Proceedings of the 2008 Winter Simulation Conference
S. J. Mason, R. R. Hill, L. Mönch, O. Rose, T. Jefferson, J. W. Fowler eds.

DEFINITION AND ANALYSIS OF COMPOSITION STRUCTURES FOR DISCRETE-EVENT MODELS

Mathias Röhl
Adelinde M. Uhrmacher

University of Rostock
Albert-Einstein-Str. 21,
Rostock, 18059, Germany

ABSTRACT

The re-use of a model by someone else than the original developer is still an open challenge. This paper presents composition structures and interface descriptions for discrete-event models. Interfaces are introduced as separate units of description that complement model definitions. As XML documents, interfaces may be stored in databases to search, select, and analyze composition candidates based on public visible property descriptions. A meta model formalizes interfaces, components, and compositions, such that the refinement of interfaces into model implementations and the compatibility of interfaces can be analyzed. The composition approach combines different hierarchical relations (type hierarchies, refinement hierarchies, and composition hierarchies) to simplify the modeling process.

1 INTRODUCTION

To become a component, a model needs to announce its functionality by a well-defined interface (Verbraeck 2004). A component should be a replaceable part of a system and be usable in unforeseen contexts for different purposes (Szyperski 2002). Interfaces should contain as much information, but to keep it analyzable not more, as is needed to use an implementation solely via its interface (de Alfaro and Henzinger 2005).

The main challenge for compositional approaches is summarized in the term compositionality, which requires that the meaning of a composition can be derived solely from the semantic descriptions of the parts together with the rules of combination (Janssen 1997). Parts have compositional properties if the semantics of a composition may be derived from the semantics of the parts (Szyperski 2002).

Modeling formalisms like DEVS (Zeigler et al. 2000) unambiguously define how model descriptions are to be interpreted. Furthermore, they usually provide modular-hierarchical refinement relations. However, discrete-event modeling currently does not treat interface descriptions as first class entities: they are part of model definitions and not separate units of definition. To enable compositional reasoning, relevant properties have to be extracted into interfaces, which exist in their own right and may be published and analyzed independently. Currently no general interface language for discrete-event models exist that can be used for compositional analysis.

The Unified Modeling Language (UML) was revised in version 2 to describe components and compositions (OMG 2007b). The Systems Modeling Language (SysML) extends UML with event-based port concepts (OMG 2007a). SysML distinguishes between diagrams that define the external interface (external block diagram) of a model and its internal structure (internal block diagram).

Whereas SysML enables flexible description of composition structures, its usage for simulation is hampered by its only partly defined semantics. SysML inherits, as an UML profile, its semantic backbone from UML. Semantics of UML is to a large extent defined in natural English. Models are not associated with a precise meaning (Harel and Rumpe 2004, Oliver and Luukkala 2006). Formalizing UML is seen as one of the major challenges for the future (Broy et al. 2006, O'Keefe 2006).

In the following, composition concepts of UML and SysML are adapted to discrete-event modeling. A general interface and composition language will be defined and equipped with formal semantics. Configurable model components may be assembled to composition structures, on which compositional reasoning can be carried out.

2 INTERFACES

For reusing a model all interaction capabilities have to be described unambiguously. Modular-hierarchical modeling approaches like Modelica (Elmqvist et al. 2001), DEVS (Zeigler et al. 2000), and Ptolemy (Brooks et al. 2007) let models exhibit ports to indicate that certain kinds of events may be received or sent. The first step for introducing interfaces as separate units of definition is to increase the flexibility for defining and relating the units for data exchange (events). Compared to basic set theory, type systems (Pierce 2002) enable the modularization of definition units.

Definition 1. *A **type** is a pair $\mathscr{T} = (id, Car)$, with $id \in QName$ and a carrier set Car. To access the elements of $\mathscr{T}$, $id_{\mathscr{T}}$ and $car_{\mathscr{T}}$ are used.*

$QName \subseteq String$ is used as the set of qualified names, with $String$ being the set of all character strings. A qualified name $qname \in QName$ may be represented by combining a namespace and a local name, separated by ':'.

Type is used as the set of types that are uniquely distinguishable via their qualified names, i.e. $\forall \mathscr{T}, \mathscr{T}' \in Type.id_{\mathscr{T}} = id_{\mathscr{T}'} \Rightarrow \mathscr{T} = \mathscr{T}'$. Given the special symbol $\perp$, which represents "undefined", the function $drefT : QName \rightarrow Type \cup \{\perp\}$ is used to de-reference a type definition for its qualified name, such that

$$drefT(qname) \overset{def}{=} \begin{cases} \mathscr{T} & \text{if } \exists \mathscr{T} \in Type.id_{\mathscr{T}} = qname \\ \perp & \text{else} \end{cases}.$$

Instead of local set definitions to define abstract points of interaction, an event port can now be based on references to types. This increases the flexibility for defining and relating basic modeling units.

Definition 2. *An **event port** is a tuple $e = (name, tid, inp)$, with $name \in String$, a reference to a type definition $tid \in QName$, and a flow direction $inp \in \mathbb{B}$, with $\mathbb{B} = \{true, false\}$. $name_e$, tid_e and inp_e are used to access the elements of e.*

If $inp_e = true$, e is called an input port and if $inp_e = false$, e is an output port. Type systems usually require for a sending port to be a sub type of the connected receiving port (Pierce 2002). $\mathscr{T}$ is a subtype of $\mathscr{T}'$, denoted $\mathscr{T} \sqsubseteq \mathscr{T}'$, if $Car_{\mathscr{T}} \subseteq Car_{\mathscr{T}'}$.

