Proceedings of the 2006 Winter Simulation Conference
L. F. Perrone, F. P. Wieland, J. Liu, B. G. Lawson, D. M. Nicol, and R. M. Fujimoto, eds.

COMPOSING SIMULATIONS FROM XML-SPECIFIED MODEL COMPONENTS

Mathias Röhl
Adelinde M. Uhrmacher

University of Rostock
Department of Computer Science
Albert-Einstein-Str. 21, D-18059 Rostock, GERMANY

ABSTRACT

This paper is about the flexible composition of efficient simulation models. It presents the realization of a component framework that can be added as an additional layer on top of simulation systems. It builds upon platform independent specifications of components in XML to evaluate dependency relationships and parameters during composition. The process of composition is split up into four stages. Starting from XML documents component instances are created. These can be customized and arranged to form a composition. Finally, a composition is transformed to an executable simulation model. The first three stages are general applicable to simulation systems; the last one depends on the Parallel DEVS formalism and the simulation system James II.

1 INTRODUCTION

The deficiencies of low-level interoperability approaches like HLA have motivated research on component-based simulation and shifted focus to higher levels of interoperability (Tolk and Muguira 2003). Semantic consistency checks are recognized to play an important role for composing simulation models (Petty, Weisel, and Mielke 2005), e.g., to account for the context of use and the level of abstraction of a model (Davis and Anderson 2004, Yilmaz 2004).

However, the use of programming languages is the predominant way to specify simulation model components today (Chen and Szymanski 2002, MacSween and Wainer 2004). Concepts for composability infrastructure (Kasputis and Ng 2000) and discussion on composability (Davis and Anderson 2004) underline the need for model component repositories in general and XML-based solutions in particular (Brutzman et al. 2002).

Based on the concepts discussed in (Röhl 2006) we present a framework for composing simulation models from XML specifications. In a first step, consistency of compositions is checked on a syntactical level.

The paper is structured as follows. The next section introduces the conceptual and architectural cornerstones of the framework. Afterwards, the implementation of the framework is sketched. The composition of a simulation model and the interplay between basic entitiesis of the framework is exemplified on the composition of a network model. The paper concludes with a discussion of related work.

2 TOWARDS A MODEL COMPONENT FRAMEWORK

Figure 1 shows the four stages that make up the conceptual "cornerstones" for composing simulation models (Röhl 2006).

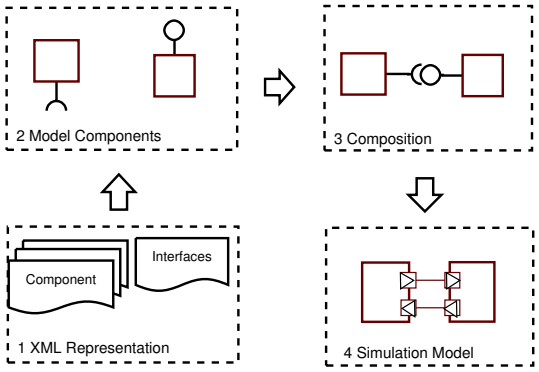

Figure 1: Stages for Composing Simulations

As much as possible of compositions should be specifiable by *XML Documents*. Declarative specifications ease database integration, readability by users, and the development of graphical user interfaces. XML-based solutions are well suited to specify meta data. Furthermore, standard

exchange formats based on XML ease the import and export of model definitions.

As regards contents, the *Component Definitions* orientate on the Unified Modeling Language 2.0 (OMG 2005). Most notably published and required interfaces of each component have to be explicitly specified. Thereby internal details of a component, i.e., the implementation of model behavior, can be hidden and direct dependencies between models eliminated.

A set of components may become customized and arranged to form a *Composition* according to the aim of a simulation study. Parameters set on component instances are evaluated and dependencies between components resolved. Specification of compositions and connections is also based on the UML 2.0.

Execution of experiments needs a *Simulation Model* specified in a certain modeling formalism. The Parallel DEVS formalism (Zeigler, Praehofer, and Kim 2000) is used for representing the executable model. The simulation system James II (Uhrmacher 2006) provides the simulation engine for executing Parallel DEVS simulation models efficiently.

