Design and Implementation of a Coordination Model for Distributed Simulations *

Rolf Hennicker, Matthias Ludwig

Institut für Informatik
Ludwig-Maximilians-Universität München
Oettingenstrasse 67
D-80538 München, Germany

Abstract: The coordination of time-dependent simulation models is an important problem in environmental systems engineering. We propose a design model based on a formal specification using the CSP-like language FSP of Magee and Kramer. The heart of our design model is a global timecontroller which coordinates distributed simulation models according to their local time scales. We show how a systematic transition from the design model to a Java implementation can be derived from the design model. The strong practical relevance of the approach is ensured by the fact that our strategy is used to produce the kernel of the integrative simulation system DANUBIA developed within the GLOWA-Danube project.

1 Introduction

In the last decade environmental systems engineering became an important application area for information and software technology. Setting out from geographical information systems and GIS-based expert systems nowadays one is particularly interested in the development of integrative systems with a multilateral view of the world in order to understand better the mutual dependencies between environmental processes. Of particular importance are water-related processes which have an 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.

There are several projects dealing with methods, techniques and tools to support a sustainable water resource management, for instance within the European research activity EESD (Energy, Environment and Sustainable Development, cf. [3]) or within the German initiative GLOWA (Global Change in the Hydrological Cycle; cf. [4]). Within the GLOWA framework the project GLOWA-Danube [8] deals with the Upper Danube watershed as a representative area for mountain-foreland regions. The principle objective of GLOWA-Danube is to develop new techniques of coupled distributed simulations that allow to

*This work is partially supported by the GLOWA-Danube project (01LW0303A) sponsored by the German Federal Ministry of Education and Research.

integrate simulation models of various disciplines in order to study water-related global change scenarios. For this purpose the integrative simulation system DANUBIA is developed which is designed as an open, distributed network integrating the simulation models of all socio-economic and natural science disciplines taking part in GLOWA-Danube. Actually seventeen simulation models, either implemented directly in Java or surrounded by a Java wrapper, are integrated in the DANUBIA system, covering the disciplines of meteorology, hydrology, remote sensing, ground- and surface water research, glaciology, plant ecology, environmental psychology, environmental and agricultural economy, and tourism. As a result of coupled simulations transdisciplinary effects of mutually dependent processes can be analysed and evaluated. For example, the agricultural economy model determines sowing and harvesting dates of different crops, while the plant ecology model simulates growing of the plants, dependent on the precipitation provided by an atmosphere model. After the harvesting date the plant ecology model returns the crop yield to the agricultural economy model, which in turn is used for future calculations of the farmers.

An important characteristics of DANUBIA is the possibility to perform integrative simulations where the single simulation models run concurrently and exchange information at runtime. Since any integrative simulation models water-related processes over a specific period of time (usually several years) and since each simulation model has an individual local time step in which computations are periodically executed (ranging from hours, like in meteorology, to months, like in social sciences) the distributed models must be coordinated to work properly together. For this purpose it must be guaranteed that during the simulation run

- all values accessed through model interfaces are in a stable state (which corresponds to the usual read/write exclusion) and, moreover, that
- every simulation model is supplied with valid data, i.e. with data that fits to the local model time of the importing simulation model.

This informal description of the coordination problem provides only an intuitive idea of the requirements for integrative simulations. In [5] the authors have presented a formalization of the coordination problem[1] and a formal design model which are both specified in terms of the language FSP (Finite State Processes) introduced by Magee and Kramer [9]. In particular, it has been shown by model checking techniques that the coordination requirements are satisfied by the design model. The basic idea of the design model is to introduce a global timecontroller which stores the current status of all simulation models participating in an integrative simulation in order to coordinate them appropriately. Technically, the timecontroller and the single simulation models are represented by FSP-processes and the simulation system itself is represented by the parallel composition of the single processes which are synchronized through appropriate shared actions.

