

# A Coordination Architecture for Time-Dependent Components\*

Michael Barth, Alexander Knapp

Institut für Informatik, Ludwig-Maximilians-Universität München

Oettingstraße 67, 80538 München, Deutschland

Tel.: +(49) 89 2180 9135/79, Fax: +(49) 89 2180 9175

{barth, knapp}@pst.informatik.uni-muenchen.de

## ABSTRACT

The integration of distributed, data dependent components requires a data synchronisation model. We consider a class of systems where data-dependent components produce data in discrete, fixed time steps but on different local time scales, and which require all components to provide recent data in order to make progress. We introduce an architecture that allows to separate the treatment of the time-based data dependencies among the distributed components from the design of their mutual exchange of data. We prove that our proposal guarantees liveness, safety, and progress of the overall system based on some mild requirements on the single components. We demonstrate the feasibility of our approach by its use in the GLOWA-Danube project.

## KEY WORDS

Modelling and simulation, software architecture, parallel and distributed systems, coordination

## 1 Introduction

Synchronisation of distributed, data-dependent components is a well-known software engineering problem [2, 7]. In general, it has to be guaranteed that no stale data are used by any component and that every component is provided with all needed data from the other components.

We consider the special class of systems of interacting components which produce data in discrete, fixed time steps but on different local time scales, and which require all components to provide recent data in order to make progress. This class of time-dependent component systems arises naturally in distributed physical simulations where different model components use the simulation data provided by other model components in an interdependent fashion but adhere to a rather conventional data import-computation-data export execution cycle. We propose an architecture that uses the local time scales to integrate the timing aspects into an overall time model and employs this model to synchronise all computation steps. In the coordination architecture, each component is embedded into a control panel which is connected to a global time controller. The time controller synchronises the computation steps of

all components whereas the control panel triggers the single computation steps of the embedded component. The control panels require only little change to the computation steps of the components. We present an abstract, formal model of the coordination layer and prove that the timely progress of the distributed system is guaranteed and that all encapsulated components will use recent data.

Our approach to integrating the different component model time scales resembles the use of logical clocks for synchronisation of distributed processes [3, 4]. Logical time is represented by the virtual global simulation time, but the timing scheme is employed simultaneously for guaranteeing mutual exclusion. In fact, the vector of the local model component times can be viewed as a vector of time stamps [6]. However, the vector of time stamps also has to contain information on the model component's computation state in order to establish mutual exclusion.

The coordination architecture has been originally developed and successfully applied for the integration of distributed physical simulation models in the GLOWA-Danube project which integrates some 15 model components for simulating water-related changes in the upper Danube basin. However, the scheme can also be applied to other time-related computations as e.g. in distributed real-time systems, where computation has to be synchronised with a timed base component, or in systems where a set of specific orders has to be joined into a single order and where timed computation is only a secondary effect.

The remainder of this paper is structured as follows: In Sect. 2 we motivate the requirements for the coordination architecture by means of the GLOWA-Danube case study. In Sect. 3, we describe the coordination architecture for time-dependent components. A formal model of the coordination layer and a proof that the coordination layer guarantees correct, safe and live computation is presented in Sect. 4. We apply and refine the coordination architecture for the GLOWA-Danube case study in Sect. 5. Finally, in Sect. 6 we conclude and delineate some loose ends of our approach to coordinating time-dependent components.

## 2 GLOWA-Danube Case Study

The GLOWA-Danube project<sup>1</sup>, started in the end of 2000, aims to model and to simulate the water-related changes

\*This research has been partially supported by the GLOWA-Danube project (07GWK04) sponsored by the German Federal Ministry of Education and Research.

<sup>1</sup><http://www.glowa-danube.de>

in the upper Danube basin. The project set out the task to integrate various existing software simulation systems for the different aspects of the water cycle, like economic, social, meteorological, biological, physical or geological models, into a dependent network of model components called DANUBIA.