The challenge for realizing a framework according to these concepts stems from the different characteristics of declarative and imperative representations. While declarative representations of components work well for data base activities like storing, querying, and retrieving of components, they bear a significant execution overhead. In contrast, imperative implementations of models are generally more efficient to execute but less eligible for data base integration.

Ideally one wants to combine the advantages of both "worlds", i.e., to compose efficient simulation models flexibly. To this end, the realization of our framework is based on the strict separation between *composition* and *execution* phase. All activities that account for the composition of models work on XML data and are kept strictly pre-executive.

The production of a simulation model is deferred after the composition phase is completed. Simulation solely works with pure model implementations. All XML data and component instances are disposed. The resulting simulation model does neither bear an XML nor a component overhead.

During the composition phase, the use of a programming language is only allowed within a very restricted scope of a component definition. It may be used locally to specify model behavior and parameter mapping. This leads to the distinction between *public* and *private* part of a component. XML is enforced to be used for the entire public part of component definitions. Thereby, the public part of a component is kept independent from a concrete modeling system and a programming language. Specification of model behavior may be done in XML, in a programming language, or a combination thereof. For reasons of simplicity, we restrict the use of programming languages to Java.

2.1 Instantiation of Components

XML documents are the entry points for the framework. Each model component requires at least one XML instance document holding the public (interface) of a component and optionally a second XML document accounting for the private (model implementation) part. For using XML specifications in a tool they have to be transformed to objects accessible via a programming language.

In the framework, XML handling is based solely on entities that are bound to an XML Schema Definition (Röhl and Uhrmacher 2005). Schema Definitions mainly define the syntax of an XML document and thereby provide the means for rendering XML documents valid or invalid. The entity that provides access to XML documents holding component data is bound to the XML Schema given in Röhl (2006). As such it comprises accessors to a set of parameters, a reference to a parameter mapper, a set of ports, a set of sub components, a set of connections (between sub components), and a reference to a model definition.

Bound entities for component and model data allow transparent unmarshaling of XML documents to Java objects and marshaling of Java objects back to XML (Röhl and Uhrmacher 2005).

In contrast to software components, XML is a good choice for defining model behavior. However, for the simulation system James II it showed especially useful to integrate Java specifications of models that use XML merely for the public part of a component. As long as no standard XML format exists for a modeling formalism, as it is the case with atomic DEVS models, tool specific model implementation provide a proper workaround (Röhl and Uhrmacher 2005). Furthermore, this allows to combine new model components with legacy models, as for the latter only an XML wrapper document has to be specified.

To put XML data objects to real use we adopt the idea of component homes as introduced by Enterprise Java Beans (Sun Microsystems 2003) and also used by the Corba Component Model (OMG 2002). A home manages creation of component instances of a certain type according to an XML document. For creating a component the home clones the bound component data and initializes a component instance. Thus, each component instance works on an individual data object. For using a component it usually needs to be customized with parameters, on which we will take a look now.

2.2 Customization

For providing customizable model components we allow to set parameters on component data. Once set, parameters have to be evaluated and internal details of a component to be adapted according to the parameter's values. The difficulty is now, that parameter evaluation cannot be done by the

component definition itself. Our component definition is an entirely declarative one holding only component data in XML. Furthermore, the implementation of model behavior is usually done within a certain modeling formalism that does not know about parameter evaluation.

Therefor, we need an additional entity that a component's configuration according to parameter values. An important constraint on the mapper is that it is only allowed to change XML data.

XSLT(W3C 1999) would be a good choice for specifying parameter mappers, as it can be specified itself in XML and thereby preserves platform independence of component specifications. 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.

2.3 Composition

Component data objects may contain references to other component definitions and may define connections between them. The tool has to instantiate and parametrize sub components, to map parameters of all sub components, to check dependencies between components, and to connect sub component via their public ports.

Much of the difficulty for realizing compositions based solely on XML specifications of model components stems from the abandonment of programming languages. Today's general purpose programming languages like C++ and Java provide a comfortable type system and flexible instantiation mechanisms.