In this work we demonstrate how the formal design model can be systematically transformed into a UML implementation model which can be directly realized by a Java program. For this purpose we first classify the given FSP-processes such that all processes

[1] An alternative formalization on a meta level using purely mathematical notations is given in [2].

representing simulation models are considered as active objects and the timecontroller is considered as a reactive object. The shared actions used in the FSP-model for synchronization are translated into synchronized methods offered by the timecontroller which must be called by a simulation model whenever the model wants to get data from other models or to provide data for other models. Hence the timecontroller is realized by a Java monitor object and the single simulation models are realized by concurrently executing Java threads. In order to abstract from concrete computations that a simulation model performs and which are not relevant for the coordination we develop an appropriate system architecture such that the developer of a concrete DANUBIA simulation model must only extend an appropriate abstract model class provided by the DANUBIA framework.

The paper is organized as follows: We start, in section 2, with a brief introduction to FSP. In section 3, we describe the coordination problem and, in section 4, we provide a solution in terms of a formal design model represented by FSP processes. In section 5 the design model is transformed into a Java implementation that leads to a flexible system architecture for the kernel of the DANUBIA system.

2 A Brief Introduction to FSP

The language FSP has been introduced by Magee and Kramer as a formalism for modeling concurrent processes. An elaborated description of the syntax and semantics of FSP can be found in [9]. Syntactically FSP resembles CSP [6]. Essential constructs for building FSP processes are

STOP	process termination
$(a \to P)$	action prefix
$(a \to P \mid \mathrm{when}\ (cond)\ b \to Q)$	choice (involving a guarded action)
$P + \{a_1, \ldots, a_n\}$	alphabet extension
$P/\{new_1/old_1, \ldots, new_n/old_n\}$	action relabeling
$(P\|\|Q)$	parallel composition
$P \setminus \{a_1, \ldots, a_n\}$	hiding
$P@\{a_1, \ldots, a_n\}$	interface definition

Each process P has an alphabet, denoted by αP, consisting of those actions in which the process can be engaged. For instance, a process $(a \to P)$ obtained by action prefix first engages in the action a and then behaves like the process P. If we build the parallel composition $(P||Q)$ then actions that are shared by P and Q (i.e., belong to αP and αQ) must be performed simultaneously. For the non-shared actions interleaving semantics of parallel processes is used. The hiding operator allows to hide certain actions which are then invisible and represented by τ. The construction of an interface is the complement of hiding.

Processes can be defined by process declarations of the form $P = E$ or, in the case of parallel processes, by $||P = (E||F)$. A (non-parallel) process declaration can be recursive and can involve local, indexed processes of the form

$$P = Q[value],$$
$$Q[i : T] = E.$$

where T is a (finite) type and i is an index variable of type T.

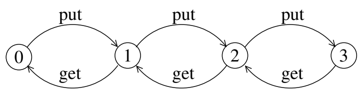
Often we will use indexed actions of the form $a[i]$. A shorthand notation for a choice over a finite set of indexed actions is $(a[T] \to P)$, which is equivalent to $(a[x] \to P \mid \ldots \mid a[y] \to P)$, where **range** $T = x..y$. We will also use labeled actions of the form $[label].a$ and choice over a finite set of labeled actions $[T].a$ with T as above. To obtain several copies of a process P we use process labeling $[label] : P$ which denotes a process that behaves like P with all actions labeled by $[label]$.

The semantics of a process is given by a finite labeled transition system (LTS) which can be pictorially represented by a directed graph whose nodes are the process states and whose edges are the state transitions labeled with actions. Since FSP is restricted to a finite number of states one can automatically check safety and progress properties of processes with the LTSA tool [7].

