JOURNAL OF OBJECT TECHNOLOGY
Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2008

Vol. 9, No. 1, January–February 2010

An Aspect-Oriented Approach for the Development of Complex Simulation Software

Tudor B. Ionescu, Andreas Piater, Walter Scheuermann, and Eckart Laurien, University of Stuttgart, Institute of Nuclear Technology and Energy Systems, Stuttgart, Germany, Email: {ionescu,piater,scheuermann,laurien}@ike.uni-stuttgart.de

We propose an aspect-oriented approach for the development of simulation software aiming at increasing the flexibility, the rapidity of development, and maintainability of simulation software. The horizontal decomposition method is used to separate the core functionality of the simulation application from simulation-specific cross-cutting concerns like distribution, tool integration, persistence, and fault tolerance. We analyze an existing dispersion simulation application to demonstrate the applicability of our approach and provide a proof of concept in form of the aspect-oriented implementation of two cross-cutting concerns, namely distribution and tool integration.

1 INTRODUCTION

In many engineering and natural sciences modeling and simulation play an important role in understanding the behavior and limitations of certain technical facilities, natural phenomena, and other physical or abstract systems. Simulation software has been developed from the very beginnings of the computer science era and one type of software component, called *simulation code*, has been established as the standard way of encapsulating a simulator for a certain physical aspect of a real system. A simulation code is a command line executable (often written in FORTRAN) which uses file-based communication with the outside world. Some of the codes also support command line parameters.

Codes are usually developed by research institutes and some of them can date back from the late 1970s. Therefore they pose legacy problems when it comes to integrating them into modern applications for educational, research, and commercial purposes. The most common way of reusing the codes is to wrap them into a class using a modern programming language. Despite the legacy problems they pose, the codes are still in use and entirely rewriting them using a modern programming language is not an option. On one hand, time and financial resources are limited whereas, on the other hand, the knowledge comprised in these codes is largely gone due to the high fluctuations in personnel (i.e., developer churn) at universities and research institutes.

In order to cope with the legacy problems, the financial restrictions, and the com-

Cite this document as follows: : *An Aspect-Oriented Approach for the Development of Complex Simulation Software*, in Journal of Object Technology, vol. 9, no. 1, January–February 2010, pages 161–181,
http://www.jot.fm/issues/issue_2010_01/article4/

plex requirements of simulation software for end users a new software development paradigm is needed. This paradigm must facilitate the encapsulation and tool integration of the legacy simulation codes as well as the rapid and easy implementation of all concerns of end users simulation software. Furthermore, it must provide the implementation with the necessary flexibility to adapt the simulation application to different usage contexts. Last but not least, a high maintainability of the software is desired in order to extend the lifespan of products of research institutes.

In this paper, we present an aspect-oriented [1] approach for the development of simulation software. We aim at increasing the flexibility and maintainability of simulation software while reducing the development time and extending the product's lifespan. The core functionality of the simulation application must be first identified and then separated from the cross cutting concerns that are specific for end user simulation software. Aspect-oriented programming is used to implement cross-cutting concerns like distribution, tool integration, persistence, and fault tolerance whereas standard object-oriented programming is used for implementing the core functionality of the simulation application. We provide a proof of concept for this approach by discussing the implementation of two concerns, namely distribution and tool integration. By employing actor-oriented modeling [2] for the design of the workflows, the horizontal decomposition (HD) method [3] for systemic decomposition, declarative programming with Java Annotations, and AspectJ [4] for the aspect-oriented implementation we claim that our approach leads to less boilerplate code[1], and increased modularity, flexibility, and maintainability of the system.

2 ABR: A DISASTER PREVENTION SYSTEM

In this study we focus on software for simulating the dispersion of radioactive pollutants. Dispersion modeling is a discipline that provides the mathematical models to calculate the concentration of a substance present in the atmosphere that was released by some source of pollution in any point of an area surrounding the point of emission. In case of radioactive pollutants the source might be the reactor of a nuclear power plant, a dirty bomb, or some other radioactive material. After being released the pollutants suffer an airborne transport from the source in all three spatial dimensions. The dispersion in the horizontal plane is caused by wind whereas atmospheric turbulence is the main cause for vertical dispersion. Further meteorological factors influencing the transport and deposition of the pollutants are air temperature and precipitation.

Our investigation started from a matured dispersion simulation system, called ABR, which serves as an early-warning system [5] for emergency situations at nuclear power plants. The dispersion calculation is performed by a number of FORTRAN and C/C++ codes linked together into different workflows, depending on the type of

[1]Boilerplate is the term used to describe sections of code that have to be included in many places with little or no alteration.

calculation that is required by an emergency situation. The workflows are controlled by a workflow service and the codes are wrapped to act as Windows services. All Windows services communicate with each other and can be hosted on different machines. The communication between the calculation services is done via a Service Access Layer (SAL) [6] which is based on CORBA [7]. The communication with remote clients takes place over a SOAP Webservice [8]. The GUI client is developed by a third party company.

In this design each service can only be hosted on a single machine and no other means of distribution has been foreseen in case that more than just a few users would start a calculation at the same time. The mixture of different technologies (e.g., SAL, CORBA, and SOAP Webservices) requires skilled programmers to maintain and extend the system. The use of "heavy-weight" middleware technologies and Windows services introduces boilerplate code because of the need to wrap every module into a Windows service and then develop some protocol to allow the modules to communicate with each other (i.e., the SAL). Hence the principle of modular decoupling of software components is violated because of the need to modify several components if a change in one component is required. Finally, the mash-like interconnection of the calculation services does not allow for any other way of extending the core functionality of the system than to implement a new Windows service and link it to the existing ones using the same technologies.

Figure 1 shows the internal workflow used to perform dispersion calculations in the ABR system which follows the latest VDI 3945 guideline [9] for atmospheric dispersion models. The internal workflow is executed once for each time step. The typical length of a time step is 10 minutes. At each new step the updated weather data (i.e. wind and precipitation conditions) are fetched from the database of the National Weather Forecast Center for the area surrounding the point of emission within a radius of 25 to 75 km. Besides the weather data the system requires an input of emission data. Emission data represent information about the quantity and nature of the released radioactive pollutants. This information is provided as one of 20 possible incident or accident categories which range from an incident without radioactive emission (cat. 20) to a catastrophic reactor core meltdown (cat. 1).