To instantiate sub component from declarative composition specifications a global entity is needed that resolves component identifiers to home instances: a home factory. Each component is granted access to this factory. Once instantiated, parameters are set on sub components, parameter mapping is initiated, and the public part of sub components is taken into consideration for consistency checks.

2.4 Transformation to Simulation Model

By specifying components and compositions we are not yet able to execute experiments. For the purpose of simulation a twofold relation between components and models shows up. Compositions have to be transformed to model elements of a concrete modeling formalism, which eventually can be executed and experimented with. Furthermore, a component's internal definition of model behavior has to be specified within a concrete modeling formalism. Our approach uses Parallel DEVS as the target formalism for both transforming component specifications and for specifying model behavior.

With respect to the transformation the target modeling formalism has to be at least expressive as the implementation formalism. Generally, DEVS and its variants provide a solid basis as targets for formalism transformations (Vangheluwe 2000). Actually, using Parallel DEVS as the target formalism transformation of component connections to model couplings becomes easy. Subcomponents map directly to sub models and each component connection maps to a set of couplings. Nevertheless, when transforming component structures to coupled Parallel DEVS models, the models must to have ports according to the public part of their component definition.

The actual transformation is divided into two stages. First, components are transformed to declarative models. In a second step the declarative models are transformed to executable ones that can be simulated by James II. Please refer to Röhl (2006) for the details of the first transformation step and to Röhl and Uhrmacher (2005) for the second.

Finally, the simulation tool James II (Uhrmacher 2006) can be used to execute the simulation model. JAMES II supports different modeling formalisms and different simulation engines (Himmelspach and Uhrmacher 2004). Simulation engines can be exchanged on demand and configured according to the specific requirements of a simulation model.

3 IMPLEMENTATION

The component framework was implemented as part of the simulation system James II. Figure 2 shows dependencies from the *component* package to the XML processing package *bind* and the model package *pdevs*.

HomeFactory forms the starting point for all compositions. It provides access to a *Home* instance for a unique component identifier. *Home* instances form the glue between XML data definitions and component logic implemented in *Component*. Each home relates to a certain component definition that has a unique identifier. A home can be used to create component instances, for which *Component* is the generic accessor. A component provides access to its underlying data definition *IComData*, on which mapping of parameters and the transformation of components into models will be done by an implementation of *IMapper*.

XmlFactory is responsible for creating bound entity according to an XML document. Bound entities encapsulate the transformation of string representations to Java objects and vice versa. Furthermore bound entities provide access to all elements and attributes of XML documents. Bound entities for component definitions were generated by the binding compiler of the JAXB framework as described in (Röhl and Uhrmacher 2005). The generated classes are integrated into the framework according to the *Adapter* pattern (Gamma et al. 1995). Thereby, generated implementations are hidden behind *IComData* and *IModelData*, which allow to access and manipulate XML data objects by convenient methods.

The class *PDevsBuilder* transforms component structures to Parallel DEVS models. Finally, *PDevsTransformer*

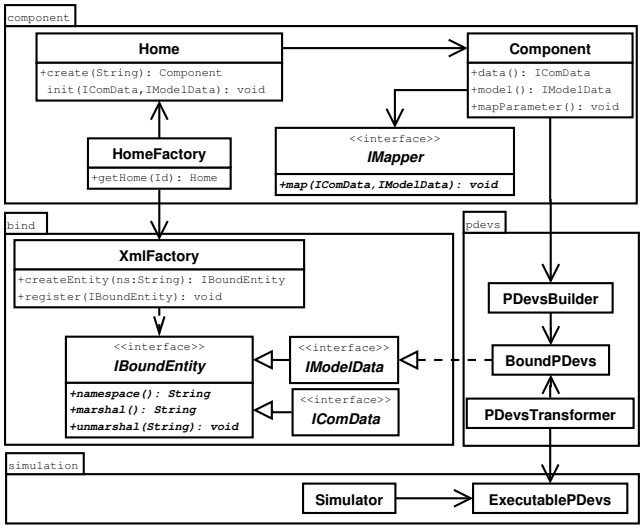

Figure 2: Basic Entities and their Relations

implements the transformation to an executable model according to Röhl and Uhrmacher (2005). Transformation ends with an instance of *ExecutableModel*, which can be simulated by James II (Uhrmacher 2006). Please note, that entities of package *simulation* know nothing about XML processing and components, i.e., they can be implemented and executed without additional overhead.