The following example shows an FSP model of a simple producer/consumer system with a bounded buffer. The bounded buffer is modeled by a parameterized FSP process BUFFER whose formal parameter MAX has the default value 1. The definition of the BUFFER process uses local, indexed processes BUF[i].

```
PRODUCER = (put -> PRODUCER).
CONSUMER = (get -> CONSUMER).

BUFFER(MAX=1) = BUF[0],
BUF[i:0..MAX] =
  (when (i<MAX) put -> BUF[i+1]
  |when (i>0)   get -> BUF[i-1]).

||SYS = (PRODUCER||CONSUMER||BUFFER(3)).
```

Note that in the composite process SYS the formal parameter of the BUFFER process is instantiated by 3. Hence, the semantics of the composite process is given by the following LTS:



3 The Coordination Problem of Integrative Simulations

A simulation model simulates a physical or social process for a finite period of time which we call simulation time. Typically a simulation model does not work on a continuous but on a discrete time scale. Thus the simulation period is represented by a strictly ordered, discrete set of points in time (denoted by natural numbers), at which data is provided by a simulation model. Each model has an individual time step (the distance between two subsequent simulation points) which depends on the simulated process. We assume that the time step of a model remains fixed during the whole simulation. Abstraction from the concrete simulated process of a simulation model leads us to the following common life cycle of each model within an integrative simulation:

After a simulation model has been started it provides first some initial data. Then it performs periodically the following steps until the end of the simulation is reached:

1. Get required data from other models.
2. Compute new data which are valid at the next simulation point.
3. Provide the newly computed data.

We can model this behavior by the following (generic) FSP process which is parameterized w.r.t. the individual time step of a simulation model. Note that in the process definition we 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.

```
const simStart = 0
const simEnd = 6
range Time = simStart..simEnd

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] -> enterProv[t+step] ->
        prov[t+step] -> exitProv[t+step] -> M[t+step])
  else  STOP.
```

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. The actions `get[x]` and `prov[x]` are enclosed by corresponding `enter` and `exit` actions which are needed for the coordination of concurrently running simulation models.

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 (cf. Section 2) and the time step of a model is determined by an actual parameter. For instance, the FSP processes $[1] : \mathrm{MODEL}(2)$ and $[2] : \mathrm{MODEL}(3)$ represent two simulation models, one with number 1 and time step 2 and the other one with number 2 and time step 3, respectively.

In an integrative simulation several simulation models are coupled in the sense that they mutually exchange data among each other at runtime. Data exchange is performed via a port which holds data that is valid at a particular point in time. Since in each time step a huge amount of data is produced previous values will be overwritten in each computation cycle. Hence the different models must be coordinated such that the following conditions are satisfied:

(C1) Whenever simulation models exchange data, the values must be in a stable state.

(C2) Every simulation model must be supplied with valid data, i. e. with data that fit to the local model time of the importing simulation model.

Condition (C1) corresponds to the well-known read/write exclusion which in our context means that `get` and `prov` actions must be mutually exclusive. The critical condition is (C2) which becomes quite complex if we consider arbitrarily many simulation models. We can, however, simplify the problem, if we consider only two simulation models at a time and, moreover, if we consider each of the two models only under one particular role, 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 requirement (R1), from the provider's point of view we obtain requirement (R2).

(R1) 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 $t_U < t_P$.

(R2) 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 $t_U \geq t_P$.

(R1) ensures that a user does not get obsolete data, (R2) guarantees that a provider does not overwrite data which is still needed. An execution trace w of an integrative simulation with an arbitrary number of simulation models $[1]$: $\mathrm{MODEL}(\mathit{Step}_1), \ldots, [n]$: $\mathrm{MODEL}(\mathit{Step}_n)$ is called *legal*, if w meets the above requirements (R1) and (R2) for all pairwise combinations of models considered as users and as providers.

4 Formal Design Model for Integrative Simulations

In this section we present a solution of the coordination problem by providing a formal design model in terms of FSP processes. The basic idea is to introduce a global timecontroller 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

$$\begin{aligned} ||\mathrm{SYS} \quad = \quad & ([1] : \mathrm{MODEL}(\mathit{Step}_1) || \ldots || [n] : \mathrm{MODEL}(\mathit{Step}_n) || \\ & \mathrm{TIMECONTROLLER}(\mathit{Step}_1, \ldots, \mathit{Step}_n)) / \{\mathrm{start}/[\mathrm{Models}].\mathrm{start}\} \end{aligned}$$

with *range* $\mathrm{Models} = 1..n$ restricts the execution traces of the uncontrolled simulation models to the legal ones. The relabeling clause `/{start/[Models].start}` ensures that the processes synchronize on the `start` action. The composite process SYS is then considered as the design model for the simulation system. The (static) structure of SYS is represented by the diagram in figure 1 which indicates the required communication links.