In case of a real emergency a dispersion calculation with a duration of 48 hours is performed in real time using measured weather and emission data. The system provides results which are plotted onto a digitized geographical map every 10 minutes. The functions of the nine C++/FORTRAN calculation codes presented in figure 1 are as follows: *CRE_TOPO* – generates topographical data on the basis of a homogeneous land surface model; *WINDO* – using the wind forecast data this module computes a 3-dimensional Cartesian wind field through interpolation; *KART_GELF* – converts the Cartesian wind field into one that takes into account the vertical dimension of land surface; *FLAECH_INT* – using precipitation forecast data this module computes the distribution of the intensity of precipitation for a given area; *INVENTAR* – computes the nuclide inventory of a nuclear reactor; *FREI_MOD* – computes the nuclide release from a reactor; *PAS2* – implements a Lagrange dis-

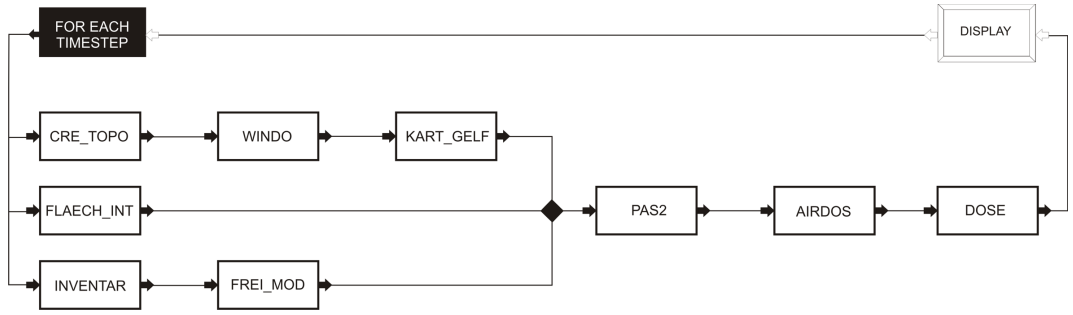


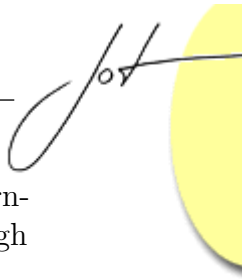
Figure 1: The workflow used to perform dispersion calculations in the ABR system.

persion model which uses the input wind field and the distribution of precipitation for the dispersion calculation; *AIRDOS* – computes the equivalent dose of different trace species; *DOSE* – computes the effective dose of radioactivity based on the equivalent dose for different age groups.

Design Goals for the New ABR System

Our experience from the past revealed that most of the problems and bugs that appear after the development of a simulation application are due to the faulty implementation of cross-cutting concerns like distribution, access control, or persistence rather than from the core functionality of the system. For this reason, the design of the new system is based on the *less is more* principle in the sense that now a method of complete separation of the cross-cutting concerns from the core functionality is sought. This way the focus is shifted toward the development and maintainance of the application core rather than of cross-cutting concerns. The core functionality represents *less code* but *more added value* since it is the actual product of the research institute. The core functionality of a simulation application is to be developed by domain experts of research institutes with the assistance of professional programmers. The implementation of cross-cutting concerns can be outsourced (i.e., developed by another department, institute, or software company). With this in mind, the new system achieves the following design goals:

- There is a clear logical and semantical separation of the distribution, access control, fault tolerance, and persistence cross-cutting concerns from the core functionality;
- The behavior of cross-cutting concerns is specified in the application code of the core functionality components through declarative programming;
- The implementation of new features is easy and affects a minimal number of system components;
- The elimination of obsolete features is equally easy to accomplish;



- There exists a mechanism for activating and deactivating the extended concern-specific functionality in order to achieve a high degree of flexibility through configuration files as well as better modular testing capabilities;

- The new design is based on a layered software architecture where communication can only take place between neighboring layers;

- The implementation of concerns like distribution, persistence, etc. is technology independent and is carried out after the implementation of a solid application core;

- The complete application copes with various usage contexts (e.g. research, commercial, educational, etc.) and different deployment configurations ranging from a cluster deployment to a notebook installation.

3 HORIZONTAL DECOMPOSITION OF THE ABR SYSTEM

Horizontal decomposition (HD) [3] is a method of systemic decomposition by which a software system is orthogonally divided into a vertical core architecture and a horizontal axis of functional extensions. The core architecture accomplishes the basic task of the software whereas the horizontal extensions provide additional functionality to the system, like distribution, security, or persistence. The cross-cutting of the two axes is realized in such a way that the core architecture be unaware of the horizontal extensions which are built around it in order to adapt the functionality of the system to different application scenarios. The aim of HD is to eliminate the implementation convolution problem. Implementation convolution refers to the fact that although the semantics of different components of a software system are distinctive, their implementations do not have clear modular boundaries within the code space but are rather tangled and inseparable.

HD relies on the aspect-oriented programming paradigm [1] and the method has been shown to be effective in refactoring existing middleware software leading to a 40% reduction in code size and a significant improvement in performance [3]. Aspect-oriented programming (AOP) aims at increasing modularity by sepparating cross-cutting concerns. A cross-cutting concern is a particular program functionality the implementation of which is spread over many components of the application. The term *aspect* designates a class that can alter the behavior of other (non-aspect) classes by applying so called *advices* at different *join-points* between these two types of classes. Advices are additional program code that can be executed *before*, *after*, or *around* (instead of) a certain method or some other code unit of a program. The additional behavior implemented through aspects is *woven* into the bytecode of the application at compile-time or run-time.

HD consists of five principles (see [3]) three of which will be considered for the current approach.