4 EXEMPLARY COMPOSITION OF A NETWORK SIMULATION MODEL

We will now illustrate the building of a simulation model from model components on a concrete example. Assume we want to build a model of a network consisting of a number of nodes, each connected to a component that represents the transport layer of the network. Figure 3 shows a UML 2.0 component diagram (OMG 2005) for the composition of a network from a transport component and a number of node components. The top level component *Network* should be customizable with respect to the number of nodes it contains.

4.1 Specification of a Network Component

We abstract from the lower layers of the OSI reference model and assume the nodes to exchange messages according to the 4th OSI layer (transport layer). Nodes expect the transport medium to accept sent messages and to deliver messages to nodes. This kind of interaction is expressed by the *Transport* interface:

```
<interface>
 <id name="Transport" version="1.0"/>
 <port name="send" type="unihro.osi.Message"
       isInput="1"/>
 <port name="receive"
       type="unihro.osi.Message"
       isInput="0"/>
</interface>
```

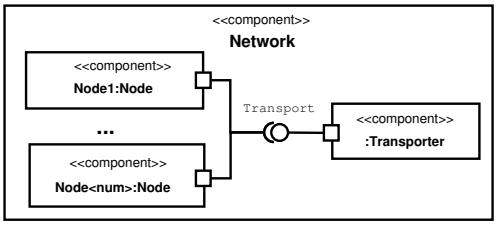

Figure 3: Composition of a Network

Within the network component a transporter component and node components become connected via their *transport* interfaces.

```
<component
 xmlns="http://www.inf.../component">
  <id name="Network" version="1"/>
  <param name="nodes" type="int"
         value="2"/>
  <mapper>unihro.com.net.v1.Mapper</mapper>
  <composition>
   <type name="Transporter" version="1"/>
   <name>trans</name>
  </composition>
  <composition>
   <type name="Node" version="1"/>
   <name>node1</name>
   <parameter name="id" value="1"/>
  </composition>
  <connection fromComponent="trans"
              fromPort="transport"
              toComponent="node1"
              toPort="transport"/>
</component>
```

For the Network component the internal details of the transporter and node component do not matter, e.g., they may be themselves composite components containing sub components. Actually, it contains a protocol component and a component that models user behavior. We omit the definition of the node component, which can be found in (Röhl 2006).

But, the transport component is required to have a port indicating the provision of the *transport* interface (facet). Conversely, each node component is expected to provide a port declaring the presence of the *transport* interface as a condition for proper functioning (receptacle).

The *Network* component takes a parameter: the number of nodes it is intended to contain. For evaluating this parameter, the network component is associated with a

class that does mapping of parameters according to the *IMapper* interface.

```
public class NetworkMapper implements IMapper {
 public final void map(IComData data,
                       IModeldata md) {
  String nodesStr =
    data.getParameter("nodes").value();
  int nodes = new Integer(nodesStr).intValue();
  data.getComposition("trans").
    setParameter("nodes", nodesStr);

  for (short id=2; id<=nodes; id++) {
   String nodeNumber = Integer.toString(id);
   String nodeName =  "node" + nodeNumber;

   IComposition nodeComp = data.addComposition(
      "Node", "1", nodeName);
   nodeComp.setParameter("id", nodeNumber);

   data.addConnection("trans", "transport",
     nodeName, "transport");
  } } }
```

The *NetworkMapper* starts with evaluating the parameter values. The parameter (*nodes*) is read and delegated to the sub component "trans". Furthermore, as many nodes as specified by the *node* parameter are added. Therefor, the data that represents the XML definition of the component is changed, i.e., a sub component of type "Node" version "1" is added in each loop cycle. In addition, it connects the transport facet of the transporter sub component to the according receptacle of each node.

