchange: Requirements

In a coupled simulation, the single simulation models exchange data at runtime. We require that for data exchange they use data interfaces. For each simulation model the interfaces appear in two different roles. First, a model must have a set of *export* interfaces to provide computed data for other models. Secondly, a model imports data that it needs for its own computations from other models. For this purpose it uses *import* interfaces (which at the same time are export interfaces of a providing model). Statically, we extend the requirements model of the base view (cf. Fig. 3) by the type DataInterface associated with the conceptual class Model by two directed associations, one for the exported and one for the imported interfaces of a simulation model, as shown in Fig. 15. A concrete example of an exported and imported interface of a groundwater simulation model is given later when we illustrate the framework instantiation in Fig. 19.

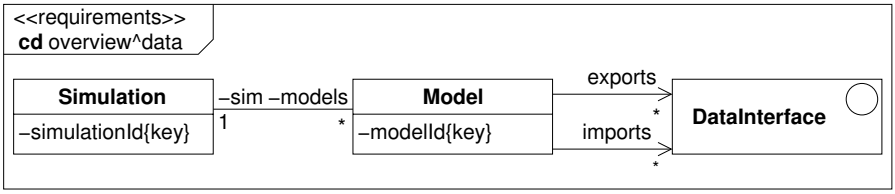

Fig. 15. Data exchange requirements: static model

The class diagram in Fig. 15 is enhanced by a consistency condition for integrative simulations which requires that for any model participating in a simulation and for each interface imported by the model there must exist exactly one simulation model which exports that interface. The following OCL invariant expresses this requirement.

```
context Simulation inv:
    self.models.forAll(m |
      m.imports->forAll(i |
        self.models->one(n |
          n.exports->includes(i))))
```

The dynamic requirements model for data exchange is a simple extension of the basic one (Fig. 4), which integrates an activity to link models via their corresponding import/export data interfaces.

4.2 Simulation Space: Requirements

Any environmental simulation model operates on some simulation space. For integrative simulations we assume that all models use the same simulation space which consists of a set of so-called proxels. The term proxel (cf. [25]) 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 Fig. 16. 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 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. These proxels must belong to the simulation area of the simulation, in which a model participates. This requirement is again expresses by an OCL invariant not shown here. Obviously, the static requirements model in Fig. 16 is an extension of the basic one in Fig. 3.

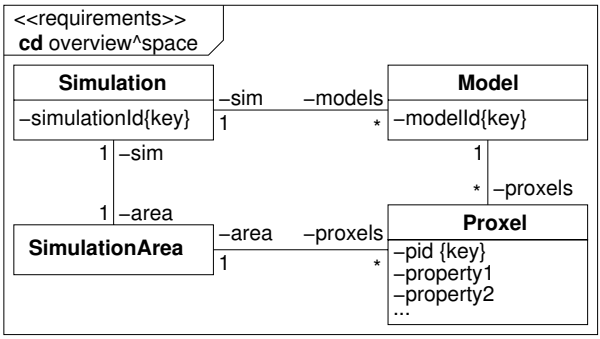

Fig. 16. Simulation space requirements: static model

5 Integration

In the last step of our development methodology the component models of the single views are integrated into an overall component model which is an extension of each view, as indicated in Fig. 17. Though we have not considered the component levels of the data exchange and space views, we still want to give an overview of the component architecture of the full simulation framework shown in Fig. 18. One can see that it extends the time view component model of Fig. 14 by the component ModelLinking, which stems from the data exchange component model, and by the two components Basedata and Proxel, both stemming from the simulation space view. The latter has been introduced as a subcomponent of the Model component. As indicated in the picture, all components are connected via appropriate provided and required interfaces.

Our integration follows a general integration procedure for static and dynamic models which produces a unique result up to renaming, similarly to a push-out construction, and which is independent of the order of the integration (up