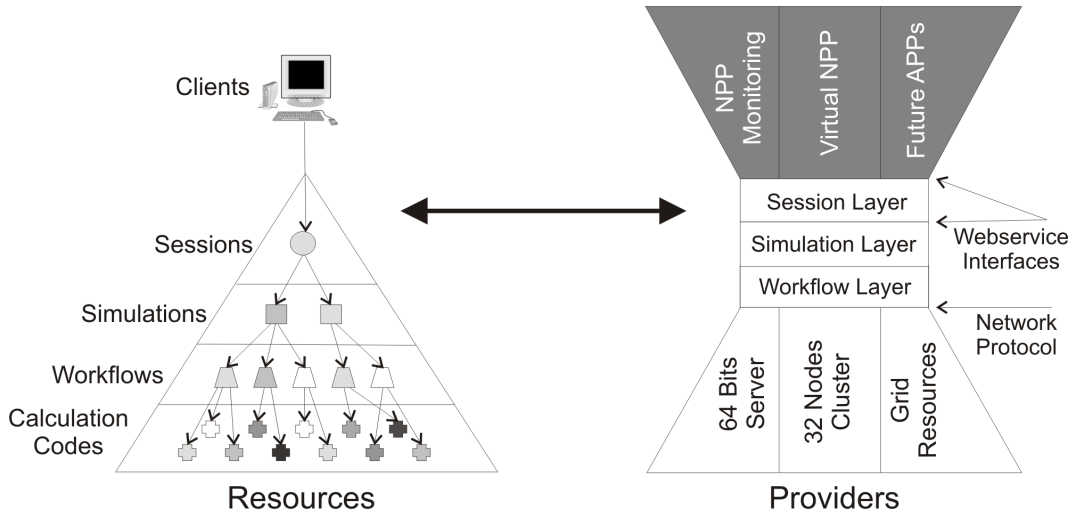


Figure 2: The layered hourglass architecture of the ABR system. NPP stands for nuclear powerplant.

First Principle *A coherent core decomposition of a system must be established.*

Finding the core of an application means identifying the components with functionality that contributes to directly performing the main task of the system. The main task of our examined system is to perform a dispersion calculation regardless of the policies used for access control, user management, and persistence. The core functionality is represented by the codes, the workflows, and the simulation scenarios needed for performing a dispersion calculation.

Second Principle *The semantics of an aspect should be defined according to the core decomposition; if the semantics and the implementation of a functionality are not local to a single component of the core then it is considered to be orthogonal to the core.*

If a functionality is orthogonal to the core's axis, as defined above, it is considered to be a cross-cutting concern to the application and its implementation represents an aspect. For example, distribution is not local to one component of the core since simulation objects can be distributed as well as calculation codes.

Third Principle *A class-directional architecture where aspects are "aware" of functional modules but modules are not "aware" of aspects must be maintained; cross-cutting concerns should be implemented class-directional towards the core.*

For example, the developer of a workflow should not have to think whether or not the workflow will be executed locally or remotely. Therefore the implementation of workflows will only focus on the core functionality whereas aspects implementing other concerns will focus on how to make the core act in different application scenarios where these concerns will play a role. A class-directional architecture assures a clean separation of concerns within the application.

The Core Architecture of the ABR System

Without risking any loss of generality, we can claim that a typical simulation application can be hierarchically decomposed into *4 layers* presented in figure 2 (left side). In this *resource-oriented model*[11], each of the 4 layers is the host of a certain type of *resource*. From the bottom-up we have the fundamental resources, called calculation services, representing software implementations of different mathematical algorithms (the ABR codes). These modules are interconnected to form workflows which, in term, are part of simulations. Finally the user represents the ultimate human resource who uses remote clients to perform simulation sessions.

The right side of figure 2 shows the hourglass physical architecture corresponding to the layered logical architecture. Each layer corresponds to a resource provider and can run on a separate machine. Objects that are instantiated at different levels in the stack represent the actual resources of that layer. The waist of the hourglass is composed of three layers that are common for all deployment and/or application scenarios. The fat top of the hourglass is represented by the different remote clients which are using the simulation framework. The fat bottom is represented by the most common types of computational resources currently available. In the current approach the communication between layers that are not hosted on the same machine is achieved through technology independent Webservices [8].

4 THE ASPECT-ORIENTED SIMULATION FRAMEWORK

We propose a framework of aspects called AoSiF (Aspect-oriented Simulation Framework) for extending the core functionality of a simulation application in such a way that no change to the code of the core be needed. The extensions consist of aspects that implement cross-cutting concerns like distribution, access control, fault tolerance, persistence, workflow engine integration, etc. The basic structure that is applicable to any of these concern is shown in figure 3. The building blocks of this implementation schema are Java annotations and AspectJ aspects although similar features of other programming languages could be used (e.g., C# attributes are equivalent to Java annotations).

From a mathematical point of view a computer program P can be expressed using a multivariate function $P = p(x_1, x_2, ..., x_n)$. Through functional decomposition we can identify a set of functions $f_1 ... f_m$ so that

$$P = \chi(f_1(x_1, ..., x_n), f_2(x_1, ..., x_n), ..., f_m(x_1, ..., x_n)) \quad (1)$$

where χ is some other function. The procedure can continue recursively with the decomposition of $f_1 ... f_m$ producing a hierarchy of dependencies between these functions that can be modeled according to a specific domain.

Now, supposing that the functions $f_1 ... f_m$ represent the core functionality of a simulation application and that we want to extend or modify the behavior of a par-

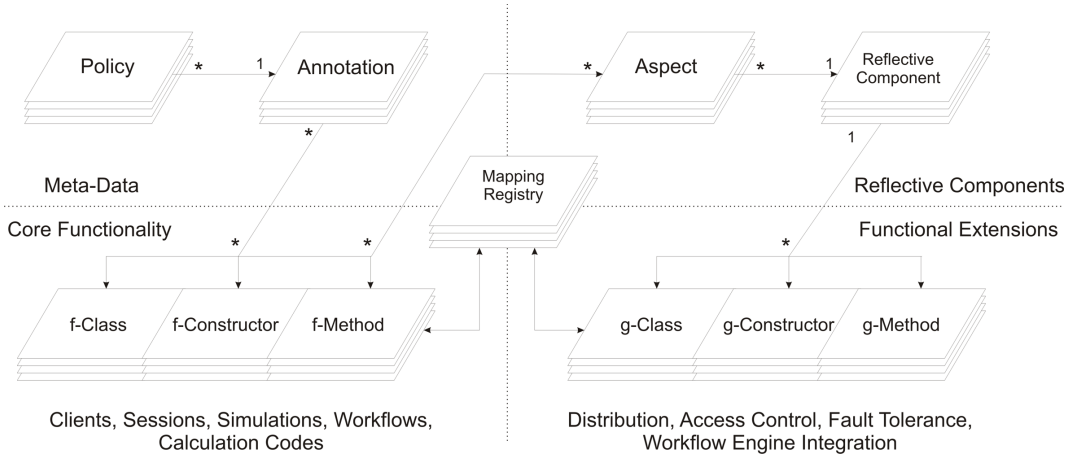


Figure 3: A generic model for horizontal decomposition using Java annotations, AspectJ, and reflection.