The transporter component may be defined like:

```
<component
 xmlns="http://www.inf...de/mosi/cosa/component"
 id="Transporter" version="1">
  <param name="nodes" type="int" value="0"/>
  <mapper>unihro.com.trans.v1.Transporter
  </mapper>
  <port name="transport" isFacet="true">
    <type name="Transport" version="1.0"/>
  </port>
  <portmapping name="transport">
    <mapping declName="send"
             implName="fromNodes"/>
    <mapping declName="receive"
             implName="toNodes"/>
  </portmapping>
  <id="TransporterModel" version="1">
</component>
```

Port names of the model and the component declaration may differ. If this is the case, port mappings have to be specified. The transport facet requires the network model to have message ports named "send" and "receive". The port mapping assigns them to "fromNodes" and "toNodes" respectively. The transporter component refers to a model definition:

```
<model
 xmlns="http://www.inf...de/mosi/cosa/model/pdevs"
 xmlns:xsi=
   "http://www.w3.org/2001/XMLSchema-instance"
 xsi:type="CompiledModel"
 name="network">
 <inport type="unihro.base.Message">send</inport>
 <impl>unihro.com.net.v1.Transporter</impl>
</model>
```

The model definition itself refers to a Java class:

```
public class Transporter
  extends AtomicModel<TransporterState> {

  public void lambda() {
    TransporterState s = getState();
    Message toDeliver = s.queue.first();
    Integer rec = toDeliver.rec().toInt();
    getOutPort(rec.toString()).write(toDeliver); }

  public void deltaInternal() {
    final TransporterState s = getState();
    s.queue.pop();
    s.queue.timeOfFirstElementPassed(); }

  public void deltaExternal(double elapsedTime) {
    final TransporterState s = getState();
    s.queue.timePassed(elapsedTime);
    IPort fromNodes = getInPort("fromNodes");
    while(fromNode.hasValue()) {
      Message msg = (Message) fromNode.read();
      s.queue.insert(msg, routingT(8*msg.size()));
    } }

  public void deltaConfluent() {
    deltaInternal();
    deltaExternal(0.0); }

  public double timeAdvance() {
    TransporterState s = getState();
    return s.queue.nextTime(); }

  private double routingT(int msgSize) { ... }
}
```

4.2 Instantiation of the Simulation Model

Figure 4 illustrates the instantiation of component from XML documents.

A *Launcher* requests the component specification with the name "Network" version "1" from the *HomeFactory*. If a home with this id was not already instantiated, XML data has to be retrieved from a database as a string object. *XmlFactory* creates bound entities according to the namespaces of the XML data. Subsequently, the XML data is unmarshalled by *IComponentData* and *BoundPDevs* respectively. A new home is created and initialized with the component's XML data and its model definition. This home is now used by the Launcher to create a component instance with the name "net". The home clones XML data for the new component instance. This ensures that setting of parameters will not take affect on other component instances of the same type.