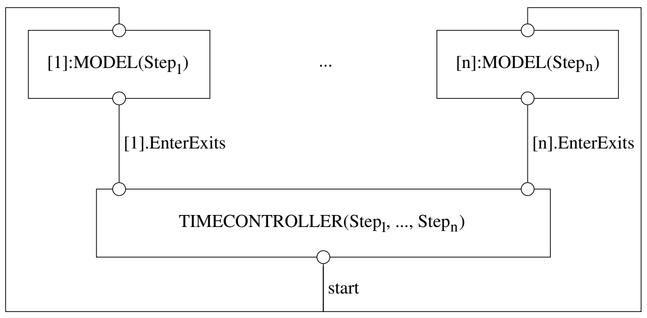


Figure 1: Structure diagram of the design model

The communication links show that each simulation model m communicates with the timecontroller via the shared `enter` and `exit` actions in the (labeled) set `[m].Enter-Exits`, where

```
set EnterExits =
  {{enterGet,exitGet,enterProv,exitProv}[Time]}.
```

This means that the simulation models synchronize with the timecontroller on actions of the form `[m].enterGet[t]` etc., where $m \in \mathrm{Models}$ and $t \in \mathrm{Time}$. It is then the task of the timecontroller to guarantee that synchronization can only occur if the constraints for integrative simulations described in Section 3 are satisfied. For this purpose the `enter` actions of the timecontroller are guarded by appropriate conditions which monitor the validity of the constraints. To express the necessary conditions in FSP the timecontroller is equipped with a local state (modeled 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 timecontroller definition is formulated for the case of two simulation models where the time steps of the two models are given by parameters (with default value 1). 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 will, of course, be used in the Java implementation). Let us still remark that the guards of the `enterGet` actions are inferred from requirement (R1) by considering each model as a potential provider and the guards of the `enterProv` actions are inferred from requirement (R2) by considering each model as a potential user.

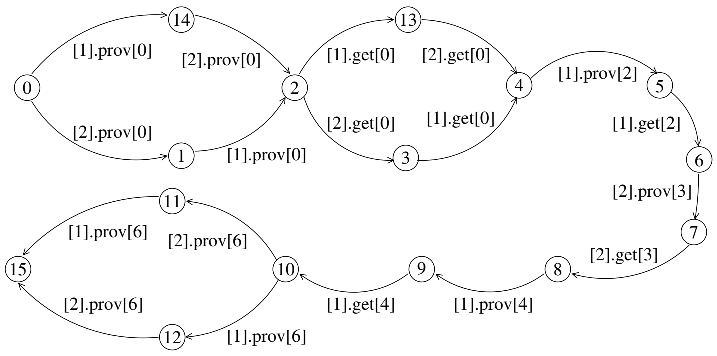
```
const nrModels = 2
range Models = 1..nrModels

TIMECONTROLLER(modelStep1=1,modelStep2=1) =
  (start -> TC[simStart][simStart][simStart][simStart]),

TC[nextGet1:Time][nextProv1:Time]
      [nextGet2:Time][nextProv2:Time] =
  (dummy[t:Time] ->
    //enterGet
    (when (t<nextProv1 & t<nextProv2)
      [Models].enterGet[t] ->
      TC[nextGet1][nextProv1][nextGet2][nextProv2]
    //exitGet
    |[1].exitGet[t] ->
      TC[t+modelStep1][nextProv1][nextGet2][nextProv2]
    |[2].exitGet[t] ->
      TC[nextGet1][nextProv1][t+modelStep2][nextProv2]
    //enterProv
    |when (nextGet1>=t & nextGet2>=t)
      [Models].enterProv[t] ->
      TC[nextGet1][nextProv1][nextGet2][nextProv2]
    //exitProv
    |[1].exitProv[t] ->
      if (t+modelStep1<=simEnd)
      then TC[nextGet1][t+modelStep1][nextGet2][nextProv2]
      else TC[simStart][simStart][simStart][simStart]
    |[2].exitProv[t] ->
      if (t+modelStep2<=simEnd)
      then TC[nextGet1][nextProv1][nextGet2][t+modelStep2]
      else TC[simStart][simStart][simStart][simStart]
    |dummy[t] ->
      TC[nextGet1][nextProv1][nextGet2][nextProv2])
  )\{dummy[Time]}.
```