ticular function f_i we could express this analytically through functional composition as follows:

$$g_2 \circ f_i \circ g_1 = g_2(f_i(g_1)) \tag{2}$$

where g_1 and g_2 represent extended functionality. This extended functionality can be implemented using AOP, that is g_2 and g_1 represent *after* and *before* advices, respectively. This means that f_i waits for the execution of g_1 which can alter some of the parameters used to call f_i whereas g_2 is executed after f_i and may use the return value(s) of f_i. Thus, additional functionality is woven into the program and P becomes

$$P = \chi(f_1(x_1, ..., x_n), ..., G, ..., f_m(x_1, ..., x_n)) \tag{3}$$

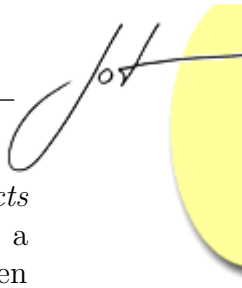
where

$$G = g_2(f_i(g_1(x_1), ..., g_1(x_n))) \tag{4}$$

Furthermore if f_i is a function that at some point needs to be replaced or removed we could simply create an *around* advice meaning that f_i is replaced by a function g_3 with the same input parameters as f_i.

Figure 3 shows how this can be realized in practice by using standard Java and AspectJ technology. In the lower part of the left hand side the core functionality is represented by so called *f-constructs* which are standard language constructs like classes, constructors or methods. The *f-constructs* are "decorated" using concern-specific annotations, e.g. `@Distributed` for a class of distributed objects. Each concern-specific annotation can contain different policies that dictate the new behavior assigned to an *f-construct* through that annotation. Annotations and policies actually expose additional information (meta-data) about a new behavior for the annotated *f-construct* in a declarative manner.

On the right hand side in the upper part of the picture there are aspects implementing different concerns that are activated by the presence of certain annotations over *f-constructs*. Depending on the type of the annotation and its applied policies,



a reflective component will instantiate so called *g-objects* containing *g-constructs* that extend the core functionality of the application. Usually for every *f-object* a new *g-object* supporting the extended functionality is needed. The mapping between *f-objects* and *g-objects* is kept in a mapping registry. This mapping is required in order to avoid duplication of *g-objects* for one *f-object* and for call-backs to the *f-object* from a *g-object*. An *f-construct* can be decorated by one or several annotations depending on the need of functional extension of the application.

Since annotations can be ignored by the Java compiler through a compiler switch the application can act the same as before even after being annotated. One direct advantage of this approach is that the core functionality can be tested as a module prior to the tests with the extended functionality. Furthermore, each extension can be switched off to allow the testing of other extensions together with the functional core.

Tool Integration

Tool integration might not seem to be an obvious cross-cutting concern. However, we will show in this section that tool integration can be regarded as a cross-cutting concern since code that makes the link between workflow components and the workflow engine is required and spread over many application components.

The codes used by the ABR system are command line executables which communicate with the outside world only through input and output ASCII files. In the beginning the codes were linked together using batch scripting or custom applications concerned with creating a basic workflow execution environment [12]. This approach proved to be wrong because it was difficult to handle exceptions, parallelism, and synchronization as well as to manage the unstructured code itself. In their basic form the ABR codes must be regarded as a heterogeneous system of interconnected modules with no consistent data flow model. The entire workflow, however, must act as a homogeneous model of computation.

The Ptolemy II (PtII) [13] scientific workflow engine supports actor-oriented hierarchical modeling of heterogeneous systems by focusing on the data flow, the synchronization of the execution, and the visual design of workflows using the Vergil GUI. The basic building block of a PtII workflow is the actor. The actor is a Java class which, in our case, wraps the FORTRAN and C/C++ codes. The actor class is restricted to have a specific structure: it has parameters, I/O ports, and action methods. Parameters are set by users; ports are used to interconnect actors, for data flow, and for token based flow control; action methods are invoked by the PtII workflow manager at different stages of execution, e.g. initialization, fire, wrapup, etc. When the *workflow manager* fires an actor, the latter one consumes the tokens on its input ports, performs its job, and produces tokens on its output ports. A *workflow director* dictates the type of interaction and the flow control rules for a particular type of workflow. The most common type of director is the synchronous data flow (SDF) director which uses token based flow control. The SDF director is

also suitable for the ABR workflow.

The ABR codes are encapsulated into PtII actors and linked together into workflows which are encoded and stored as XML files. For each code the following actors are needed:

- *input preparation actor* - collects or generates input files for the code;
- *process launcher actor* - launches a local or grid process using the executable code;
- *report actor* - parses the output files and stores relevant results produced by the code into a central database.

These three actors are encapsulated into a *composite actor* that is specific to each module. A composite actor is actually a subworkflow that has no specified director. The input preparation and process launcher actors can also be composed of several actors and therefore can be composite actors themselves. Composite actors provide the workflow designer with the possibility of having a clear hierarchy of components. It is also possible to save simple and composite actors in the user library for later use.

Now, there is a drawback to PtII: because actors are Java classes with a strict specific structure the developer either implements all components of the system as PtII actors or a PtII actor wrapper class is needed for each component that has to be integrated into the workflow. The first solution is unacceptable since components must be reusable in other applications whereas the second solution generates more boilerplate code. Furthermore, the implementation of PtII actors is not trivial in terms of required programming skills. At this point it becomes more evident that tool integration is actually a cross cutting concern.

Fortunately, by using Java annotations and AOP it is possible to avoid writing a new actor wrapper class for each component of the workflow without changing the code of the original component class. A simple PtII actor class contains three main sections:

- *declarations* - actor ports, parameters, and other resources are declared here;
- *constructor* - actor ports and parameters are initialized and type constraints are applied;
- *action methods* - these methods are invoked by the workflow manager in a certain order and perform the actual job of an actor.