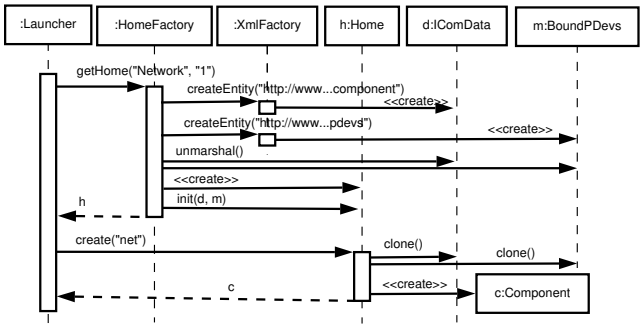

Figure 4: Instantiation of the Network Component

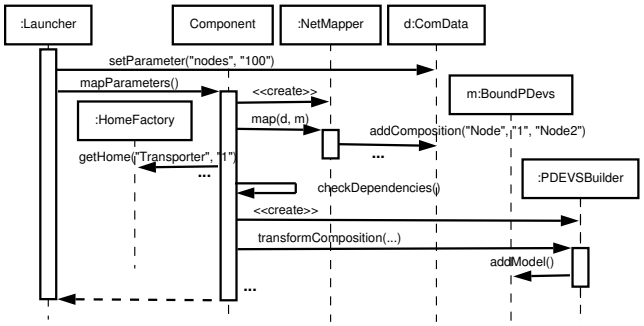

Figure 5: Transformation of the Component to a Simulation Model

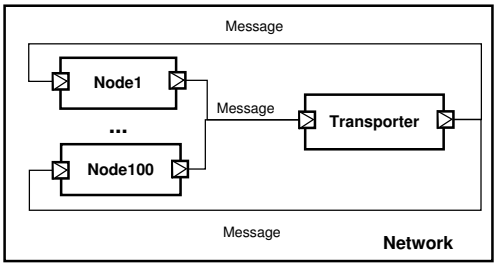

Figure 6: Target Simulation Model

Figure 5 shows the construction of a simulation model from the net component. First, parameters according to an experiment definition are set, e.g., our network component is parametrized to contain 100 nodes. When the function *mapParameters()* becomes called, the *NetworkMapper*, which is referenced in the XML document, becomes instantiated and changes the underlying XML data of the component. Afterwards, all sub components of the network are instantiated, i.e., the transporter component and each of the 100 node components. Subsequently, parameters of sub components are set and parameter mapping of them initiated. Finally, it is checked whether the receptacles of all sub component instances are connected to according facets.

The composition is now complete and we are free to transform it to a simulation model. Therefore, a model builder is created that transforms all component constructs to elements of the Parallel DEVS formalism, see Röhl (2006) for details. Finally, the declarative representation of the "pure" Parallel DEVS model is transformed to a representation that can be executed by `james.core`. The realization of transformer classes was already described in (Röhl and Uhrmacher 2005).

Figure 6 shows the outcome of the composition and transformation process for the network component with parameter *nodes* set to 100.

5 RELATED WORK

To be composable by third parties the requirements and provisions of components have to explicitly specified in an indirect manner by references to interfaces (Szyperski 2002). With respect to this, the presented framework is basically in line with software engineering approaches to composition. The implementation of a component is hidden for others and is not relevant for consistency checks. The syntax for defining required and provided interfaces, for specifying compositions, for defining connections, and for using attributes to configure component instance is similar to the UML 2.0 (OMG 2005) and the Corba Component Model (OMG 2002).

Software component approaches like the Corba Component Model (OMG 2002) and Enterprise Java Beans (Sun Microsystems 2003) also use declarative specifications to enrich programming language implementations. For the CCM the Interface Definition Language is used and for EJB Java Annotations are employed. We developed a small but expressive XML Schema Definition for simulation model components that defines a component syntax tailored for simulation purposes (Röhl 2006).

An important difference between our approach and software components is that the latter have to be deployed as binary units (Szyperski 2002). Software components are usually directly executable and to be treated as black boxes. The limited number of binary platforms allows software components to be packaged with executable implementations for all relevant platforms. To package simulation model components as executables is not really sensible, as a binary standard for simulation systems does not exist. To compose binary representations of models their simulation engines have to be plugged together. This is the realm of low level interoperability protocols HLA (IEEE 2000), which support the interoperability of entire simulation systems.

Modeling and simulation approaches that incorporate a declarative specification layer like the SES/MB framework (Zeigler and Sarjoughian 2002) do not currently account for the explicit specification of provided and required interfaces.

The presented framework builds heavily on XML descriptions and defers the generation of tool-specific executables as much as possible. The outcome of the composition process is a pure model according to the Parallel DEVS formalism. This is different from most existing approaches that weave composition and modeling facilities (Chen and Szymanski 2002; MacSween and Wainer 2004; Lang, Jacobs, and Verbraeck 2003).

Our approach is close to Gustavson and Chase (2004) in exploiting the flexibility of XML together with XSD and striving for platform independent specifications. Both approaches suggest to transform platform independent XML representations into platform specific models. While BOMs focus on HLA compliance, the approach presented here puts a strong emphasize on interface definitions close to UML.

6 SUMMARY

We propose a strictly layered approach to integrate component facilities into modeling and simulation systems. Compositions are realized as an additional processing layer on top of modeling formalisms and a concrete execution engine of a simulation system. XML forms the basis for representing interfaces, components, and compositions. Data binding is used to execute transformations between XML documents, each of which is bound to a certain XML Schema Definition, and object instances, which can be used by the component layer. However, component instances exist only temporarily until the generation of the simulation model is completed.