Originally, each model covers a specific aspect and the related set of data, whereas data from different aspects have to be retrieved from external databases. DANUBIA provides bridges between the different models and replaces the databases by imports from partner models. All models compute their specific data values based on data they have to import and prepare their results for export to their model partners. Fig. 1 depicts part of the resulting communication structure between the different models. In the inte-

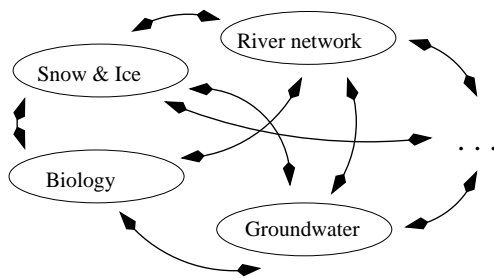


Figure 1. Communication structure between the DANUBIA software models.

grated simulation process, all models compute new simulation data values repeatedly. However, every model computes data on its own local time scale which ranges from five minutes for meteorological models to a month for the economic models. Thus, the data dependencies between the different software models are accompanied by the time dependencies between the different local time scales and the need of every software model to be provided with the most recent data from its own perspective.

In the software engineering process, first all existing models are transformed into model components with import and export contracts, see Fig. 2. The resulting compo-

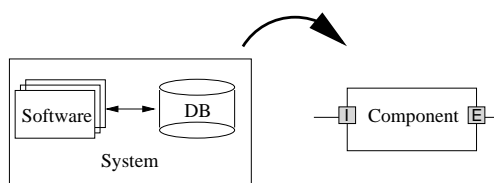


Figure 2. Transforming software to components. Port I is importing data from other models, port E is exporting data.

nents are connected according to the communication structure above as shown in Fig. 3 (cf. [1, 5]).

As it stands, the contracts for import and export ports well reflect the principal data relation between the model components. However, each component has to fulfil the

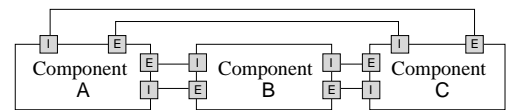


Figure 3. Integrating components. All components are exchanging data with each other.

contract of its ports continuously over time and to deliver valid and stable data whenever these are requested. Consequently, the time dependencies which are incurred by the different time-scales local to the model components are neglected, as a requesting component not only wants to import some data but only the most recent on its own local time scale. Fig. 4 shows an example of the dependencies

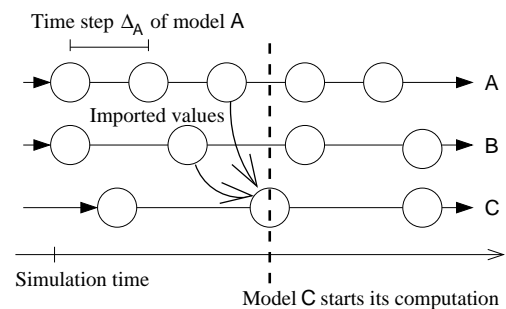


Figure 4. Time dependencies of components. Model C, at the dashed line, is about to compute its data. It has to import required data from A and B in their most recent state.

and the most recent states of communicating models. A more technical data availability hazard results from the fact that preparing data for export will not be instantaneous, in general, and thus a model component can not provide data during the update process.

The state model in Fig. 5 shows the two characteristic critical states of a model component. The first state,

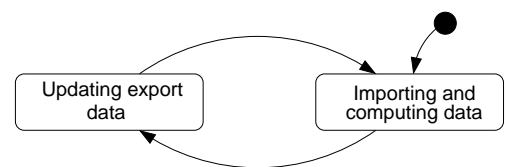


Figure 5. Basic states of a component.

importing and computing data, requires the components to provide data that are stable and most recent. In the second state, when the component finished its computation step, the component's data are not available during the updating process. Thus, at all times all components have to be in the same kind of state. Additionally, the entire set of computation steps has to be put in a single order that respects the local time requirements of the model components. For DANUBIA, this common order can be inter-

preted as a model of virtual time. For the time being, we abstract from the simulation of physical time.