Our goal is to eliminate the need of implementing a new actor wrapper class for each component of the workflow. Given a component class of a simulation system that is part of a workflow, the proposed solution is to decorate the original methods of this class with annotations corresponding to key elements of PtII actors. This

```
public class ModuleActor implements Serializable{
    String paramName = null;
    Object simulation = null;

    @ActorConstructor
    public ModuleActor(){}

    @ActorParameter(name="nameParameter")
    public void setParamName(String paramName){this.paramName = paramName;}

    @ActorAction(actionMethods={ActionType.Fire},callingOrder=0)
    @ActorPort(name="outputToken",direction=PortDirection.Output)
    public Object start(
            @ActorPort(name="inputToken",direction=PortDirection.Input)
            Object inputToken
    )throws Exception{
        // doing some work here ...
        Simulation sim = (Simulation)simulation;
        if(sim != null)
            sim.receiveActorEvent(this, ActorEventType.Update);
        return inputToken;
    }

    public Object getResults(){ // return some results set }

    public void setSimulation(Object simulation){this.simulation = simulation;}
}
```

Figure 4: Annotated component class.

way the component class will act as a PtII actor class with no need to change its original code. Figure 4 presents such an annotated component class. The following four annotations were needed to transform this class into a PtII actor:

- `@ActorPort` - over a method states that the return value of the method is to be encapsulated into a token and put on an output port; over an input argument of a method states that the value of an argument is received on an input port of the actor;
- `@ActorParameter` - over a setter method indicates that the value which is set represents a user definable PtII actor parameter;
- `@ActorAction` - over a method states that the method is to be called when the action method of the PtII actor class specified by the `actionMethods` annotation parameter is invoked by the workflow manager;
- `@ActorConstructor` - over a constructor indicates the constructor that has to be indirectly invoked by the PtII workflow manager in order to instantiate an object of the component class (explanation follows).

The principle used for integrating such a class into a PtII workflow is based on the generalized model presented in figure 3. For all of the annotated component classes we need one generic PtII actor class, named `ActorBase`, to provide the link to the PtII workflow. The `ActorBase` class extends the PtII `TypedAtomicActor` class and therefore exposes all the action methods of a PtII actor. The `ActorBase`

class is invisible to the developer and contains arrays of ports and parameters that have to be initialized by an `ActorAspect`.

Now, when an `ActorBase` actor is instantiated from the user library in Ptolemy II[2] its name has to be changed to a fully qualified class name. This name string is actually used to initialize an object of the specified component class in the constructor of `ActorBase` which is invoked by the workflow manager at initialization time. The link between the `ActorBase` object and the object of the annotated component class is then realized through reflection. The proper constructor from the component class is identified by its `@ActorConstructor` annotation. At this point the `ActorAspect` reacts on the basis of the invocation of a constructor annotated by `@ActorConstructor`. Using reflection it looks for the methods and arguments of the actor class that are annotated by `@ActorPort` and `@ActorParameter` and initializes the port and parameter arrays of the `ActorBase` object. It also identifies and adds references to the methods of the component class marked with the `@ActionMethod` annotation into an array of the `ActorBase` object. Finally, the `ActorAspect` applies an advice every time an action method of the `ActorBase` class is invoked by the workflow manager. Using the methods array of the `ActorBase` object the method(s) of the component class marked with the `@ActorAction` annotation which match the type of action method invoked by the workflow manager are identified and executed using reflection. The `ActorAspect` also manages the consumption and production of tokens on the I/O ports defined using the `@ActorPort` annotation.

The `setSimulation` method is used to provide the actor with a mean of communicating with the simulation object. A reference to the simulation object is passed from the workflow manager object to the actor by the `ActorAspect` at initialization time. The reference must be passed by the simulation object itself to the workflow manager when the workflow is started. Therefore a customized version of the workflow manager containing a `setSimulation` method is needed. The communication between the actor and the simulation actually represents the link between the simulation and the workflow layers of the core architecture and will be explained in more details in the next section.

Distributed Resources

There are two main reasons why the distribution concern plays a role in the development of applications. The first reason is related to the need of distributed processing which, unlike parallelization, applies when a single processor can deliver the result of a computation in due time but there are too many computations that need to be performed at the same time, possibly is a multi-user scenario. The second main reason for distribution is motivated by the need of external software tools or hardware resources that are not available on the same machine from licensing, compatibility, or performance reasons. Thus different components of the application need to run

[2]In PtII's GUI users can drag and drop custom actors from a menu called *User Library*. `ActorBase` is such a custom actor.

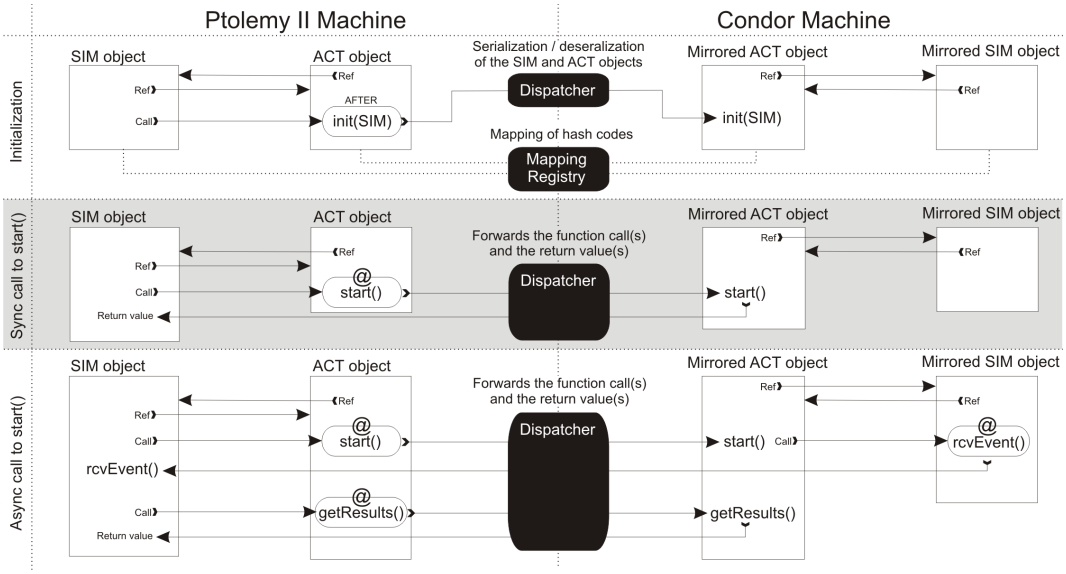


Figure 5: AoSiF's distribution mechanism.

on different machines and communicate with each other in order to provide the user with a final result. AoSiF addresses these two aspects of distributed computing.

