Property-Driven Development of a Coordination Model for Distributed Simulations *

Rolf Hennicker and Matthias Ludwig

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

Abstract. The coordination of time-dependent simulation models is an important problem in environmental systems engineering. We propose a solution based on a rigorous formal modelling of the participating processes. Methodologically, our approach is driven by property processes which are used for the formal specification of the coordination problem. Property processes are supported by the CSP-like language FSP of Magee and Kramer which will be used throughout this paper for modelling the system requirements and the system design. The heart of our design model is a global time controller which coordinates distributed simulation models according to their local time scales. We will show with model checking techniques that all safety and liveness requirements are guaranteed by the timecontroller design.
The strong practical relevance of the approach is ensured by the fact that our strategy is used to produce a formally verified design for 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,

* This work is partially supported by the GLOWA-Danube project (07GWK04) sponsored by the German Federal Ministry of Education and Research.

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 identify, examine and develop new techniques of coupled distributed modelling for the integration of natural and socio-economic sciences. For this purpose the integrative simulation system DANUBIA is developed which supports the analysis of water-related global change scenarios. DANUBIA 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 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.

An important characteristics of DANUBIA is the possibility to perform integrative simulations where the single simulation models run concurrently and exchange information at run time. Since any simulation models water-related processes over a specific period of time (usually some years) a global time control is necessary which coordinates the distributed models to work properly together. This is a non-trivial task since each simulation model has an individual time step in which computations are periodically executed ranging from hours, like in meteorology, to months, like in social sciences. To ensure that an integrative simulation provides reliable results 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 synchronization conditions provides only an intuitive idea of the coordination problem to be considered. For a full understanding it is necessary to clarify several issues, like the notion of time and the precise timing conditions for correct data exchange on the basis of the local time scales of the cooperating models. Taking into account that a distributed simulation is an open system where in principal arbitrarily many models (with different time scales) can participate it is obvious that the coordination problem soon becomes untractable without the use of formal specification techniques.

An example of a formalization of the coordination problem on a meta level using purely mathematical notations is given in [2]. Here we will use as a specification formalism the language FSP (Finite State Processes) of Magee and Kramer [9] which provides an appropriate basis for applying model checking techniques. Moreover, FSP allows us to follow a property-oriented approach where first the system requirements are specified by means of so-called property processes. The use of property processes has several advantages which are

essential for our application. First, a requirements specification can be developed piecewise by collecting single property processes which focus on one aspect of the system at a time. This is particularly useful for the coordination problem where it is sufficient to consider the cooperating simulation models pairwise and under different roles, one model acting (only) as a provider and the other one acting (only) as a user of information. In this way the complexity of the problem can be drastically reduced. Also the exclusion condition for providing and retrieving data can be specified by a seperate property process. However, as usual, there is still a danger that the requirements are not adequately met by the single property processes. To deal with this issue FSP assigns to each property process a finite labelled transition system which can be animated with the LTSA tool (Labelled Transition System Analyzer; cf. [7]). Thus we can reveal the legal and illegal execution paths which is indeed helpful to analyse and validate whether the single property processes reflect correctly the desired time dependent coordination constraints. In addition to the property processes which represent safety conditions we also specify liveness conditions stating that each simulation model must repeatedly provide data during the whole simulation period according to its local time scale.

The requirements specification developed in this paper is a good example of a highly non-constructive formal specification in the sense that it cannot be directly transformed into an executable program. In the next step we will focus on the design of a constructive solution for the coordination problem. For this purpose we define a global timecontroller process which stores the current status of all simulation models in order to coordinate them appropriately. The design of the whole simulation system is then given by the parallel composition of the timecontroller and all concurrent simulation models. It is shown with model checking techniques that the design model indeed satisfies the desired safety and liveness properties. All processes occurring in the system design are also represented in terms of FSP notation and model checking is performed with the LTSA tool. The separation of property specifications from design is of great methodological value for our application. This approach is well supported by FSP but not e.g. by SPIN [6] or related model checkers where it is necessary to integrate the assertions into a given design model.