### 3 Coordination Architecture

The time-integration problem influences the components, the control flow of the computation process, as well as the traffic on the import and export interfaces. For local synchronisation, we embed each component into a control panel; each control panel is connected with a centralised time controller component. The resulting coordination architecture, see Fig. 6, integrates the local time scales of the model components into a global virtual time of the time controller. In particular, the global time controller does not modify the components directly, but only uses the additional infrastructure of the local control panels.

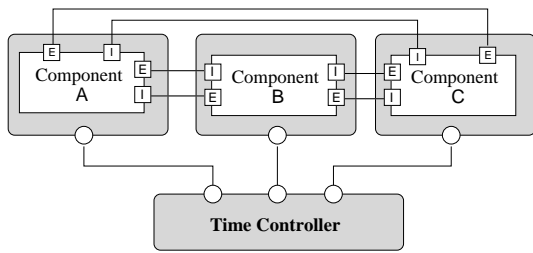


Figure 6. Component coordination architecture. The surrounding control panels and the time controller are shown in grey, the coordination ports between the time controller and the control panels are depicted as circles.

The control panels extend the basic model of computation (cf. Fig. 5) by two additional wait states which represent the points of synchronisation, see Fig. 7. Each com-

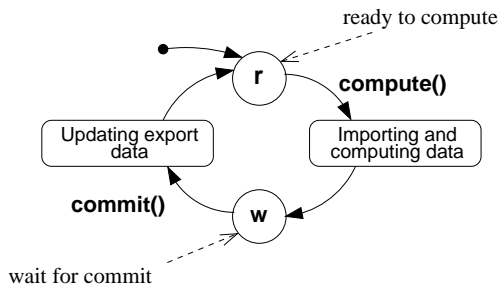


Figure 7. Extended state machine model of a component.

ponent has to remain idle in these wait states until triggered by the global time controller to enter the subsequent critical state. The time controller will send the message `compute` when a model component may start to import data and to compute; it will send the message `commit` when a model component may safely do its update and provide new data to the model component partners. We assume that the importing and updating phases always terminate and we may abstract from the time consumption in these phases for the

time being. It has to be stressed that whenever a model component adheres to this protocol, it can use its import and export ports as if they were time independent.

The coordinating time controller has to ensure that all time-integrated computations of the model components are safe and live: For safety, all computations of a model component have to be executed with valid data from the model component's partners, where the partners' data are valid, if they stem from their most recent computation steps. In the situation of Fig. 8, model component C, which holds at time  $t_C$ , may only use the most recent data of the model components A and B, which thus must not export data for a time point beyond  $t_C$ , i.e.  $t_A \leq t_C$  and  $t_B \leq t_C$ ; and must export data which are still recent enough, i.e.  $t_C - \Delta_A < t_A$  and  $t_C - \Delta_B < t_B$  (where  $\Delta_A$  and  $\Delta_B$  denote the local time scales of A and B). In particular, a model component may

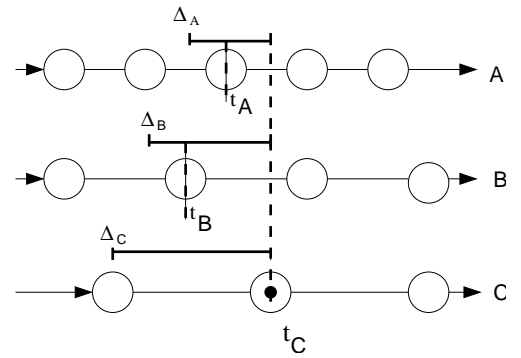


Figure 8. Characteristic model component attributes.

only receive a `compute` message if all the data it needs for this step have already been prepared for import. Furthermore, a model component must not receive a `commit` message prematurely and overwrite data that are still needed by another model component. For liveness, coordinated runs of the model components must not get stuck but reach beyond any prescribed time limit. From a model component's view, liveness of the system in particular means, that its local time will make unbounded progress.