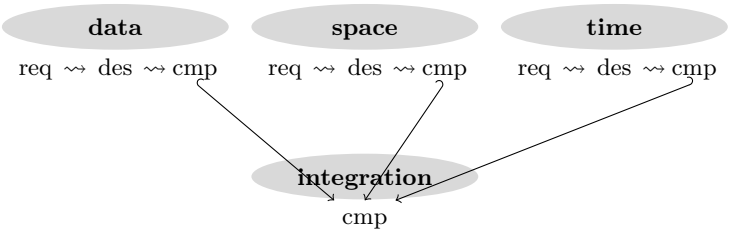

Fig. 17. Integration

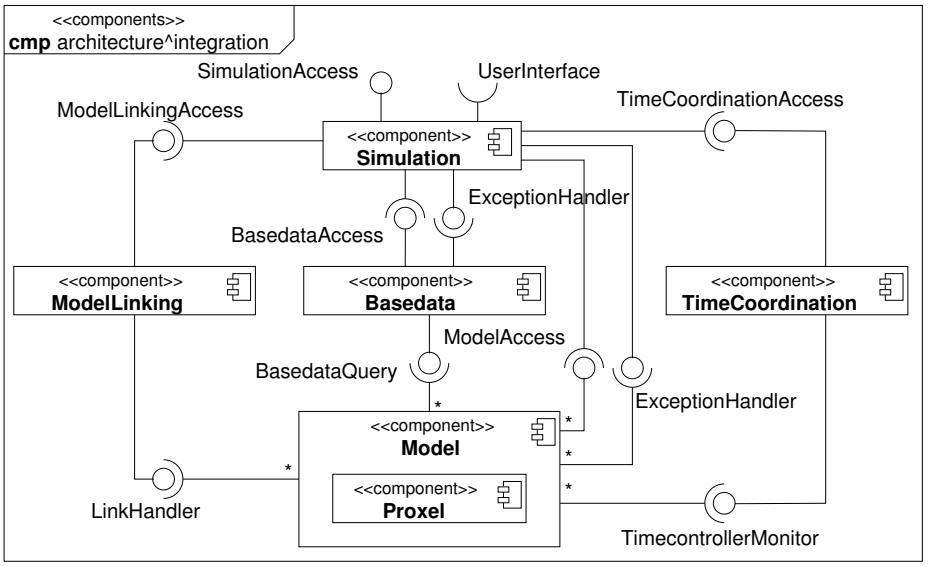

Fig. 18. Integrated component model: Overview

to renaming) in the case of more than two diagrams. For static models, our procedure works on an excerpt of the UML metamodel for class and component diagrams; it is widely adopted from the UML package merge construction as explained, e.g., in [14].

But we have also to consider the integration of dynamic models in the form of sequence diagrams, which we have used in the dynamic design models of each view and which are still valid in the single component models. The task is to describe how two sequence diagrams extending a common base sequence diagram are integrated. For this purpose we have defined general rules which work on an excerpt of the UML metamodel for sequence diagrams. An integrated sequence diagram comprises the lifelines, messages and interaction fragments of its constituent parts. During the integration process the interaction fragments of the base diagram act as synchronisation points whereas the other interaction

fragments of the extended diagrams can be arbitrarily interleaved. Hence, in the integrated sequence diagram they are arranged, between the synchronisation points, in separate operands of the UML par construct to express parallel executions. Our construction ensures that the partial order of interactions of each single sequence diagram is preserved by the integration. In general, it may however happen, that the resulting set of interaction fragments is not partially ordered, i.e. the result is not necessarily a well-formed sequence diagram; cf. [17]. Thus our integration construction for dynamic models defines in fact a partial function. Concerning our simulation framework the integration of the sequence diagrams of the single views is rather involved and presented in detail in [21].

An integration process similar to ours, but without using a common base defining the synchronisation points, is presented in [4]. The approach is based on a categorical construction using labelled prime event structures [28]: the synchronisation points of two sequence diagrams are calculated as a pull-back, and the integration as a push-out.

Finally, let us still emphasise that all models of the simulation framework are programming language independent. The integrated component model is, however, sufficiently detailed such that it can be directly transformed into a concrete implementation. We have constructed a reference implementation of the framework in Java following a client/server architecture such that network communication is performed by means of Java's Remote Method Invocation interface RMI. Since Java does not support the concept of components we have developed a transformation pattern such that UML components are mapped to Java packages, each package containing a (public) manager class that is responsible for generating objects that implement the provided interfaces of the components in accordance with the component model.

6 Application of the Framework

Within the GLOWA-Danube project [22,26] our simulation framework has been instantiated 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. The ultimate 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 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.

How a concrete simulation model is integrated into the framework is shown in Fig. 19 in terms of a groundwater model. The upper layer indicates (part of) the framework core and the middle layer (part of) the developer interface as

discussed in Sect. 1. 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, gwLevel` etc., and by providing implementations for the plug-in operations like, e.g., `compute` and `computeProxel`. Thereby the framework's core functionality concerning runtime coordination, management tasks and the like is completely hidden from model developers.