The proposed 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.

2 A Brief Introduction to FSP

The language FSP has been introduced by Magee and Kramer as a formalism for modelling concurrent processes. An elaborated description of the syntax and

semantics of FSP can be found in [9]. Syntactically FSP resembles CSP [5]. Frequently used constructs for building FSP processes are

STOP	process termination
$(a \rightarrow P)$	action prefix
$(a \rightarrow P \mid \text{when } (cond)\ b \rightarrow Q)$	choice (involving a guarded action)
$P + \{a_1, \ldots, a_n\}$	alphabet extension
$(P\|\|Q)$	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. 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 $(a[T] \rightarrow P)$, which is equivalent to $(a[x] \rightarrow P \mid \ldots \mid a[y] \rightarrow P)$, where **range** $T = x..y$. We will also use labelled actions of the form $[label].a$ and choice over a finite set of labelled actions $[T].a$ with T as above. To obtain several copies of a process P we use process labelling $[label] : P$ which denotes a process that behaves like P with all actions labelled by $[label]$.

The semantics of a process is given by a finite labelled 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 labelled with actions. Since FSP is restricted to a finite number of states one can automatically check safety and progress properties of processes. This will be essential for checking the correctness of our design model for distributed simulations. FSP is equipped with a model checking tool LTSA [7] which will be used for this purpose.

3 Simulation Models

Before we can specify the system requirements we have to analyse the problem domain. Let us first consider single simulation models and provide a formal description of their general behaviour. A simulation model simulates a physical or

social process for a certain period of time which we call simulation time. The simulation time is finite which means that there is always a start and an end time. The whole 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 which determines the distance between two subsequent simulation points. For instance, a meteorological model provides the air temperature every hour, while a groundwater model provides the amount of groundwater withdrawal only once a day. We assume that the time step of a model remains fixed during the whole simulation.

A simulation model provides data for other models via export ports and gets data from other models (needed for its own computations) via import ports.

3.1 Lifecycle of a Simulation Model

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 (via the import ports).
2. Compute new data which are valid at the next simulation point.
3. Provide the newly computed data (via the export ports).

Since any simulation model has the same lifecycle we can model its general behaviour 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 defintion we have to provide a default time step (e.g. Step = 1) which is necessary according to the finite states assumption of FSP. For the same reason it is necessary to model the simulation start and the simulation end by some predefined constants.

