PLATFORM INDEPENDENT SPECIFICATION OF SIMULATION MODEL COMPONENTS

Mathias Röhl
University of Rostock
Institute of Computer Science
Albert-Einstein-Str. 21, D-18059 Rostock, Germany
Email: mroehl@informatik.uni-rostock.de

Abstract— **Simulation model composability is a highly debated issue, especially the specific requirements of model components as compared to software components. Selected software component techniques are reviewed and adapted according to simulation models needs. Standard technologies like UML and XML are exploited to form the basis for a specification layer that wraps model definitions. This layer accounts for explicit dependencies, compositions, and parametrization of simulation models. Thereby, the specification of model components can be done in a platform and simulation system independent manner. For the purpose of simulation a mapping to a concrete modeling formalism becomes necessary. This is here done on the example of the Parallel DEVS formalism.**

Keywords— **Model Components, XML, UML, Parallel DEVS**

Introduction

The need to combine models whose development is spread in space and time increases. Middleware approaches like HLA (IEEE, 2000) focus on the interoperability of entire simulation systems. Thereby, the integration of very heterogeneous models is supported. HLA is based on low-level synchronization between simulation systems via events (Tolk, 2002). More recent work has shifted focus from pure wiring solutions to the semantic dimensions of interoperability (Brutzman et al., 2002). Today, it is accepted that interoperability is necessary but not sufficient to compose complex simulation models (Tolk and Muguira, 2003) as the composition of models alludes to the grand challenge of re-using models (Overstreet et al., 2002).

There is a vivid discussion about composability of simulation models and how simulation model components differ from software components. While many simulationists stress the differences between software and model components (Davis and Anderson, 2004; Kasputis and Ng, 2000), some argue that software components and model components face basically the same challenges. This paper takes a pragmatic approach to the problem of model composition by reviewing existing software component solutions and adapting them according to their suitability in the context of simulation. UML specifications of components and compositions (OMG, 2005), XML representations of entities, and the Parallel DEVS formalism (Zeigler et al., 2000) will serve as the integrative cornerstones of the presented approach.

The paper is structured as follows. It starts with a short introduction to software components. Subsequently, it relates them to the problem of simulation model composability and the special requirements of model components. Four central cornerstones for a simulation model component approach are identified and arranged with activities of a simulation study. Each cornerstone is presented in detail and illustrated on simple examples. Finally, related work is discussed.

Composability

Literature on software components (Szyperski, 2002) defines a component as a unit of independent deployment. Components are separated from their environments and encapsulate their constituent features. These characteristics serve the need of *composition* which refers to the "Assembly of parts (components) into a whole (a composite) without modifying the parts." (Szyperski, 2002).

To be composable by third parties the requirements and provisions of a component have to be clearly specified. In software component approaches this is usually done with the help of interfaces. References to interfaces instead of implementations allow components to avoid direct dependencies. The implementation part is hidden for others and is not relevant for consistency checks.

Interfaces are not part of a component but situated between components (Szyperski, 2002), i.e. their existence is not bound to that of a particular component. Components do not own interfaces but merely refer to them for the purpose of publishing provided and required services. Thereby, providers and users of services are decoupled, same as the development of the corresponding provider and customer components.

Besides the ability to let components expose well-defined interfaces that allow connecting them, components should be flexible enough to be useful in a variety of circumstances. Configurability enables reuse of components without modifying their internal implementations. The Corba Component model uses attributes (OMG, 2002) to let components represent a whole class of implementations.