### 4 Formal Coordination Model

The coordination layer has to guarantee safety, liveness, and progress of the overall system. We construct a formal model for the coordinating time controller and subsequently prove that this model shows the desired properties.

From the coordination layer's perspective, a model component is characterised by a *time step*  $\Delta > 0$  and its *state*  $\sigma = \langle s, t \rangle$  with *local state*  $s \in \{r, w\}$  and *local time*  $t \in \mathbb{R}$ . The local state determines whether the model component is ready to compute (local state  $r$ ) or is waiting for commit (local state  $w$ ). The local time represents the time stamp of the data that are currently ready for export by the model component. A *system* consists of a number of model components, which, for simplicity, we assume

to be numbered by  $1, \dots, n$  with  $n > 0$ . The state  $\Sigma$  of such a system thus is given by an  $n$ -tuple  $\sigma_1, \dots, \sigma_n$ , the time steps of the system by  $\Delta_1, \dots, \Delta_n$ . In a system state  $\langle s_1, t_1 \rangle, \dots, \langle s_n, t_n \rangle$ , from model component  $k$ 's view the data of model component  $i$  are valid if they stem from the most recent computation step of  $i$ , i.e.

$$t_k - \Delta_i < t_i \leq t_k.$$

The design of the coordination layer splits the basic data validity requirement for starting a new computation and committing data using the fact that states  $r$  and  $w$  take turns. We only consider the lower bound of the validity requirement for starting a computation; for requesting a commit, we concentrate on the corresponding upper bound.

In a system state  $\langle s_1, t_1 \rangle, \dots, \langle r, t_k \rangle, \dots, \langle s_n, t_n \rangle$ , the model component  $k$  is only allowed to start a computation step, if all its partners provide recent data. Hence, the coordination layer has to ensure that  $k$  only starts its computation if

$$\forall 1 \leq i \leq n. t_k - \Delta_i < t_i. \quad (*)$$

On the other hand, in a system state  $\langle s_1, t_1 \rangle, \dots, \langle w, t_k \rangle, \dots, \langle s_n, t_n \rangle$ , the model component  $k$  is only allowed to commit its computation now being valid for time stamp  $t_k + \Delta_k$ , if the data valid for time stamp  $t_k$  are not needed any more by its partners. Hence, the coordination layer has to ensure that  $k$  only commits its data if

$$\forall 1 \leq i \leq n. (s_i = r \supset t_k + \Delta_k \leq t_i) \wedge (s_i = w \supset t_k + \Delta_k \leq t_i + \Delta_i). \quad (**)$$

Thus, a system evolves by the following two rules:

$$\begin{aligned} &\langle s_1, t_1 \rangle, \dots, \langle r, t_k \rangle, \dots, \langle s_n, t_n \rangle \xrightarrow{\text{compute}_k} \\ &\langle s_1, t_1 \rangle, \dots, \langle w, t_k \rangle, \dots, \langle s_n, t_n \rangle \quad \text{if } (*) \\ &\langle s_1, t_1 \rangle, \dots, \langle w, t_k \rangle, \dots, \langle s_n, t_n \rangle \xrightarrow{\text{commit}_k} \\ &\langle s_1, t_1 \rangle, \dots, \langle r, t_k + \Delta_k \rangle, \dots, \langle s_n, t_n \rangle \quad \text{if } (**). \end{aligned}$$

We write  $\Sigma \rightarrow \Sigma'$  if system state  $\Sigma'$  results from system state  $\Sigma$  by applying one of the rules above. Furthermore, for a system state  $\Sigma = \langle s_1, t_1 \rangle, \dots, \langle s_n, t_n \rangle$ , we use the following abbreviations:

$$\begin{aligned} t + \Delta &\equiv \min\{t_i + \Delta_i \mid 1 \leq i \leq n\}, \\ T &\equiv \max\{t_i \mid 1 \leq i \leq n\}. \end{aligned}$$

We first prove an invariant of the coordination layer that ensures that no model component can be left behind its model component partners and that all model components that are ready to start a computation provide their most recent data. In particular, the invariant establishes the safety property of the coordination layer: Every model component that starts a computation is provided with valid data.