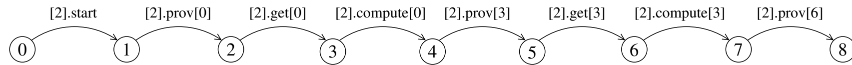
```
const SimStart = 0
const SimEnd = 6
range Time = SimStart..SimEnd

MODEL(Step = 1) = (start -> INIT),
INIT = (prov[SimStart] -> M[SimStart]),
M[t:Time] =
     if (t+Step <= SimEnd)
     then (get[t] -> compute[t] -> prov[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. Indeed the choice of the time dependent indices of the actions is crucial for the behaviour of the whole system to be developed. To explain our choice let

us assume for the moment that the simulation time is a multiple of the model's time step. Then, according to the above process description, the last data that a model gets is valid at time $SimEnd - Step$ and the last data a model provides is valid at time $SimEnd$. For the whole simulation, this means that imported data is considered to be *last recently valid* for the computation of new export values to be valid at time t if the imported data is valid at time $t - Step$.

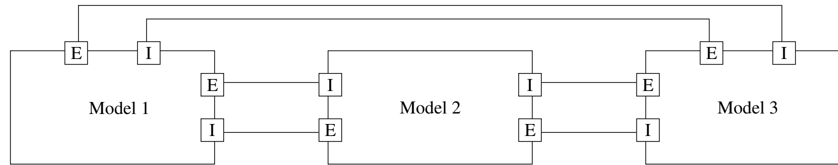
Of course, there are other choices for the definition of last recently valid data. For instance, the intuitively best choice would be to require that the imported values used for the computation of exported values to be valid at time t are also valid at time t (instead of being valid at time $t - Step$). But then the analysis of any attempt to construct a design model for the coordination problem will show that there is no deadlock-free solution (whenever there are, as usual, mutually dependent export and import data). Exactly for this kind of problem analysis, which is not further elaborated here, the use of formal models is indispensable.

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] : MODEL(2) and [2] : MODEL(3) represent two simulation models, one with number 1 and time step 2 and the other one with number 2 and time step 3, resp. The behaviour of model 2 is illustrated by the following LTS.



3.2 Integrative Simulations

In an integrative simulation various simulation models work together by mutually exchanging data via their import and export ports.



Each of the participating models performs a local simulation for the same overall time period (the global simulation time) but has usually a different local time step. It is crucial for integrative simulations that each model gets, whenever needed, the last recently valid data from partner models. A first attempt to

model an integrative simulation could be to simply combine the processes which represent the single simulation models by parallel composition. For instance, for the two simulation models from above we would obtain the following composite process:

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

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

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

1. *Missing import data*

$$\texttt{start} \rightarrow [1].\texttt{prov}[0] \rightarrow [1].\texttt{get}[0] \rightarrow \ldots$$

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

2. *Obsolete import data*

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

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

3. *Overwritten import data*

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

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

4 Formalization of the Coordination Problem

In this section we provide a formalization of the coordination problem in terms of safety and liveness conditions.

4.1 Safety Properties

In Section 3 we have pointed out the essential difficulties concerning the validity of exchanged data when simulation models cooperate concurrently with different time scales. We start by formalizing the corresponding synchronization conditions by means of FSP property processes. The crucial idea is that the problem can be simplified if we consider only two simulation models at a time and, moreover, if we consider each of the two models only under one particular aspect, either as a provider or as a user of information. In the following let U denote a user model and let P denote a provider model. From the user's point of view we obtain the following condition (1), from the provider's point of view we obtain condition (2).

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

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

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

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

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

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

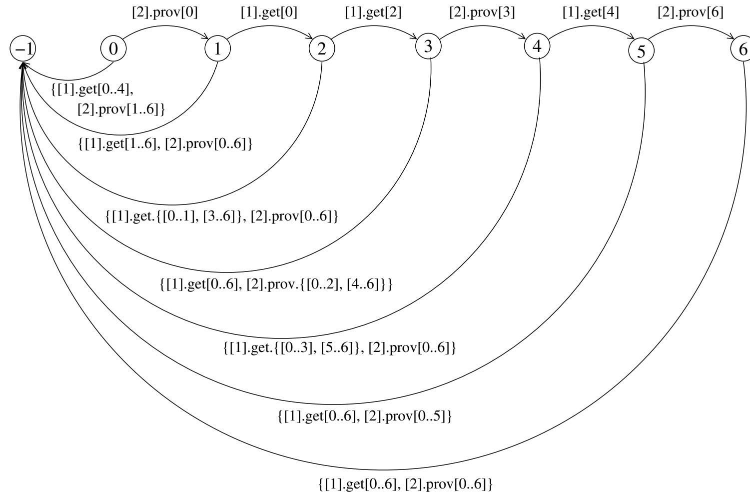


Fig. 1. LTS of the property process VALIDDATA(1,2,2,3)

property process VALIDDATA(1,2,2,3). The labelled transition system of this process is shown in Figure 1.

Labelled transition systems assigned to property processes have an error state, pictorially represented by -1, and are complete in the sense that for any action and any state (apart from the error state) there is always an outgoing transition. This transition leads to the error state if it is not properly defined in the property process definition. Thus the legal and illegal execution traces determined by a property process are revealed. For instance, the three example traces considered in Section 3.2 are illegal w.r.t. the property process VALIDDATA(1,2,2,3), because their restrictions to the alphabet of VALIDDATA(1,2,2,3) lead to the error state.

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

To formalize mutual exclusion we first enclose the critical regions, which in our case are represented by the `get` and `prov` actions, by corresponding `enter` and `exit` actions. For this purpose the process definition for simulation models of Section 3.1 is slightly adapted in the following way.

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