Simulation Model Composability

A recent definition of simulation model composability reads: "Composability is the capability to select and assemble simulation components in various combinations into simulation systems to satisfy specific user requirements." (Morse et al., 2004). Basically this def-

Proceedings 20th European Conference on Modelling and Simulation
Wolfgang Borutzky, Alessandra Orsoni, Richard Zobel © ECMS, 2006
ISBN 0-9553018-0-7 / ISBN 0-9553018-1-5 (CD)

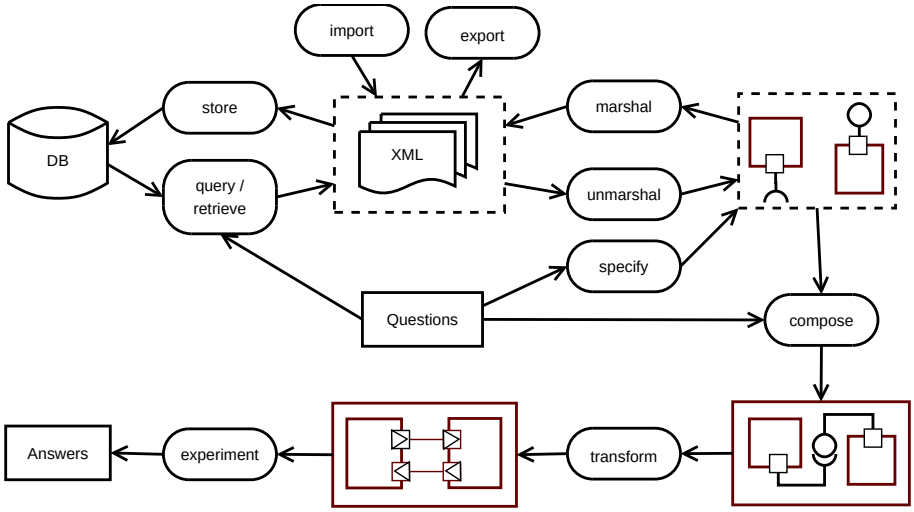

Fig. 1. Activities and Resources of a Simulation Study that involve Model Components

inition is in line with software components. A clear and precise specification of dependencies is interpreted as a fundamental requisite for a component-based design also in the context of simulation (Yilmaz, 2004b).

To compose models components dependencies between components have to be evaluated. Components have to be connected according to their interface specifications. While software components are usually either present or not and thereby services are either provided or not, within a simulation model often a multitude of instances of the same component becomes necessary.

An important difference between model and software components arises from the fact that a simulation model shall represent a system having a certain objective (set of experiments) in mind. Modeling implies purpose-driven abstraction. Model components have to be enriched with meta data descriptions accounting for the context of use and the level of abstraction. Various publications emphasize this as one of the central requirements distinguishing simulation model components from software components (Davis and Anderson, 2004; Yilmaz, 2004a; Heisel et al., 2004). Meta data descriptions require high flexibility, e.g. to incorporate application domain specific parts (Uhrmacher et al., 2005). XML-based solutions may provided this flexibility (Harold, 2002).

While component specifications and compositions work on the conceptual level, simulation requires executable models. Each simulator supports one or a set of modeling formalisms. Thus, in order to simulate, component specifications have to be related to model formalisms. Compositions and connections have to be *transformed* to model elements of a concrete simulation modeling formalism, which eventually can be executed and *experimented* with.

Within a component-based design of simulation models we distinguish four "cornerstones":

- XML documents
- Single Components
- Compositions
- Simulation Model

Figure 1 arranges these four cornerstones clockwise into activities of a simulation study and shows their intertwining. Database activities like *storing*, *querying*, and *retrieving* are completely based on XML representations of model entities. XML eases the *import* and *export* of model components. For the purpose of usage the XML representations of building blocks have to be converted from/to objects of a programming language by *marshal* and *unmarshal* operations (Röhl and Uhrmacher, 2005). Taking into account the purpose of the simulation study complex models can then be *composed*. Please note, that questions that motivate a simulation study have not only a direct impact on the specification of models but gain influence on the selection/retrieval and composition of model components. The driving idea behind this is to decouple specification and use of models.

In the following we will take a closer look at each of the four cornerstones and their associated activities. We start with the conceptual layer by introducing UML notations for components and compositions. Afterwards, an XML representation for simulation model components will be presented. Finally the transformation of the composite to a simulation model is shown.

Single Components

The Unified Modeling Language (OMG, 2005) provides a standard notation to describe components and interfaces. Component diagrams of UML 2.0 specify components in accordance to the discussion above, namely as a "modular unit with well-defined interfaces that is replaceable within its environment". We will use the graphical means of UML diagrams for introducing basic ideas.