**Lemma (Invariant).** *Let  $I$  denote the following property of a system state  $\Sigma = \langle s_1, t_1 \rangle, \dots, \langle s_n, t_n \rangle$ :*

$$I(\Sigma) \equiv (T \leq t + \Delta) \wedge (\forall 1 \leq i \leq n. s_i = r \supset t_i = T)$$

*If  $I(\Sigma)$  and  $\Sigma \rightarrow \Sigma'$ , then  $I(\Sigma')$ .*

*Proof.* Let the system states  $\Sigma = \langle s_1, t_1 \rangle, \dots, \langle s_n, t_n \rangle$  and  $\Sigma' = \langle s'_1, t'_1 \rangle, \dots, \langle s'_n, t'_n \rangle$  be as in the claim.

Let  $\Sigma'$  result from  $\Sigma$  by an application of the compute rule. Then  $I(\Sigma')$  follows from  $I(\Sigma)$ , as the local times are not changed from  $\Sigma$  to  $\Sigma'$ .

Let  $\Sigma'$  result from  $\Sigma$  by an application of the commit rule to the model component  $k$ . First, observe that then  $t_k + \Delta_k = t + \Delta$  by the application condition of the commit rule: If  $t_k + \Delta_k > t + \Delta$  then there is a model component  $i \neq k$  with  $t_i + \Delta_i = t + \Delta$ . But this model component must be in local state  $r$  as  $t_k + \Delta_k > t_i + \Delta_i$ , and thus  $t_k + \Delta_k \leq t_i < t_i + \Delta_i$ , which contradicts  $t_k + \Delta_k > t_i + \Delta_i$ . Now,  $T' = \max\{t_1, \dots, t_k + \Delta_k, \dots, t_n\}$  and  $t_i \leq T \leq t + \Delta$  for  $i \in \{1, \dots, n\} \setminus \{k\}$  from  $I(\Sigma)$  and  $t_k + \Delta_k = t + \Delta$ . Thus  $T' \leq t + \Delta$ . Furthermore,  $(t + \Delta)' = \min\{t_1 + \Delta_1, \dots, t_k + 2\Delta_k, \dots, t_n + \Delta_n\}$  and  $t + \Delta \leq t_i + \Delta_i$  for  $i \in \{1, \dots, n\} \setminus \{k\}$  and  $t + \Delta \leq t_k + 2\Delta_k$  since  $\Delta_k \geq 0$ . In particular,  $T' \leq t + \Delta \leq (t + \Delta)'$ . It remains to show for all  $1 \leq i \leq n$  with  $s'_i = r$  that  $t'_i = T'$ . We make a case distinction on whether  $T = T'$  or  $T < T'$ : If  $T = T'$ , then the property holds by  $I(\Sigma)$  and  $T' \geq t'_k = t + \Delta \geq T$ . If  $T < T'$  then  $T' = t'_k = t + \Delta$ , since all other local times are left unchanged by the commit rule. For  $i \neq k$ , let  $s_i = s'_i = r$ . Then  $t'_i = t_i \geq t_k + \Delta_k$  by the condition of the commit rule and, by  $t'_i \leq T' = t'_k$ , we must have  $t'_i = t'_k = T'$ .  $\square$

Note that any time in the interval  $[T, t + \Delta]$  can serve as a virtual integrated global time. However, the progress made by the coordination layer will not be continuous, in general.

For the liveness guarantee, we show that in each system state  $\Sigma$  that satisfies the invariant  $I(\Sigma)$  one of the rules is applicable. In particular, if started in a system state satisfying  $I$  the coordination layer will not deadlock. For instance, all  $\Sigma = \langle a, t \rangle, \dots, \langle a, t \rangle$  with  $t \in \mathbb{R}$  satisfy  $I(\Sigma)$ .