A central point of the distributed objects (DO) model [14] is the fact that, after the relocation of objects using some middleware technology, the programmer will have no knowledge of whether an object is local or not [15]. In middleware platforms like CORBA this is achieved through automated generation of stubs for each distributed class. Instead of invoking methods of the actual object one invokes the stubs which are always local. The middleware then locates the actual object on the network and forwards the call to the proper machine. Web services technology is based on the same principle: stubs are generated for local invocation whereas the actual objects (Webservices) are hosted on a remote web server.

The goal of our approach is to eliminate the need of stubs for distributed objects and inline calls to middleware API from the core application code. By using Java annotations and AOP we can automate the actual distribution of the objects. One good reason for this is the fact that although stubs are automatically generated by IDEs or other middleware specific tools they still represent boilerplate code and sometimes can even contain erroneously generated code[3].

The Working Scenario. In the previous section we discussed the concern of tool integration of existing component classes. As part of workflows these components become actors which communicate with other actors through ports. This communication takes place within the lowest level of the layered architecture presented in figure 2. The modules that are wrapped as PtII actors also need to communicate with the simulation objects from which the PtII workflow has been instantiated.

[3]In Netbeans 6.0 Webservice stubs are sometimes generated erroneously due to some faulty argument type identification mechanism.

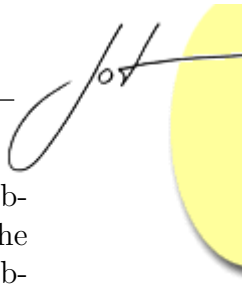
This way actors can provide the simulation object with information regarding the progress in the execution of the workflow and the results of the computation. The simulation, in term, sends this information to its upper layer and so on. With this in mind, we can imagine the following scenario:

- Actors must be fired on a machine running the Condor workload management system [16] for computer grids;
- Workflows run on a PtII machine together with the simulation objects;
- The upper layers of the stack are hosted on a separate web server;
- Actors must report their progress to the upper layer.

Starting from this distribution scenario we are now interested in describing how actors communicate with the simulation object from which the workflow has been launched (see figure 5). Figure 6 shows the same component class presented in the previous section with the difference that now some distribution related annotations have been added. `@DistributedResource` indicates that objects of this class have to be instantiated on the server (or server group) specified by `targetName`. This target is specified in the host's AoSiF configuration file.

The Initialization. Returning to figure 5, we have a simulation object (SIM object) residing on the PtII Machine that instantiates a workflow which contains, among others, the `ModuleActor` object (ACT object) from figure 6. The configuration file, however, states that PtII and Condor are hosted on different machines. Therefore when the `ModuleActor` object is initialized an aspect, named `DistributedResourceAspect`, processes the `@DistributedResource` annotation and reads the `targetName` parameter of the `ModuleActor` class. The aspect contains an *after* advice that requests the `Dispatcher` to process the newly created resource. Since `targetName="condor_machine"` and this is the PtII machine it decides to serialize and transport the object to the Condor machine.

Transport and Remote Calls. Our implementation of the Dispatcher is actually composed of two components: a routing component which is called by the aspect processing the `@DistributedResource` annotation and a Webservice component. The routing component selects the target server, based on the configuration file and the parameters of the annotation, serializes the object(s), and calls the Webservice component on the specified target server. This one processes the call, deserializes the object and stores a reference in the server's registry. One thing to note is that the deserialization process also comprises a call to the class's constructor. This means that the hashcode of the deserialized object will be different from the one the object had before being serialized. The dispatcher therefore creates and sends along with the resource object another object of the type `ResourceBase` which contains all the information regarding the object being transported, including the old hashcode. The dispatcher also inserts a reference to the object being transported



in the local registry of the Condor machine (a simple `Hashtable` object). Both objects are serialized and stored in a `ResourceContainer` object and sent from the PtII machine to the Condor machine as arguments of the *host* method of the Web-service component of the dispatcher. Here the new hashcode is mapped to the old one and the deserialized object becomes a mirror of the initial object. Further calls to methods of the ACT object residing on the PtII machine will be forwarded to the mirrored ACT object from the Condor machine in a similar fashion. Figure 5 also illustrates how these calls are processed and forwarded by the dispatcher synchronously or asynchronously. The only difference from the initialization phase is that now an *around* advice is used instead of an *after* advice, meaning that the `start` method is called on the ACT object but is actually executed on the mirrored ACT object.

Handling Callbacks. The same mechanism as the one described above is used to perform callbacks to the parent resource in the asynchronous case. Remember that at this point the `setSimulation` method of the `ModuleActor` has already been locally[4] invoked by the `ActorAspect` in order to provide the `ModuleActor` with a reference to the `Simulation` object. Moreover, all arguments of the constructors and methods of the remote object are also mirrored on the target server. This way a mirrored `Simulation` object is available to the mirrored `ModuleActor` object. Now, looking at the implementation of the `start` method of the `ModuleActor` object in figure 6 one can notice that the callback is performed by invoking the `receiveActorEvent` method of the mirrored `Simulation` object. This time the distribution scheme is inverted: the `Simulation` class has also been annotated with `@DistributedResource` but its `targetName="ptolemy_machine"`. The dispatcher uses the Condor machine's AoSiF configuration file to identify the target and forwards the call to the original `Simulation` object hosted by the PtII machine. Finally, the `Simulation` object could optionally call the `getResults` method of the `ModuleActor` to get the results of the calculation.

Distribution Features and Policies. At this point it becomes clear that regardless of how many machines are used to distribute objects, on each of them the same version of the application is needed. The only thing that differs from machine to machine is the AoSiF configuration file. The dispatcher uses reflection to resolve the type of the object being transported as well as for all the other mirrored objects. This allows for a very simple web service interface with only a few web methods that can handle any kind of object types.