UML 2.0 component descriptions comprise a set of ports, a set of parts, a set of connectors, and a behavior. Ports are used to announce points of interactions of a component by means of interfaces. Ports that rep-

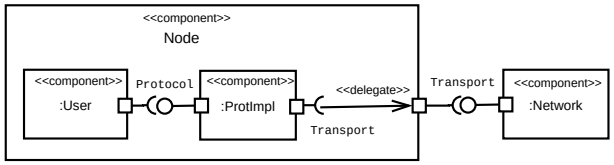

Fig. 2. Compositions with Connections based on Facets and Receptacles

resent required interfaces are called *receptacles* and are denoted by antennas. Provided interfaces are called *facets* and look like lollipops.

Figure 2 contains the UML notation for a component *ProtImpl* that has a facet of type *Protocol* and a receptacle of type *Transport*.

Compositions

For composition providers and consumers have to be connected via their required and provided interfaces. Components may be connected in two different ways. First we can connect provided and required ports by a so called *assembly* connector. In contrast, the *delegation* connector connects two ports of the same type between a component and one of its sub components. Thereby, hierarchical components become possible.

Figure 2 shows the *User* and *ProtImpl* components connected via an assembly connector between their *Protocol* ports. The composite component *Node* delegates the *Transport* receptacle of the *ProtImpl* sub comopnent to its own *Transport* port. The receptacle of the node component is finally connected to a network component that offers an according facet.

XML Documents

The XML representation of UML specifications (XMI) bears overhead that does not account for simulation purposes. UML ports can have multiple interfaces and interfaces are usually based on method declarations. In the following we develop a small but expressive XML Schema Definition for simulation of model components that defines component syntax tailored for simulation purposes.

Storage within and retrieval from a database requires unique identification of entities. To this end identifiers are introduced that comprise a name and a version.

```
<xsd:complexType name="Id">
 <xsd:attribute name="name" type="xsd:string"
                use="required"/>
 <xsd:attribute name="version" type="xsd:string"
                use="required"/>
</xsd:complexType>
```

Late binding of components is realized by interfaces. Interfaces typically contain method declarations in the realm of software engineering. Within discrete-event modeling and simulation models do not call methods of other models but exchange events with each other. This is actually a relaxing condition since software component approaches like the Corba Component Model have to account for both method invocations and event passing (OMG, 2002).

A model interface definition comprises a unique identifier and a set of event port declarations that allow exchanging typed events. Events may flow in two directions. The boolean attribute *isInput* indicates whether a port declaration denotes a potential flow of events towards an entity, or it declares a port that is intended to emit events from an entity.

```
<xsd:complexType name="Interface">
  <xsd:sequence>
    <xsd:element name="id" type="Id"/>
    <xsd:element name="decl" type="PortDeclaration"
                 minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="PortDeclaration">
  <xsd:attribute name="name" type="xsd:string"
                 use="required"/>
  <xsd:attribute name="type" type="xsd:string"
                 use="required"/>
  <xsd:attribute name="isInput" type="xsd:boolean"
                 use="required"/>
</xsd:complexType>
```

Component instances need to refer to interfaces to exhibit provided interfaces (facets) and required interfaces (receptacles). A Port refers to exactly one interface specification by means of the interface's identifier and a name attached to it. Using ports instead of direct references to interfaces increases flexibility as multiple ports can be typed by the same interface while carrying different names. The boolean attribute *isFacet* is used to distinguish provided from required ports.

```
<xsd:complexType name="Port">
 <xsd:sequence>
  <xsd:element name="type" type="Id"/>
 </xsd:sequence>
 <xsd:attribute name="name" type="xsd:string"
                use="required"/>
 <xsd:attribute name="isFacet" type="xsd:boolean"
                use="required"/>
</xsd:complexType>
```

According to the above discussion we introduce a type that allows parametrization of model components.

```
<xsd:complexType name="Parameter">
 <xsd:attribute name="name" type="xsd:string"
                use="required"/>
 <xsd:attribute name="type" type="xsd:string"
                use="required"/>
 <xsd:attribute name="value" type="xsd:string"
                use="required"/>
 <xsd:attribute name="description" type="xsd:string"/>
</xsd:complexType>
```