**Lemma (Liveness).** *Let the system state  $\Sigma = \langle s_1, t_1 \rangle, \dots, \langle s_n, t_n \rangle$  satisfy  $I(\Sigma)$ . Then there is a  $\Sigma'$  with  $\Sigma \rightarrow \Sigma'$ .*

*Proof.* Assume that for all  $1 \leq k \leq n$  with  $s_k = r$  we have  $t_k \geq t + \Delta$ . If there is no module component with local state  $s_i = w$ , we have  $t + \Delta = \min\{t_1 + \Delta_1, \dots, t_n + \Delta_n\} \geq t + \Delta + \min\{\Delta_1, \dots, \Delta_n\}$ , contradicting  $\Delta_i > 0$  for all  $1 \leq i \leq n$ . Let  $M \subseteq \{1, \dots, n\}$  be the set of model component indices  $i$  with  $s_i = w$  and let  $k \in M$  be such that  $t_k + \Delta_k = \min\{t_i + \Delta_i \mid i \in M\}$ . If  $t_k + \Delta_k > t + \Delta$  then there is an  $i \in \{1, \dots, n\} \setminus M$  with  $t_k + \Delta_k > t_i + \Delta_i = t + \Delta$  and  $s_i = r$ . But this contradicts  $t_i \geq t + \Delta$  and  $\Delta_i > 0$ . Thus  $t_k + \Delta_k = t + \Delta$ . Finally, let  $1 \leq i \leq n$  with  $s_i = r$ . Then  $t_i \geq t + \Delta = t_k + \Delta_k$ .  $\square$

Finally, in order to prove that all runs of the coordination layer make unbounded progress in time, it is enough to ensure that after a finite number of steps some progress in time occurs. The lower bound for this progress in time will be  $\min\{\Delta_1, \dots, \Delta_n\}$  by the commit rule, and thus every given time limit can be transgressed.

**Lemma (Progress).** Let  $\Sigma \rightarrow \Sigma'$  with  $\Sigma = \langle s_1, t_1 \rangle, \dots, \langle s_n, t_n \rangle$  and  $\Sigma' = \langle s'_1, t'_1 \rangle, \dots, \langle s'_n, t'_n \rangle$  and let  $I(\Sigma)$  and  $I(\Sigma')$  hold. If  $t + \Delta = (t + \Delta)'$  then

$$|\{i \mid s_i = r\}| + 2 \cdot |\{i \mid t_i + \Delta_i \neq t + \Delta\}| > \\ |\{i \mid s'_i = r\}| + 2 \cdot |\{i \mid t'_i + \Delta_i \neq t + \Delta\}|$$

*Proof.* If  $\Sigma'$  results from  $\Sigma$  by an application of the computation rule, the number of model components in state  $r$  is reduced by one.

If  $\Sigma'$  results from  $\Sigma$  by an application of the commit rule to model component  $k$  then  $t_k + \Delta_k = t + \Delta$  by the same argument as in the proof of the invariant lemma. Thus the number of model components in state  $r$  is increased by one, but the number of model components not showing  $t + \Delta$  as their local time is reduced by one.  $\square$

As by the invariant no model component can be left behind in time indefinitely, all model components can make unbounded progress.

## 5 Application to GLOWA-Danube

The general DANUBIA architecture is exemplified in Fig. 9 for two model components A and B. Wrapper classes AWrapper and BWrapper are used to make the different implementations of A and B platform independent, the interfaces AToB and BToA realise the components' communication handshake. The class TimeController represents the controller part of the proposed coordination architecture, the TCMoelClient classes implement the control panels.