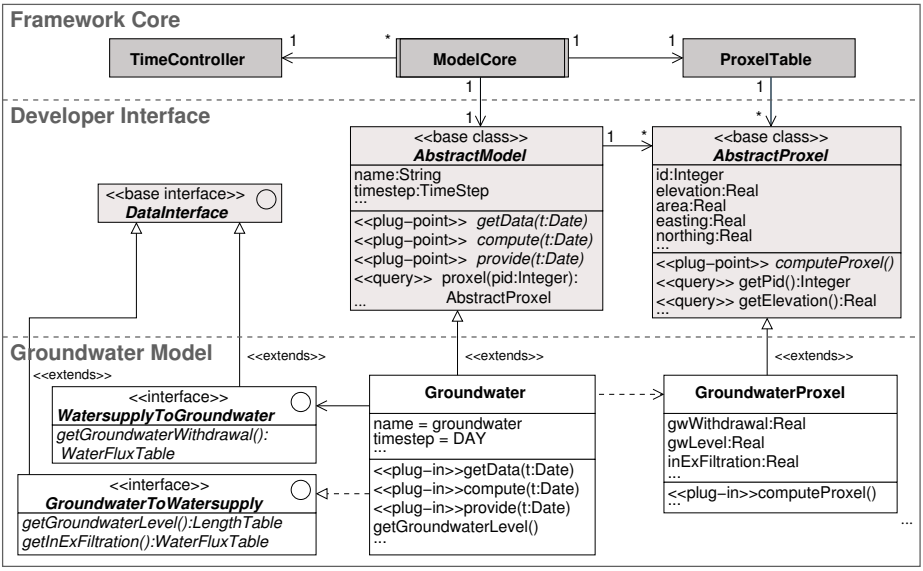

Fig. 19. Instantiation of the framework

While the framework is primarily intended for the development of new simulation models, legacy models can yet be integrated into the framework as long as their computation steps are controllable from the outside. In this case the legacy model is surrounded by a wrapper which must implement the (plug-in) operations like any other model. The concrete computation steps of the legacy model can then be initiated by using the Java Native Interface.

Of course, the performace of a coupled simulation run depends strongly on the number and type of the participating simulation models. For instance, a simulation that couples only socio-economic models (together with a groundwater model needed to interact with the water supplier model) runs actually between three and four days for a simulation period of 50 years. In this case the smallest local time step is one day. If, however, all 15 models participate in a simulation run, then for the same simulation period of 50 years the simulation execution takes approximately 70 days. Hence, performance is still an issue and the obvious approach to improve efficiency would be to figure out further parallelisation

possibilities which may concern the framework as well as the implementations of the single simulation models. For instance, the coordination could be made more liberal if in addition to the local time steps individual dependencies and independencies of simulation models would be taken into account. The models themselves may also identify further parallelisable parts, though we have already provided templates for parallel computations of different proxels.

7 Conclusion

We have described 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. It allows us to construct in a flexible way networks of distributed, dependent simulation models which are concurrently executed. The framework has been successfully applied to construct the integrative simulation system DANUBIA which integrates 15 simulation models for natural and socio-economic processes.

For the development of the framework we have investigated a view-based methodology which, we believe, can be useful for the development of other complex software systems as well under the following assumptions: First, a partition of the functionality into several prominent views must be meaningful, secondly it should be possible to identify a common base view such that the other views are othogonal extensions of the base, and, for applying our refinement and extension relations, the static, dynamic and component models must conform to the excerpt of the UML metamodel used in our approach. Actually, we are looking for further case studies to apply our methodology which finally should be supported by tools for various tasks. For instance, to manage views, check refinements and extensions, and to compute integrations and reference implementations.

We have applied formal specification techniques to specify and check the temporal coordination being the heart of integrative simulations with dependable models. We are not aware of any other system of comparable complexity which has been completely modelled and specified in such a rigorous manner up to the last step, in which a full, implementation language independent model of the whole system is constructed. The models and specifications serve at the same time as a complete documentation for maintenance, furher developments and adaptations of the framework. The framework as well as the simulation models developed in the GLOWA-Danube project have been published under the name OPENDANUBIA under an Open Source Licence. Thus the framework is accessible for model developers for instantiation and also for framework developers, who may want to add further features (e.g., to support dynamic changes of simulation configurations). More information about OPENDANUBIA and a comprehensive list of publications discussing the application of the framework for particular scenarions and simulation results from various perspectives can be found at the GLOWA-Danube web page [26].

Acknowledgement. We would like to thank the anonymous reviewers of this paper for their careful reading and for their useful suggestions for improvement.