The type *Parameter* supports simple parameters directly. Expectation values for random distributions, e.g. mean waiting or activity times, may be useful parameters for model components. For the purpose of parameter delegation the type *Composition* can be annotated with parameter values.

```
<xsd:complexType name="Composition">
 <xsd:sequence>
  <xsd:element name="type" type="Id"/>
  <xsd:element name="name" type="xsd:string"/>
  <xsd:element name="parameter" type="Parameter"
               minOccurs="0" maxOccurs="unbounded"/>
 </xsd:sequence>
</xsd:complexType>
```

The definition of *Connector* keeps the distinction between assembly and delegation connectors implicit. A connector is interpreted as an assembly connector if it connects two sub components. If the connection involves the component and own of its sub components it becomes interpreted as a delegation connector.

```
<xsd:complexType name="Connector">
 <xsd:attribute name="fromComponent" type="xsd:string"
                use="required"/>
 <xsd:attribute name="fromPort" type="xsd:string"
                use="required"/>
 <xsd:attribute name="toComponent" type="xsd:string"
                use="required"/>
 <xsd:attribute name="toPort" type="xsd:string"
                use="required"/>
</xsd:complexType>
```

Given the type definitions above we are prepared for defining a component type:

```
<xsd:complexType name="Component">
 <xsd:sequence>
  <xsd:element name="id" type="Id"/>
  <xsd:element name="param" type="Parameter"
               minOccurs="0" maxOccurs="unbounded"/>
  <xsd:element name="mapper" type="xsd:string"
               minOccurs="0"/>
  <xsd:element name="port" type="Port"
               minOccurs="0" maxOccurs="unbounded"/>
  <xsd:element name="component" type="Composition"
               minOccurs="0" maxOccurs="unbounded"/>
  <xsd:element name="connection" type="Connection"
               minOccurs="0" maxOccurs="unbounded"/>
  <xsd:element name="model" type="Id"/>
  <xsd:element name="portmapping" type="PortMapping"
               minOccurs="0" maxOccurs="unbounded"/>
 </xsd:sequence>
</xsd:complexType>
```

The definition of *Component* contains three elements that were not mentioned before, namely a reference to a model, a mapper, and port mappings.
Very little is assumed here about the shape of a model. It is only required that for the purpose of identification each model has a unique identifier. One strength of this approach lays in the independence from a concrete modeling formalism, i.e. a component instance documents can refer to any model definitions as its type derives from *Model*.

Unlike software components where components are executable implementations, our model components are declarative in nature. Evaluation of parameter values can neither be done by a model component itself nor by the simulator. Simulators do not normally know about parameters. Thus, we need a third entity that maps parameter values to elements of a component. The *mapper* is exactly for this purpose. At this point nothing is said about the language a mapper may be specified in. XSLT would be a good choice, which can be specified itself in XML and thereby preserves platform independence of the component specification. Nevertheless, mappers may also be specified in a programming language, as it is the case in our current realization of the component framework, which uses Java.

Instances of type *PortMapping* associate the declarations of a component's published ports to "real" ports of the contained model. For reasons of brevity we omit the complete definition of this type.

Let us take a look at a simple example. Assume we want to develop a model to simulate users in a network. Users should communicate via a certain protocol. We start with the definition of the *Protocol* interface. This interface declares event flow of type *Call* and of type *Response*.

```
<interface>
  <id name="Protocol" version="1.0"/>
  <decl name="call" type="Call" isInput="true"/>
  <decl name="response" type="Response" isInput="false"/>
</interface>
```

Now we could start to implement the *User* component. Additionally, we could develop different versions of it. Each time we have to explicitly state that it assumes another component to exist that behaves according to the `Protocol` interface. Independently from this component we can develop a *ProtImpl* component, that provides the required interface.