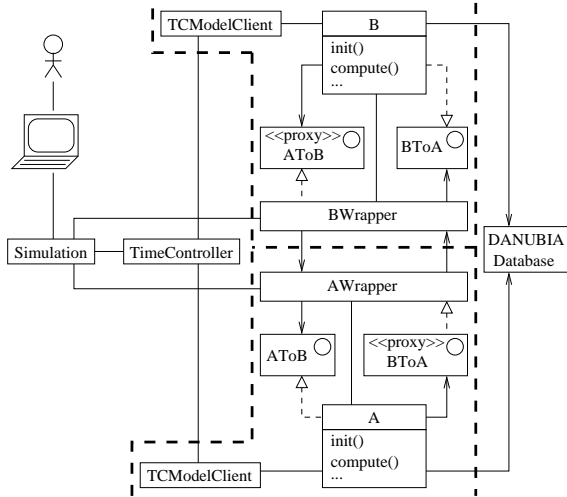


Figure 9. DANUBIA architecture.

The time controller provides the heartbeat of the DANUBIA system and guarantees data consistency over the whole model time period as well as mutual exclusion of reading and writing of data. Figure 10 exemplifies the synchronisation of the computation of the DANUBIA land surface models by the time controller, integrating stand-alone software models into a single control flow.

The LandsurfaceController component bundles the data exchange of several model components, but does not participate in the timing scheme.

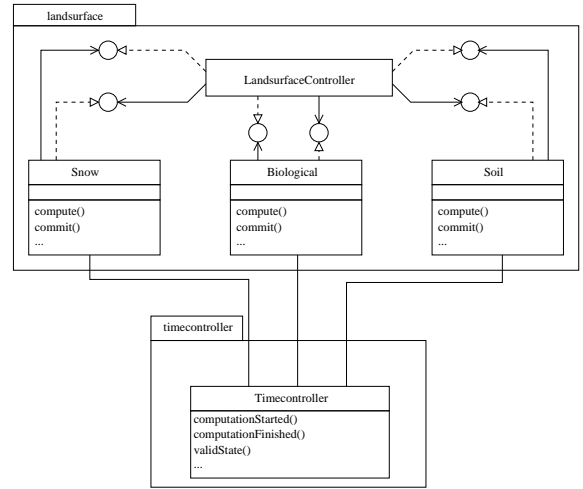


Figure 10. Components coordinated by the time controller.

### 5.1 Control Panel Refinement

In the DANUBIA system, model components spend most of the time on the computation of new data. Also, time for updating data cannot be neglected as these are sometimes up to 2 GB of size. The number of concurrently active components can be increased, if the set of data that can be exported is stored in an export table while computing on an internal copy, such that data can be exported during computation. At the end of each computation step the export table has to be updated with the values of the internal copy.

The actual UML state diagram of all DANUBIA simulation models is shown in Fig. 11. This state machine

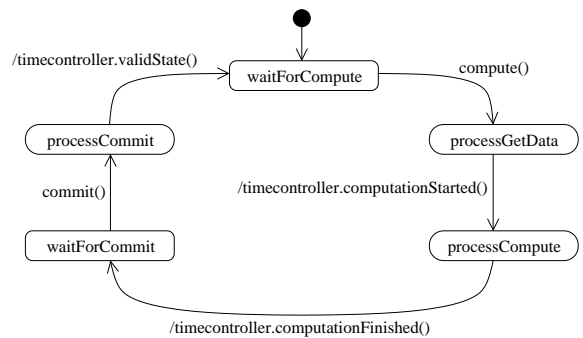


Figure 11. States of a simulation model.

is a refinement of the state machine for model components in Fig. 7 in order to take into account the time consumption of the computation and updating processes. The states processGetData and processCompute refine the state Importing and computing data, state processCommit

replaces Updating export data. The states `waitForCompute` and `waitForCommit` correspond to `r` and `w`, respectively. As computing and updating take time, the time controller is informed of the simulation models' progress by the messages `computationStarted`, `computationFinished`, and `validState`. It is the message `computationStarted` that allows the time controller to take a simulation model actually performing its computation as being in the abstract state `w` and thus to allow other simulation models to commit their data. This refinement is valid as long as the time controller distinguishes between the actual state of the simulation model and the state shown to the other simulation models.