```
        exitProv[t+Step] -> M[t+Step])
  else  STOP + {Labels}.
```

where

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

Note that the alphabet extension by `Labels` is necessary for technical reasons because the alphabet of property processes must be included in the alphabet of processes to be checked. By means of the `enter` and `exit` actions the desired exclusion conditions can now be expressed by a further property process, called EXCLUSION, which follows a standard scheme; cf. [9].

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

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

4.2 Liveness Properties

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

5 Design Model for Integrative Simulations

5.1 Design of the Timecontroller

The specification of the system requirements of the last section is highly non-constructive. In this section we focus on a solution of the coordination problem

which can be easily transformed into an executable program. 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

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

restricts the execution traces of the uncontrolled simulation models to the legal ones. The composite process SYS is then considered as the design model for the system. The (static) structure of SYS is represented by the diagram in Figure 2 which indicates the required communication links.

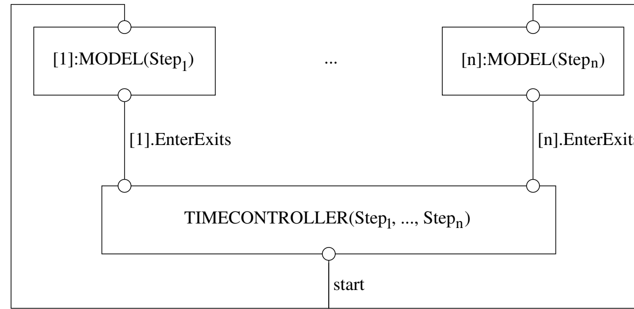


Fig. 2. Structure diagram of the design model

The communication links show that each simulation model m communicates with the timecontroller via the shared `enter` and `exit` actions in the (labelled) set `[m].EnterExits` (see Section 4.1 for the definition of `EnterExits`). This means that the simulation models synchronize with the timecontroller on actions of the form `[m].enterGet[t]` etc., where $m \in$ Models and $t \in$ Time. It is then the task of the timecontroller to guarantee that synchronization can only occur if the constraints determined by *all* property processes (given in Section 4) are satisfied. For this purpose the `enter` actions of the timecontroller are guarded by appropriate conditions which monitor the validity of the safety properties. To express the necessary conditions the timecontroller is equipped with a local state (modelled by index variables) which records the execution status of all simulation models to be coordinated. More precisely, the timecontroller stores for each model the time for which it gets the next import data (represented by the index `nextGet`) and the time for which the model will provide the next export data (represented by the index `nextProv`).

The following time controller definition is formulated for the case of two simulation models where the time steps of the two models are given by parameters.

It is obvious that this description provides a general pattern which can be easily applied to an arbitrary number of simulation models. For a timecontroller definition which is generic w.r.t. the number of simulation models one would need array types which are not available in FSP (but would be available in SPIN [6]). Let us still remark that the guards of the `enter` actions are inferred from the requirements specification by building the conjunction of the guards occuring in the property processes for the validity of data. Moreover, note that model checking shows that the exclusion property for `get` and `prov` is already guaranteed by these conditions and therefore does not need a special treatment.

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

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

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

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

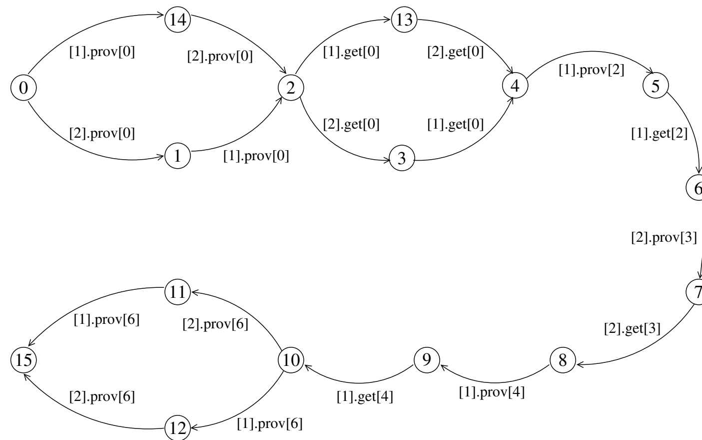
```
                TC[nextGet1][nextProv1][nextGet2][nextProv2]
...
```

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

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

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