Within the simulation model we need to compose these two components and plug them together, i.e. to connect receptacles to according facets. This is done inside the *Node* component. It declares *User* and *ProtImpl* as compositions and connects their *Protocol* ports via an assembly connector. Furthermore the *Node* component connects its own *Transport* receptacle to that of the *ProtImpl* component by a delegation connector.

```
<component
 <id name="Node" version="1.0"/>
  <param name="id" type="int" value="0"/>
  <param name="radiorange" type="double"
    value="250"/>
  <mapper>unihro.com.node.v1.Mapper</mapper>
  <port name="transport" isFacet="false">
    <type name="Transport" version="1.0"/>
  </port>
  ...
  <composition>
   <type name="User" version="1.0"/>
   <name>user</name>
  </composition>
  <composition>
   <type name="ProtImpl" version="1.0"/>
   <name>protocol</name>
  </composition>
  <connection fromComponent="protocol"
    fromPort="protocol"
    toComponent="user" toPort="protocol"/>
  <connection fromComponent="this"
    fromPort="transport"
    toComponent="protocol" toPort="transport"/>
  ...
</component>
```

For this composition to work, the *User* component has to provide a receptacle with type *Protocol*.

```
<component id="User" version="1.0">
 ...
 <receptacle name="protocol">
   <type name="Protocol" version="1.0"/>
 </receptacle>
 ...
 <model name="UserModel" version="1.0"/>
</component>
```

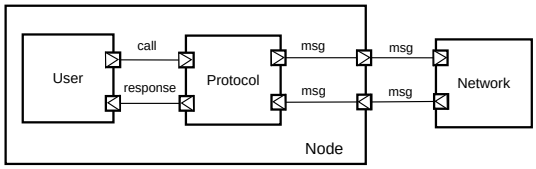

Fig. 3. Composition from Fig. 2 Mapped to a Parallel DEVS Model

Analogously, the *ProtImpl* component has to exhibit a protocol facet. Thereby, compositions rely only on XML specifications and direct dependencies between the implementation parts of the *User* and *ProtImpl* components are eliminated.

Simulation Models

By specifying components we are not yet able to execute experiments. We need to transform the component constructs to models represented within an executable modeling formalism. For the component and composition specifications presented above this can be done employing the Parallel DEVS formalism (Zeigler et al., 2000).

A coupled Parallel DEVS model $\mathscr{N}$ is specified by a tuple $\langle X, Y, D, \{M_d | d \in D\}, EIC, EOC, IC\rangle$, where $d \in D$. $X = \{(i,v) | i \in InPorts, v \in X_i\}$ denotes the set of input ports and values. $Y = \{(o,v) | o \in OutPorts, v \in Y_o\}$ is the set of output ports and values. D is a set of component references (names). For each $d \in D$, M_d is either an atomic or a coupled PDEVS model. EIC is the set of external input couplings that connect input ports of the coupled model to input ports of sub models. EOC are external output couplings and IC denote internal couplings between sub models.

Please note the difference between DEVS model ports and component ports as introduced above. While DEVS ports are typed by value ranges, component ports refer to a certain interface to declare the existence of model ports. When mapping component structures to simulation models, the models must ensure that they have model ports according to the ports declared by their published interfaces. On the provider side input port declarations require the actual model to have an according input port while on the customer side they require the model to have an output port respectively.

Using Parallel DEVS as the target formalism the mapping of component connections to model couplings and their association with model ports becomes relatively easy. Each component connection maps to a set of couplings $c \in EIC \cup EOC \cup IC$. Assembly connectors map to internal couplings and delegation connectors map to external input and output couplings. At the end, for ensuring consistency, it has to be checked for each receptacle of a component whether it got connected.

Our examples use Parallel DEVS not only as the target formalism for mapping component specifications but also for specifying model behavior. If other formalisms than Parallel DEVS should be used for specifying behavior, transformations have to be executed. To this end, the target modeling formalism has to be at least so expressive like the implementation formalism. This does not constrain the generality of the presented approach as Parallel DEVS in principle can be used as the target for many formalism transformations (Vangheluwe, 2000).