Let us still mention that the actions `dummy[t:Time]` are only introduced for technical reasons, such that the index variable t is known where necessary. The `dummy` actions are finally made invisible by applying the hiding operator.

As an example, the design model of a distributed simulation with two simulation models with time steps 2 and 3 resp. is given by the following composite process SYS where the formal paramaters of the single, parallel processes are instantiated appropriately.

```
const stepModel1 = 2
const stepModel2 = 3
||SYS =
   ([1]:MODEL(stepModel1)||[2]:MODEL(stepModel2)||
    TIMECONTROLLER(stepModel1,stepModel2))
      /{start/[Models].start}.
```

We cannot visualize the labeled transition system of the process SYS because it has too many states and transitions. However, for an analysis of the behavior of the design model we can consider different views on the system which can be formally defined by means of the FSP 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` is defined as `set GetProvs =` {{`get,prov`}`[Time]`}. The corresponding LTS, after minimalization w.r.t. invisible actions, is shown in the following diagram.



In [5] we have formalized the coordination requirements (R1) and (R2) by so-called property processes of FSP and we have shown by model checking with the LTSA tool that the timecontroller-based design model is a correct solution of the coordination problem.

5 Implementation

In this section we will show how to derive in a systematic way a Java implementation from the timecontroller-based design model. In principle, many steps of the translation procedure, which follows the pragmatic ideas of [9], 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.

5.1 Static Structure

Let us first consider the static structure of the implementation model which is given by the UML class diagram in figure 2. In the following we will explain how this diagram evolved from the design model.

At first the processes TIMECONTROLLER, MODEL and SYS give rise to the classes `Timecontroller`, `Model` and `Sys` respectively. As the process MODEL runs through