## 5.2 Time Controller Implementation

The development of the time controller resulted in 5600 non-commented lines of Java code. The distribution of the simulation models is realised using Java's Remote Method Invocation (RMI) technology. The overall goal of the design and the implementation of the time controller was to ensure simple integration with the existing simulation models by using a minimal set of interaction points encapsulating all protocols and communication tasks.

The responsibilities of the simulation models and of the time controller are grouped into interfaces. Figure 12 shows a UML class diagram for the `TCModelAccess` interface of the simulation model and for the corresponding `TCConfirmation` interface of the time controller. The

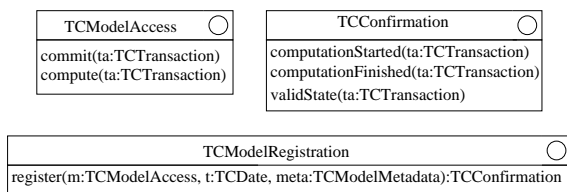


Figure 12. Interfaces describing the protocol.

messages at the transitions of the component model's state diagram in Fig. 11 correspond to the operation names. The time controller also implements the interface `TCModelRegistration` to allow initial registration of the simulation models implementing the `TCModelAccess` interface. The additional parameters contain information about the model time and its local time step. The states of the simulation model shown in Fig. 11 have been directly imported into the implementation model of the time controller using the State Design Pattern.

The time controller implementation has been integrated into the DANUBIA system running on a 40 CPU Linux cluster and shows encouraging performance.

## 6 Conclusions

We introduced an architecture for the integration of time-dependent distributed components. This coordination ar-

chitecture guarantees safety, liveness, and progress of the integrated system. Components only have to comply to a few mild requirements to ensure safe cooperation. Adapting already existing components to the coordination architecture requires only a few schematic changes. Finally, coordination does only incur some simple conceptual restriction on the design of new components.

We applied our approach to the GLOWA-Danube case study demonstrating its benefits for a large-scale component network of integrated physical simulations. The experiences in GLOWA-Danube and the DANUBIA development showed that time-related aspects and other structural concerns can be considered separately and that the analysis complexity can thus be reduced. The application of this architecture to the entire DANUBIA system lead to a clearly structured, easily extensible, and reliable design.

The mutual exclusion of the updating and importing states between two components is necessary only, if these models mutually exchange data. Our architecture guarantees exclusion of data availability hazards, but assumes that each component may exchange data with each other. A formal model that considers only existing connections may also prove sufficient safety and liveness properties, but we defer this to future work.

## References

- [1] M. Barth, R. Hennicker, A. Kraus, and M. Ludwig. An Integrated Simulation System for Global Change Research in the Upper Danube Basin. In *1<sup>st</sup> World Congr. Information Technology in Environmental Engineering (ITEE 2003)*, Gdansk. ICSC-NAISO Academic Press.
- [2] D. F. D'Souza and A. C. Wills. *Object, Components, Frameworks with UML: The Catalysis Approach*. Addison-Wesley, Reading, Mass., &c., 1998.
- [3] L. Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Comm. ACM*, 21(7):558–565, 1978.
- [4] L. Lamport. Concurrent Reading and Writing of Clocks. *ACM Trans. Comp. Sys.*, 8(4):305–310, 1990.
- [5] R. Ludwig, W. Mauser, S. Niemeyer, A. Colgan, R. Stolz, H. Escher-Vetter, M. Kuhn, M. Reichstein, J. Tenhunen, A. Kraus, M. Ludwig, M. Barth, and R. Hennicker. Web-based Modeling of Water, Energy and Matter Fluxes to Support Decision Making in Mesoscale Catchments — The Integrative Perspective of GLOWA-Danube. *Physics and Chemistry of the Earth*, 2003. To appear.
- [6] M. Raynal and M. Singhal. Logical Time: Capturing Causality in Distributed Systems. *IEEE Computer*, 29(2):49–56, 1996.
- [7] A. S. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, Upper Saddle River, New Jersey, 2002.