To separate between a component's storage format and its executable representation has two benefits. First, the feature to compose models can be added to existing tools without modifying the latter. Second, component definitions do not put an overhead to the execution efficiency of simulation models. However, the simulation engine must support a modeling formalism that is sufficiently expressive to be generated from UML-like component definitions, e.g., as in our case Parallel DEVS.

The presented framework was implemented as part of the simulation system James II (Uhrmacher 2006). It is currently used for evaluating overlay protocols for mobile ad-hoc networks, which requires the combination of different user and protocol components. Future work aims at integrating semantically stronger concepts for checking consistency of compositions, e.g., on base of application domain specific meta data (Uhrmacher et al. 2005).

ACKNOWLEDGMENTS

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

REFERENCES

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

Chen, G., and B. K. Szymanski. 2002. COST: a component-oriented discrete event simulator. In *Proceedings of the 2002 Winter Simulation Conference*, 776–782.

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

Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1995. *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley.

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

Himmelspach, J., and A. M. Uhrmacher. 2004. A component-based simulation layer for James. In *Proceedings of the Eighteenth Workshop on Parallel and Distributed Simulation (PADS '04)*, 115–122. Kufstein, Austria: IEEE Computer Society Press.

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

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

Lang, N. A., P. H. Jacobs, and A. Verbraeck. 2003, October. Distributed open simulation model development with DSOL services. In *ESS'2003, Proceedings 15th European Simulation Symposium 2003 - Simulation in Industry*, 210–218. Delft.

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

OMG. 2002, June. CORBA Components version 3.0 (document formal/02-06-65). Available online via <www.omg.org/cgi-bin/doc?formal/02-06-65> [accessed July 13, 2006].

OMG. 2005, July. UML superstructure specification version 2.0 (document formal/05-07-04). Available online via <www.omg.org/cgi-bin/doc?formal/05-07-04> [accessed July 13, 2006].

Petty, M. D., E. W. Weisel, and R. R. Mielke. 2005. Composability theory overview and update. In *Proceedings of the Spring 2005 Simulation Interoperability Workshop*, 431–437.

Röhl, M. 2006. Platform independent specification of simulation model components. In *ECMS 2006*, 220–225.

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

Sun Microsystems 2003, November. Enterprise JavaBeans Specification, version 2.1. Available online via <java.sun.com/products/ejb/docs.html> [accessed July 13, 2006].

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

Tolk, A., and J. Muguira. 2003, September. The levels of conceptual interoperability model. In *Fall Simulation Interoperability Workshop (SISO), Orlando*.

Uhrmacher, A. M. 2006. James II project description. Available online via <wwwmosi.informatik.uni-rostock.de/mosi/projects/cosa/james-ii> [accessed July 13, 2006].

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

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

W3C 1999, November. XSL transformations (XSLT) version 1.0. Available online via <www.w3.org/TR/xslt> [accessed July 13, 2006].

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

Zeigler, B. P., H. Praehofer, and T. G. Kim. 2000. *Theory of modeling and simulation*. 2nd ed. London: Academic Press.

Zeigler, B. P., and H. S. Sarjoughian. 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.

AUTHOR BIOGRAPHIES

MATHIAS RÖHL holds a MSc in Computer Science from the University of Rostock. His research interests are on component-based modeling and agent-oriented simulation. He is currently a research scientist at the Modeling and Simulation Group at the University of Rostock. His e-mail address is <mroehl@informatik.uni-rostock.de> and his Web address is <www.informatik.uni-rostock.de/~mroehl>.

ADELINDE M. UHRMACHER is an Associate Professor at the Department of Computer Science at the University of Rostock and head of the Modeling and Simulation Group. Her research interests are in modeling and simulation methodologies, particularly agent-oriented modeling and simulation and their applications. Her e-mail address is <lin@informatik.uni-rostock.de> and her Web page is <www.informatik.uni-rostock.de/~lin>.