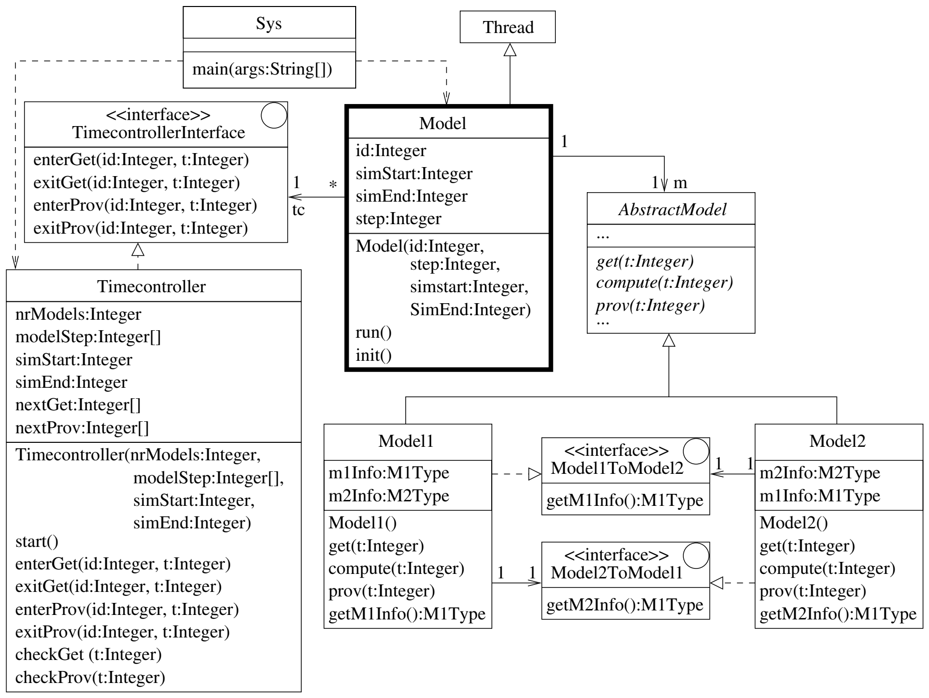


Figure 2: Class diagram of the implementation model

the model's life cycle, the class `Model` is an active class and inherits from the Java class `Thread`. All actions of the MODEL process are considered as output actions which correspond to method calls on a `Timecontroller` object on the one hand (`enterGet`, `exitGet`, `enterProv`, `exitProv`) and on a concrete simulation model object on the other hand (`get`, `compute`, `prov`). These methods are extracted to the interface `TimecontrollerInterface` and the abstract class `AbstractModel` respectively. A `Model` instance communicates with the `Timecontroller` via synchronous method calls which are specified by the interface `TimecontrollerInterface`. Note that one `Timecontroller` instance controls arbitrarily many `Model` instances which is denoted by the multiplicities 1 and * at the respective ends of the (directed) association to `TimecontrollerInterface`.

Let us briefly explain how a concrete simulation model is integrated into the DANUBIA system. DANUBIA provides a core system that is divided into a common runtime environment on the one hand, and a framework containing classes to be used by individual model developers on the other hand (the so-called developer framework). For example the implementation of a model's life cycle in the class `Model` is part of the runtime environment, while the class `AbstractModel` is part of the developer framework (cf. figure 3). To integrate a concrete model its developer has to build a subclass of `AbstractModel` and to implement the abstract methods *get*, *compute*, *prov*, the so called plug points. E.g.

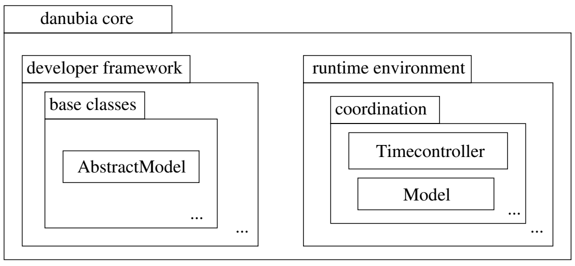


Figure 3: Components of the DANUBIA core system (extract)

the plug point *compute* allows to realize the concrete computation algorithm of a model. Besides the plug points depicted in figure 2 there are some further plug points (e.g. for simulation recovery) which are out of the scope of this paper. Let us now consider the attributes of the classes `Timecontroller` and `Model`. The attribute `nrModels` of the class `Timecontroller` corresponds to the constant `nrModels` of the FSP process. The array `modelStep` contains the individual time steps of the models involved in the simulation. It complies with the parameters of the TIMECONTROLLER process. The attributes `nextGet` and `nextProv` correspond to the indices of the local processes TC. For the class `Model` the attributes result as follows. While `step` derives from the parameter of the process MODEL and `simStart` and `simEnd` are derived from global constants of the FSP model, the model identifier `id` represents the process label in the composite process SYS.

Data exchange between distributed simulation models is performed by corresponding import and export ports. These ports are realized by interfaces (`Model1ToModel2`, `Model2ToModel1`) on the one hand and by attributes (`m1Info`, `m2Info`) which store a spatial set of data (a so-called data table) on the other hand. Let us consider this from the viewpoint of the class `Model1`. The interface `Model1ToModel2` acts as an export interface and `Model2ToModel1` as an import interface for `Model1`. The attribute `m1Info` is the corresponding export table which is updated with new data each time the method `prov` is called and returned when the method `getM1Info` is invoked (by `Model2`). Vice versa the attribute `m2Info` is an import table that is updated within the `get` method by invoking `getM2Info` on the import interface. Network communication between distributed simulation models is realized by the Java Remote Method Invocation (RMI) technology and hidden from the model developer by appropriate classes in the runtime environment.