References

1. Allen, R., Garlan, D.: A formal basis for architectural connection. ACM Trans. Softw. Eng. Methodol. 6(3), 213–249 (1997)
2. Allen, R., Garlan, D., Ivers, J.: Formal modeling and analysis of the HLA component integration standard. In: Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT 1998/FSE-6), pp. 70–79 (1998)
3. Barth, M., Knapp, A.: A coordination architecture for time-dependent components. In: Proc. 22nd Int. Multi-Conf. Applied Informatics. Software Engineering (IASTED SE 2004), pp. 6–11 (2004)
4. Bowles, J.K.F., Bordbar, B.: A Formal Model for Integrating Multiple Views. In: International Conference on Application of Concurrency to System Design, pp. 71–79 (2007)
5. Cengarle, M.V., Knapp, A., Mühlberger, H.: Interactions. In: Lano [19], ch. 9, pp. 205–248
6. Dahmann, J.S., Fujimoto, R., Weatherly, R.M.: The department of defense high level architecture. In: Winter Simulation Conference, pp. 142–149 (1997)
7. D'Souza, D., Wills, A.: Objects, Components and Frameworks with UML – The Catalysis Approach. Addison-Wesley, Reading (1999)
8. Giupponi, C., Jakeman, A.J., Karssenberg, D., Hare, M.P. (eds.): Sustainable Management of Water Resources – An Integrated Approach. Edward Elgar Publishing, Cheltenham (2006)
9. Gregersen, J.B., Gijsbers, P.J.A., Westen, S.J.P.: OpenMI: Open modelling interface. Journal of Hydroinformatics 9(3), 175–191 (2007)
10. Haugen, Ø., Husa, K., Runde, R., Stølen, K.: STAIRS towards formal design with sequence diagrams. Software and Systems Modeling 4, 355–357 (2005), 10.1007/s10270-005-0087-0
11. Haugen, Ø., Stølen, K.: STAIRS – Steps To Analyze Interactions with Refinement Semantics. In: Stevens, P., Whittle, J., Booch, G. (eds.) UML 2003. LNCS, vol. 2863, pp. 388–402. Springer, Heidelberg (2003)
12. Hennicker, R., Ludwig, M.: Property-Driven Development of a Coordination Model for Distributed Simulations. In: Steffen, M., Zavattaro, G. (eds.) FMOODS 2005. LNCS, vol. 3535, pp. 290–305. Springer, Heidelberg (2005)
13. Hillyer, C., Bolte, J., van Evert, F., Lamaker, A.: The ModCom modular simulation system. European Journal of Agronomy 18(3), 333–343(11) (2003)
14. Hitz, M., Kappel, G., Kapsamer, E., Retschnitzegger, W.: UML@Work – Objektorientierte Modellierung mit UML 2 (3. Auflage). dpunkt.verlag, Heidelberg (2005)
15. Jacobson, I., Booch, G., Rumbaugh, J.: The Unified Modeling Language User Guide, 2nd edn. The Addison-Wesley Object Technology Series. Addison-Wesley (2005)
16. Jagers, H.R.A.: Linking Data, Models and Tools: An Overview. In: Swayne, D.A., Yang, W., Voinov, A.A., Rizzoli, A., Filatova, T. (eds.) Proceedings of the iEMSs Fifth Biennial Meeting: International Congress on Environmental Modelling and Software (iEMSs 2010), Ottawa, Canada. International Environmental Modelling and Software Society (July 2010)
17. Klein, J., Caillaud, B., Hélouet, L.: Merging Scenarios. In: 9th International Workshop on Formal Methods for Industrial Critical Systems (FMICS), Linz, Austria, pp. 209–226 (September 2004)

18. Kralisch, S., Krause, P., David, O.: Using the Object Modeling System for hydrological model development and application. Advances in Geosciences 4, 75–81 (2005)
19. Lano, K. (ed.): UML 2 Semantics and Applications. John Wiley & Sons (2009)
20. Letcher, R.A., Bromley, J.: Typology of Models and Methods of Integration. In: Giupponi, et al. [8], vol. 11, pp. 287–323
21. Ludwig, M.: Modelling and Architecture of a Generic Framework for Integrative Environmental Simulations. Berichte aus der Informatik. Shaker, Aachen (2011)
22. Ludwig, R., Mauser, W., Niemeyer, S., Colgan, A., Stolz, R., Escher-Vetter, H., Kuhn, M., Reichstein, M., Tenhunen, J., Kraus, A., Ludwig, M., Barth, M., Hennicker, R.: 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)
23. Magee, J., Kramer, J.: Concurrency: state models & Java programming, 2nd edn. Wiley, Chichester (2006)
24. Rahman, J.M., Seaton, S.P., Perraud, J.-M., Hotham, H., Verrelli, D.I., Coleman, J.R.: It's TIME for a New Environmental Modelling Framework. In: Proceedings of MODSIM 2003 International Congress on Modelling and Simulation, Townsville, Australia, vol. 4. Modelling and Simulation Society of Australia and New Zealand Inc. (July 2003)
25. Tenhunen, J.D., Kabat, P. (eds.): Integrating Hydrology, Ecosystem Dynamics, and Biogeochemistry in Complex Landscapes. Wiley, Chichester (1999)
26. GLOWA-Danube Project Website, `http://www.glowa-danube.de` (last visited May 10, 2011)
27. Warmer, J., Kleppe, A.: The Object Constraint Language, 2nd edn. Addison-Wesley (2003)
28. Winskel, G., Nielsen, M.: Models for concurrency. In: Abramsky, S., Gabbay, D.M., Maibaum, T.S.E. (eds.) Handbook of Logic in Computer Science, vol. 4, pp. 1–148. Oxford University Press, Oxford (1995)