We have used the `@DistributedResource` annotation to exemplify the distribution mechanism. AoSiF provides three other types of annotation listed in table I. `@DistributedConstructor` can be used together with `@DistributedResource` indicating that only this constructor has to be invoked on a different server than the rest of the methods of the class. This is useful for database constructors. There are two policies that can be used together with this annotation as parameters to it:

[4]Notice the `@NotDistributable` annotation over this method stating that it should be invoked locally on all original or mirrored objects.

```
@DistributedResource(
    targetName="condor_machine",
    cleanupMethod="getResults",
    transportable=true
)
public class ModuleActor implements Serializable{
    String paramName = null;
    Object simulation = null;

    @NotDistributable
    @ActorConstructor
    public ModuleActor(){}

    @NotDistributable
    @ActorParameter(name="nameParameter")
    public void setParamName(String paramName){this.paramName = paramName;}

    @ActorAction(actionMethods={ActionType.Fire},callingOrder=0)
    @ActorPort(name="outputToken",direction=PortDirection.Output)
    public Object start(
            @ActorPort(name="inputToken",direction=PortDirection.Input)
            Object inputToken
    )throws Exception{
        // doing some work here ...
        Simulation sim = (Simulation)simulation;
        if(sim != null)
            sim.receiveActorEvent(this, ActorEventType.Update);
        return inputToken;
    }

    public Object getResults(){ // return some results set }

    @NotDistributable
    public void setSimulation(Object simulation){this.simulation = simulation;}
}
```

Figure 6: A distributed actor class.

`Static` and `Dynamic` distribution modes. `Static` distribution means that the annotated constructor needs to be invoked and mirrored on a remote server because other remote method calls will follow. `Dynamic` distribution means that after having invoked the constructor on a remote server, the newly created object is returned to the initiating host without mirroring it on the remote host. The `@DistributedMethod` annotation's effect is similar with the difference that no record of the object exists on the remote server by the time the annotated method was invoked. This type of behavior is desired when a distributed class contains methods that make use of hardware or software resources that are spread on several servers. Finally, whenever present the `@NotDistributable` annotation simply cancels the effects of the other annotations for classes, methods, and constructors.

Related Work

Although there are a number of approaches for aspect-oriented distribution of objects [17, 18, 19, 20] which build upon different middleware technologies, none of these approaches uses Java annotations or similar declarative programming techniques. Furthermore, to the best knowledge of the authors, our approach is the only

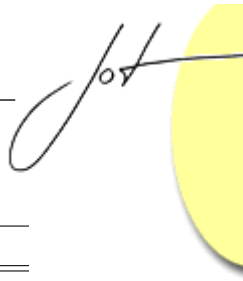


Table 1: Distribution related annotations

Annotation	*Target*
`@DistributedResource`	Class
Effect: indicates that the objects of this class are to be hosted by a specific server or server group.	
`@DistributedConstructor`	Constructor
Effect: indicates that this constructor has to be invoked on a specific server or server group. When present, this annotation overrides the effect of the @DistributedResource annotation.	
`@DistributedMethod`	Method
Effect: indicates that this method has to be invoked on a specific server or server group. When present, this annotation overrides the effect of the @DistributedResource annotation.	
`@NotDistributable`	Constructor or Method
Effect: indicates that this method or constructor has to be invoked locally. When present, this annotation overrides the effect of all the other distribution related annotations.	

one to use Webservices as middleware technology. In [21] XDoclet [22] templates are used to generate aspect code. XDoclet comments are the precursors of Java annotations but here they are used to annotated the aspects rather than the code of the core application.

There are also a number of development efforts related to distributed Ptolemy II actors and workflows [23, 24], none of which being based on aspect-oriented programming. In [25] the need for the separation of concerns in actor-oriented heterogeneous modeling is mentioned. Here the concerns of communication, conservation of internal semantics, and adaptation to the host model of computation are identified when it comes to domain-polymorphic component design [26] and the authors plan on using AOP for future developments. Concerning the aspect-oriented automated integration of existing software components into workflow engines, we have no knowledge of such approaches from the research literature that is publicly available.

Other aspect based approaches for decoupling cross-cutting concerns from the core software architecture target the concerns of quality of service [15], automated software updates [27], and fault tolerance [27, 28, 15]. Security [29] and persistence [30] annotations are already standardized Java 5.0 features. There is however no aspect based implementation of the actual security and persistence features that would eliminate the need of writing additional code for the security and persistence related business-logic.

Conclusion and Future Work

We have presented a new approach for the development of simulation software which relies on the combination of aspect-oriented, declarative, and reflective programming. We have provided a proof of concept for this new approach by implementing the concerns of workflow engine integration and distribution using AspectJ, Java annotations, and reflection for a dispersion calculation simulation application. In essence, the method we presented enables the reuse of legacy simulation codes and their integration into new applications for different research, commercial, and educational purposes. We have shown that the code needed to encapsulate and integrate computer codes into PtII workflows is minimal and therefore allows for rapid application development. The same applies when it comes to the concern of distribution. By following the proposed aspect-oriented approach, research institutes have more freedom in choosing how to implement different concerns in simulation software and can even outsource their implementation. This higher degree of modularity allows for better quality and a longer lifespan of simulation software while also reducing the time needed for the development of end user simulation applications.

We created an open source library called AoSiF[5] that we intend to extend with the implementations of other concerns related to but not limited to access control, persistence, and fault tolerance.

REFERENCES

[1] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *ECOOP '97 - Object-Oriented Programming*. Springer-Verlag Berlin, 1997, pp. 220–242.

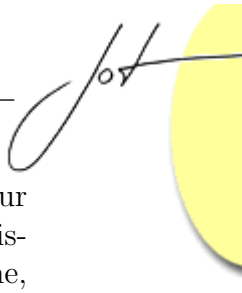
[2] G. Agha, *Actors: a model of concurrent computation in distributed systems*. Cambridge, MA, USA: MIT Press, 1986.

[3] C. Zhang and H.-A. Jacobsen, "Resolving feature convolution in middleware systems," *SIGPLAN Not.*, vol. 39, no. 10, pp. 188–205, 2004.

[4] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of aspectj," in *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*. London, UK: Springer-Verlag, 2001, pp. 327–353.

[5] W. Hrster and T. Wilbois, "Early warning and risk management an interdisciplinary approach," *Information Technologies in Environmental Engineering*, vol. 7, pp. 343–356, 2004.

[5] AoSiF can be downloaded at http://code.google.com/p/AoSiF.



[6] A. Grohman, "Entwicklung und erprobung eines dienstleistungskonzepts zur integration von simulationen in die kernreaktor-fernberwachung," Ph.D. dissertation, Universitt Stuttgart, Institut fr Kernenergetik und Energiesysteme, 2002.

[7] OMG, "The common object request broker: Architecture and specification," 1995.

[8] W3C, "Simple object access protocol 1.1," 2003.

[9] K. R. der Luft im VDI und DIN, "Richtlinie vdi 3945 blatt 3. umweltmeteorologie. atmosphrische ausbreitungsmodelle. partikelmodell." 2007.