5.2 Dynamic Behavior

Let us now consider the dynamic behavior of the single entities. Since `Model` is an active class we have to implement its `run` method. For this purpose we translate the actions of the MODEL process into appropriate method calls. The `while` loop corresponds to the conditional recursive call of the local FSP process M. The actions of the local process

INIT are extracted to a private method `init`. To make the code better readable we abstain from a proper exception handling here.

```
public void run() {
  init();
  int t=0;
  while (t+step<=simEnd) {
    try {
      tc.enterGet(id, t);
    } catch (InterruptedException e) {}
    m.get(t);
    tc.exitGet(id, t);
    m.compute(t);
    try {
      tc.enterProv(id, t+step);
    } catch (InterruptedException e) {}
    m.prov(t+step);
    tc.exitProv(id, t+step);
    t = t+step; } }

private void init() {
  try {
    tc.enterProv(id, 0);
  } catch (InterruptedException {}
  prov(0);
  tc.exitProv(id, 0); }
```

The TIMECONTROLLER process is implemented as a passive entity that reacts on method calls. The class `Timecontroller` acts as a monitor whose (relevant) state is determined by the values of the array attributes `nextGet` and `nextProv` which store the information about the current progress of the models involved in the simulation. These attributes are initialized within the method `start`, such that each array entry is set to `simStart`. The remaining fields of the timecontroller are initialized within the constructor of the class `Timecontroller`. All public methods of the class `Timecontroller` offered by its interface are synchronized methods which are implemented according to the behavior specified in the TIMECONTROLLER process. For the implementation of the guarded enter and exit actions we apply a transformation rule provided in [9] which translates an FSP expression of the form

```
when (cond) op -> MONITOR[nextState]
```

with some condition `cond` and action `op` into the following Java code:

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

The action `op` is implemented by the synchronized method `op`. 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 all waiting threads by `notifyAll`. Note that the `while` loop ensures that the condition is checked again after a thread has been released. We demonstrate the application of this rule with the `enterGet` action. The action `enterProv` is implemented analogously. The TIMECONTROLLER specification reads

```
...
   when (t<nextProv1 & t<nextProv2)
      [Models].enterGet[t] ->
        TC[nextGet1][nextProv1][nextGet2][nextProv2]
...
```

This results in the following implementation:

```
public synchronized void enterGet(int id, int t)
  throws InterruptedException {
    while (!checkProv(t)) wait(); }
```

Note that a call of `notifyAll` is not necessary here, since the monitor state is not changed in this method. Note also that the action label in `Models` which denote the model identifiers and the index `t` denoting the model time are translated into method parameters. For reasons of better readability the implementation of the guard condition is outsourced to the private auxiliary method `checkProv`.

```
private boolean checkProv(int t) {
  boolean b = true;
  for (int i = 0; i < nrModels; i++) {
    b = (b && (t < nextProv[i])); }
  return b; }
```

In contrast to the enter actions, for the exit actions `exitGet` and `exitProv` no guard is provided, but the timecontroller changes its state. As an example, let us translate the `exitGet` action. Since FSP does not allow arrays we had to specify the effect of the action for each model separately:

```
...
   |[1].exitGet[t] ->
     TC[t+modelStep1][nextProv1][nextGet2][nextProv2]
   |[2].exitGet[t] ->
     TC[nextGet1][nextProv1][t+modelStep2][nextProv2]
...
```

By taking advantage of arrays the two lines are subsumed by

```
public synchronized void exitGet(int id, int t) {
  nextGet[id-1]=nextGet[id-1]+modelStep[id-1];
  notifyAll(); }
```

Note that we must subtract 1 from the array index to match the correct model identifier.