Related Work

DEVS has gained a widespread use as formalism for specifying model components. DEVS supports delayed binding of model parts. A model does not itself determine where an output will be forwarded to. More specifically DEVS allows specifying of modular systems that receive inputs and produces outputs over a time base depending on their (hidden) state). Putting a model into a certain context is within the responsibility of the surrounding coupled model. However, existing DEVS component approaches (Zeigler and Sarjoughian, 2002; MacSween and Wainer, 2004) lack explicit compositions based on published interfaces. Thereby, current DEVS solutions provide the means for hierarchical model construction but not for the *act* of composition.

The System Entity Structure/Model Base (SES/MB) framework equips DEVS with an additional layer that allows to manage alternative decompositions of models. While component approaches are usually bottom-up and are all about the integration of third party components, SES provides a centralized top-down approach that aims at representing possible structures of a system (its design space) (Zeigler et al., 2000). SES organizes families of models for the purpose of systematic experimentation.

Software component technologies are already utilized for simulation model components at the level of source code, e.g. using Microsoft's component standard COM (Cho and Kim, 2002) or building upon CORBA (Zeigler et al., 1999).

The component approach presented here is complementary to research on semantical compositions. Our specification of model component approach is open for integration of stronger semantic consistency checking, e.g. as described by (Yilmaz, 2004a) who specified interaction constraints of DEVS models in the temporal dimension explicitly by means of I/O system specification. These constraints could be integrated into the XML representation as Meta Data descriptions. The same pertains to application domain specific model annotations, e.g. as in (Uhrmacher et al., 2005).

(Zinoviev, 2005) suggests another mapping between DEVS models and UML component diagrams. His UML representation is not intended to provide a unified representation on top of a modeling formalism for the purpose of composition but as a replacement for the original DEVS specification.

Our approach is close to that of BOMs (Gustavson

and Chase, 2004) in exploiting the flexibility of XML and XSD and striving for platform independent specifications. Both approaches suggest transforming platform independent XML representations into platform specific models. While BOMs focus on HLA compliance, the approach presented here puts a strong emphasis on interface definitions according to UML.

Conclusion

To support a component-based design of simulation models concepts of UML were exploited. This applies to the specification of components as well as compositions. Component constructs work on top of simulation model representations facilitating a separation between the public and private part of a model component.

XML forms another important cornerstone for specification of components. Much care was taken to let all public visible parts of simulation model components be specifiable via *XML representations*. To this end, an XML Schema Definition was tailored to model component purposes. The schema is easily extensible to include semantically stronger concepts by means of meta data descriptions.

Finally, the mapping of component constructs to a *simulation model* was shown on the example of the Parallel DEVS formalism. Parallel DEVS fits this purpose very well as its worldview, build upon ports, channels, and hierarchical construction, closely resembles the one of UML component specifications.

The presented concepts were implemented based on the simulation system James II (Himmelspach et al., 2006).

Further work is directed towards bringing specification of simulation model components event closer to the UML, e.g. by the definition of a proper UML 2.0 profile. This would enable standardized exchange of component descriptions by means of XMI.

Acknowledgments

This research is supported by the DFG (German Research Foundation).

References

Brutzman, D., Zyda, M., Pullen, M., , and Morse, K. L. (2002). Extensible modeling and simulation framework (XMSF) — challenges for web-based modeling and simulation. Technical report, SAIC.

Cho, Y. I. and Kim, T. G. (2002). DEVS framework for component-based modeling/simulation of discrete event systems. In *The Proceedings of the 2002 Summer Computer Simulation Conference*.

Davis, P. K. and Anderson, R. H. (2004). Improving the composability of DoD models and simulations. *JDMS*, 1(1):5–17.

Gustavson, P. and Chase, T. (2004). Using XML and BOMs to rapidly compose simulations and simulation environments. In *Proceedings of the 2004 Winter Simulation Conference*, pages 1467–1475.

Harold, E. R. (2002). *Processing XML with Java*. Pearson Education.

Heisel, M., Luethi, J., Uhrmacher, A. M., and Valentin, E. (2004). A description structure for simulation model components. In *Proceedings of the Summer Computer Simulation Conference (SCSC'04)*, San Jose, California, USA.