We cannot visualize the labelled transition system of the process SYS because it has too many states and transitions. However, for an analysis of the behaviour of the design model we can consider different views on the system which can be formally defined by means of the interface operator. For instance, if we want to focus only on the `get` and `prov` actions executed by the system we can build the process `SYS@{[Models].GetProvs}` where the set `GetProvs` has been defined in Section 4.1. The corresponding LTS, after minimalization w.r.t. invisible actions, is shown in the following diagram.



5.2 Checking the Safety Properties

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

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

The analysis with the LTSA tool shows that no errors occur, i.e. the design model satisfies the coordination requirements for the validity of data and for get/provide exclusion. For more complex configurations more efficient model checkers like SPIN [6] should be used. Several runs with SPIN have shown that the efficiency of model checking the design of the timecontroller depends strongly on the distribution of the individual model steps whereby it is beneficial if their greatest common divisor is as small as possible. Otherwise one may run out of memory and therefore appropriate abstraction techniques have still to be investigated.

5.3 Checking the Liveness Properties

In section 4 we have stated a liveness property which requires that each simulation model provides data during the whole simulation period at any time that fits to its local time step. To check this condition with LTSA we can define a collection of progress properties of the form

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

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

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

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

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

6 Conclusion

We have demonstrated the usefulness of a rigorous formal modelling approach for the development of a solution for a non-trivial coordination problem occurring, for instance, in environmental systems engineering. The general strategy of this approach which is driven by property processes can, however, be applied in all situations where single components run concurrently with local time scales but must cooperate according to some predefined global order. We believe that the incremental specification of system requirements by using property processes is methodologically very useful. This method is supported by the language FSP [9] but not by SPIN [6] or related model checking approaches. On the other hand we have seen that FSP has also some technical deficiencies (concerning array types and guarded indexed actions) which is not the case for SPIN. Also for checking complex models the performance of the SPIN tool is much better than the one of the FSP tool LTSA. To check complex configurations, however, we still need appropriate abstraction techniques to overcome the problem of state explosion.

For lack of space we have not shown in this paper how to construct an implementation of the timecontroller-based design model. Indeed for this purpose we can apply a general translation scheme which transforms the design model into a Java implementation realizing the single simulation models by concurrently executing threads and the timecontroller by a monitor object with appropriate synchronized methods which implement the enter and exit actions of the timecontroller.

Acknowledgement

We are grateful to Alexander Knapp for carefully reading a draft of this paper and for valuable suggestions. Many thanks also to Michael Barth for in-depth discussions on the coordination problem and for a timecontroller implementation and its integration in the DANUBIA system.

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 2005/03/17)
4. GLOWA, http://www.glowa.org (last visited 2005/03/17)
5. Hoare, C. A. R.: Communicating Sequential Processes, Prentice-Hall, 1985.
6. Holzmann, G., The SPIN Model Checker — Primer and Reference Manual, Addison-Wesley, 2004.
7. LTSA, http://www-dse.doc.ic.ac.uk/concurrency/ (last visited 2005/03/17)
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.