Finally let us consider the class `Sys` which represents the composite process SYS.

```
const stepModel1 = 2
const stepModel2 = 3
||SYS = ([1]:MODEL(stepModel1)||[2]:MODEL(stepModel2)||
   TIMECONTROLLER(stepModel1,stepModel2))
     /{start/[Models].start}.
```

Within its sole method `main(String[] args)` the global constants `simStart` and `simEnd` give rise to local variables with the same names. Furthermore a local array variable `modelStep` is filled with the time steps of the participating simulation models, where the array indices correspond to the model identifiers. The essential task of the main method is to create and start a `Timecontroller` object and a number of `Model` objects with appropriate actual parameters. The call of the `start` method on each created object corresponds to the synchronization of the `start` actions in the FSP process which is expressed by the relabelling clause `/{start/[Models].start}`.

```
public static void main(String[] args) {
  int simStart = 0;
  int simEnd = 6;
  int nrModels = 2;
  int[] modelStep = new int[] { 2, 3 };
  Timecontroller tc =
    new Timecontroller(nrModels, modelStep, simStart, simEnd);
  tc.start();
  new Model(1, 2, simStart, simEnd, tc,
            new Model1()).start();
  new Model(2, 3, simStart, simEnd, tc,
            new Model2()).start(); }
```

6 Conclusion

We have shown how to construct an implementation of a formal design model for time-dependent integrative simulations. For this purpose we have applied a general translation scheme which transforms simulation models represented by FSP processes into concurrently executing threads and the global timecontroller process into a monitor object with appropriately synchronized methods. In order to hide the coordination problem from the developers of concrete simulation models we have proposed a system architecture which allows to plug in a specific simulation model by implementing particular plug points provided by the DANUBIA developer framework.

The strategy proposed here to coordinate coupled simulations is different from the approach pursued in the OpenMI Standard [10] where single simulation models do not follow a global time control but act autonomously by requesting data from other models whenever needed. Then, if no data for the respective point in time is yet available, it will be estimated by inter- or extrapolation.

Our approach can be applied to all kinds of systems where concurrently executing components must be coordinated in accordance with some discrete order. Within the GLOWA-Danube project the approach is of high practical relevance for the development of the DANUBIA system because integrative simulations are the heart of all current and future features of DANUBIA and hence the reliability of the whole system depends on the correctness of the coordination implementation.

Acknowledgement

We are grateful to anonymous referees of this paper for helpful comments and suggestions.

References

[1] Barth M., Hennicker R., Kraus A., Ludwig M.: DANUBIA: An Integrative Simulation System for Global Research in the Upper Danube Basin. Cybernetics and Systems, Vol. 35, Nr. 7–8, pages 639–666, 2004.

[2] Barth M., Knapp A.: A Coordination Architecture for Time-Dependent Components. Proc. 22nd Int. Multi-Conf. Applied Informatics. Software Engineering (IASTED SE'04), pages 6–11, 2004.

[3] EESD, `http://www.cordis.lu/eesd` (last visited 2006/02/17)

[4] GLOWA, `http://www.glowa.org` (last visited 2006/02/17)

[5] Hennicker R., Ludwig, M.: Property-Driven Development of a Coordination Model for Distributed Simulations. Proceedings of the 7th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2005), LNCS 3535, pp. 290–305. Springer, 2005.

[6] Hoare, C. A. R.: Communicating Sequential Processes, Prentice-Hall, 1985.

[7] LTSA, `http://www-dse.doc.ic.ac.uk/concurrency/` (last visited 2006/02/17)

[8] 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, Vol. 28, pages 621–634, 2003.

[9] Magee J., Kramer J.: Concurrency — State Models and Java Programs, John Wiley & Sons, 1999.

[10] Moore R., Tindall I., Fortune D.: Update on the Harmonit Project — The OpenMI Standard for Model Linking. Proceedings of the 6th International Conference on Hydroinformatics, Vol. 2, pages 1811–1818, 2004.