[10] R. S., "A 3d lagrangian particle model for direct plume gamma dose rate calculations," *Journal of Radiological Protection*, vol. 21, pp. 145–154(10), 2001.

[11] A. Piater, T. B. Ionescu, and W. Scheuermann, "A distributed simulation framework for mission critical systems in nuclear engineering and radiological protection," *INT J COMPUT COMMUN CONTROL*, vol. 3, no. Suppl. Issue - ICCCC 2008, pp. 448–453, 2008.

[12] M. Weigele, "Berechnung der nassen deposition von spurenstoffen im rahmen des notfallschutzes," Ph.D. dissertation, Universitt Stuttgart, Institut fr Kernenergetik und Energiesysteme, 1997.

[13] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, "Taming heterogeneitythe ptolemy approach," in *Proceedings of the IEEE*, vol. 91, 2003, pp. 127–144.

[14] M. B. Juric, I. Rozman, M. Hericko, A. P. Stevens, and S. Nash, "Java 2 distributed object models performance analysis, comparison and optimization," in *ICPADS '00: Proceedings of the Seventh International Conference on Parallel and Distributed Systems (ICPADS'00)*. Washington, DC, USA: IEEE Computer Society, 2000, p. 239.

[15] G. Duzan, J. Loyall, R. Schantz, R. Shapiro, and J. Zinky, "Building adaptive distributed applications with middleware and aspects," in *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*. New York, NY, USA: ACM, 2004, pp. 66–73.

[16] D. Thain, T. Tannenbaum, and M. Livny, "Condor and the grid," in *Grid Computing: Making the Global Infrastructure a Reality*, F. Berman, G. Fox, and A. Hey, Eds. John Wiley & Sons Inc., April 2003.

[17] P. Soule, T. Carnduff, and S. Lewis, "A distribution definition language for the automated distribution of java objects," in *DSAL '07: Proceedings of the 2nd workshop on Domain specific aspect languages*. New York, NY, USA: ACM, 2007, p. 2.

[18] M. Ceccato and P. Tonella, "Adding distribution to existing applications by means of aspect-oriented programming," in *SCAM '04: Proceedings of the Source Code Analysis and Manipulation, Fourth IEEE International Workshop*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 107–116.

[19] S. Soares, E. Laureano, and P. Borba, "Implementing distribution and persistence aspects with aspectj," *SIGPLAN Not.*, vol. 37, no. 11, pp. 174–190, 2002.

[20] M. Nishizawa and S. Chiba, "Jarcler: Aspect-oriented middleware for distributed software in java," in *Dept. of Math. and Comp. Sciences Research Reports C-164, Tokyo Institute of Technology*, 2002.

[21] E. Tilevich, S. Urbanski, Y. Smaragdakis, and M. Fleury, "Aspectizing server-side distribution," *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, pp. 130–141, Oct. 2003.

[22] C. Walls, N. Richards, and R. Oberg, *XDoclet in Action (In Action series)*. Greenwich, CT, USA: Manning Publications Co., 2003.

[23] D. L. Cuadrado, A. P. Ravn, and P. Koch, "Automated distributed simulation in ptolemy ii," in *PDCN'07: Proceedings of the 25th conference on Proceedings of the 25th IASTED International Multi-Conference*. Anaheim, CA, USA: ACTA Press, 2007, pp. 139–144.

[24] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao, "Scientific workflow management and the kepler system: Research articles," *Concurr. Comput. : Pract. Exper.*, vol. 18, no. 10, pp. 1039–1065, 2006.

[25] M. Feredj, F. Boulanger, and A. M. Mbobi, "A model of domain-polymorph component for heterogeneous system design," *J. Syst. Softw.*, vol. 82, no. 1, pp. 112–120, 2009.

[26] M. Feredj, F. Boulanger, and M. Mbobi, "An approach for domain-polymorph component design," *Information Reuse and Integration, 2004. IRI 2004. Proceedings of the 2004 IEEE International Conference on*, pp. 145–150, Nov. 2004.

[27] S. Fleissner and E. L. A. Baniassad, "Epi-aspects: aspect-oriented conscientious software," *SIGPLAN Not.*, vol. 42, no. 10, pp. 659–674, 2007.

[28] S. Bouchenak, N. D. Palma, S. Fontaine, and B. Tête, "Aosd for internet service clusters: the case of availability," in *AOMD '05: Proceedings of the 1st workshop on Aspect-oriented middleware development*. New York, NY, USA: ACM, 2005.

[29] S. Microsystems, "Security annotations and authorization in glassfish and the java ee 5 sdk," 2006.

[30] C. Bauer and G. King, *Hibernate in Action (In Action series)*. Greenwich, CT, USA: Manning Publications Co., 2004.

ABOUT THE AUTHORS

Tudor Basarab Ionescu is a doctoral candidate at the Institute of Nuclear Technology and Energy Systems of the University of Stuttgart, Germany and a member of the SimTech Cluster of Excellence. His research interests include dispersion modeling and simulation, software engineering and evolution, and parallel and distributed computing. He can be reached at ionescu@ike.uni-stuttgart.de.

Andreas Piater is a postdoctoral researcher at the Institute of Nuclear Technology and Energy Systems of the University of Stuttgart, Germany. His research interests are simulation of complex systems, software engineering, and knowledge systems. He can be reached at piater@ike.uni-stuttgart.de.

Walter Scheuermann has received his doctorate degree from the University of Stuttgart and is now the head of the Knowledge Engineering Department of the Institute of Nuclear Technology and Energy Systems of the University of Stuttgart, Germany. His research interests are simulation of complex systems, dispersion modeling and simulation, and parallel and distributed computing. He can be reached at scheuermann@ike.uni-stuttgart.de.

Eckart Laurien is a full professor and the head of the Institute of Nuclear Technology and Energy Systems of the University of Stuttgart, Germany. 1985 – Dissertation (Dr.-Ing), University of Karlsruhe, 1986/87 – Research Fellowship, University of Arizona, Tucson (USA), Aerospace and Mechanical Engineering, 1987/88 – Research Associate, University of Colorado, Boulder (USA), Mechanical Engineering Sciences, 1995 – Habilitation (Dr.-Ing. habil.), University of Braunschweig, Institute for Fluid Mechanics, 1996 – Professor, University of Stuttgart, Departement for Thermofluiddynamics. He can be reached at laurien@ike.uni-stuttgart.de.