Himmelspach, J., Röhl, M., and Uhrmacher, A. M. (2006). James II project description. Available online via `http://wwwmosi.informatik.uni-rostock.de/mosi/projects/cosa/james-ii`.

IEEE (2000). Standard for modeling and simulation (M& S) High Level Architecture (HLA) — Framework and Rules. Document 1516-2000.

Kasputis, S. and Ng, H. C. (2000). Composable simulations. In *Proceedings of the 2000 Winter Simulation Conference*, pages 1577–1584.

MacSween, P. and Wainer, G. (2004). On the construction of complex models using reusable components. In *Proceedings of SISO Spring Interoperability Workshop*, Arlington, VA,. USA.

Morse, K. L., Petty, M. D., Reynolds, P. F., Waite, W. F., and Zimmerman, P. M. (2004). Findings and recommendations from the 2003 composable mission space environments workshop. In *Proceedings of the 2004 Spring Simulation Interoperability Workshop*, pages 313–323.

OMG (2002). CORBA Components version 3.0 (document formal/02-06-65). `http://www.omg.org/cgi-bin/doc?formal/02-06-65`.

OMG (2005). UML superstructure specification version 2.0 (document formal/05-07-04). `http://www.omg.org/cgi-bin/doc?formal/05-07-04`.

Overstreet, C. M., Nance, R. E., and Balci, O. (2002). Issues in enhancing model reuse. In *International Conference on Grand Challenges for Modeling and Simulation, Jan. 27-31*, San Antonio, Texas, USA.

Röhl, M. and Uhrmacher, A. M. (2005). Flexible integration of XML into modeling and simulation systems. In *Proceedings of the 2005 Winter Simulation Conference*, pages 1813–1820.

Szyperski, C. (2002). *Component software: beyond object-oriented programming*. ACM Press/Addison-Wesley Publishing Co., 2nd edition.

Tolk, A. (2002). Avoiding another green elephant — a proposal for the next generation hla based on the model driven architecture. In *Simulation Interoperabilty Workshop*, Orlando. SISO.

Tolk, A. and Muguira, J. (2003). The level of conceptual interoperability model.

Uhrmacher, A. M., Degenring, D., Lemcke, J., and Krahmer, M. (2005). Towards re-using model components in systems biology. In *CMSB 2004*, volume 3082 of *LNBI*, pages 192–206. Springer.

Vangheluwe, H. (2000). DEVS as a common denominator for multi-formalism hybrid system modeling. In *Proceedings of the IEEE International Symposium on Computer Aided Control System Design*, pages 129–134, Anchorage, Alaska.

Yilmaz, L. (2004a). On the need for contextualized introspective models to improve reuse and composability of defense simulations. *JDMS*, 1(3):141–151.

Yilmaz, L. (2004b). Verifying collaborative behavior in component-based devs models. *Simulation*, 80(7–8):399–415.

Zeigler, B. P., Kim, D., and Buckley, S. J. (1999). Distributed supply chain simulation in a devs/corba execution environment. In *WSC '99: Proceedings of the 31st conference on Winter simulation*, pages 1333–1340, New York, NY, USA. ACM Press.

Zeigler, B. P., Praehofer, H., and Kim, T. G. (2000). *Theory of Modeling and Simulation*. Academic Press, London, 2nd edition.

Zeigler, B. P. and Sarjoughian, H. S. (2002). Implications of m& s foundations for the v& v of large scale complex simulation models. In *Proceedings of the Foundations for V&V in the 21st Century Workshop*, Laurel, MD.

Zinoviev, D. (2005). Mapping DEVS models onto UML models. In *Proc. of the 2005 DEVS Integrative M&S Symposium*, pages 101–106, San Diego, CA.

Author Biographies

Mathias Röhl was born in Ludwigslust, Germany. He studied computer science at the University of Rostock and received his MSc in 2002. Since then he has worked as a research scientist at the Modeling and Simulation Group at the University of Rostock. His Web address is `http://www.informatik.uni-rostock.de/~mroehl`.