An UML component usually exhibits a set of interfaces and requires another component to provide according counterparts (OMG 2007b). In addition to method-oriented standard ports of UML, SysML introduces *flow ports*, which declare asynchronous communication end points. Similar to *flow specifications* of SysML, event-based interaction capabilities are now grouped in roles.

Definition 3. *A **role** $\mathscr{R}$ is a pair (id, EP) with a qualified name $id \in QName$ and a finite set of event ports EP, such that port names are unique and type references valid, i.e. $\forall e, e' \in EP.name_e = name_{e'} \Rightarrow e = e'$ and $\forall e \in EP.drefT(tid_e) \neq \perp$. The elements of $\mathscr{R}$ may be accessed via $id_{\mathscr{R}}$ and $EP_{\mathscr{R}}$.*

Roles represent interface information with respect to a certain abstraction by declaring a set of directed event ports, which have a logical relation. The function $drefR(qname)$ returns a role $\mathscr{R}$ if $\exists \mathscr{R} \in Role.id_{\mathscr{R}} = qname$ or $\perp$ if no according role exists, where *Role* is used as the set of all roles.

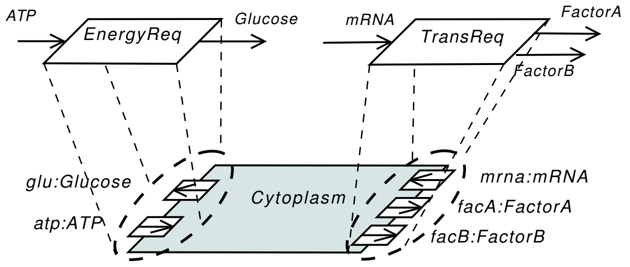


Figure 1: Grouping atomic interaction capabilities in roles

Example 1. *A biological cell comprises different parts, e.g. the nucleus and mitochondria (the "power plant" of cells). A model of a cell may represent the nucleus and mitochondria as sub models and pool all other parts in a third sub model named "cytoplasm". Figure 1 shows how the interaction capability of the cytoplasm may be captured by two roles.* `EnergyReq` *describes the capability to send* `Glucose` *and receive* `ATP`*. According to* `TransReq` *the cytoplasm may receive* `mRNA` *and send events of type* `FactorA` *and* `FactorB`*. Formally,* `EnergyReq` $= (id, EP)$*, with $id =$ "base:EnergyReq" and $EP = \{($"atp", "mol:ATP", $true)$, $($"gly", "mol:Glucose", $false)$.$\}$, where mol= "unihro/cbio/-molecules" and base= "unihro/cbio/base".*

Interfaces are used to combine a set of roles, each one describing a certain aspect of a model's overall communication potential. Whereas event-ports announce "atomic" interaction capabilities, roles refer to complex ones. Using composite ports, an interface declares that it may be coupled several times in the same manner with different models according to a certain role.

Definition 4. *A **composite port** is a tuple $p = (name, rid, min, max)$, with a $name \in String$, a reference to a role $rid \in QName$, and a connectivity range given by $min \in \mathbb{N}^+$ and $max \in \mathbb{N}^+ \cup \{\infty\}$. The reference to a role must be valid $drefR(rid) \neq \perp$ and $min \leq max$. We write $name_p$, rid_p, min_p, and max_p to refer to the elements of p.*

A model component needs to be customizable, such that a component represents a whole class of models, which are suited for a set of similar composition contexts.

Definition 5. *A **parameter** is a triple $p = (name, tid, value)$, with a $name \in String$, a reference to a type $tid \in QName$, and an value. For $\mathscr{T} = drefT(tid)$ it must hold that $\mathscr{T} \neq \perp \wedge value \in car_{\mathscr{T}}$.*

Based on parameters and composite ports, interfaces can now be defined to declare configuration points and abstract points of interaction.

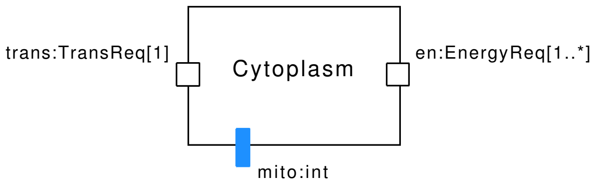


Figure 2: Interface with two composite ports and a parameter

Definition 6. *An **interface** $\mathscr{I}$ is a tuple $(id, impl, Params, Ports)$, with an $id \in QName$, a reference to a component (cf. Definition 13) $impl \in QName$, a finite set of parameters Params, and a finite set of ports Ports. Port and parameter names must be unique, i.e. $\forall p, p' \in Ports.name_p = name_{p'} \Rightarrow p = p'$ and $\forall p, p' \in Params.name_p = name_{p'} \Rightarrow p = p'$. For accessing the elements of $\mathscr{I}$, $id_{\mathscr{I}}$, $impl_{\mathscr{I}}$, $Params_{\mathscr{I}}$, and $Ports_{\mathscr{I}}$ may be used.*

Iface is used as the set of interfaces with unique ids and $drefI(qname) = \mathscr{I}$ if $\exists \mathscr{I} \in Iface.id_{\mathscr{I}} = qname$ or $\perp$ if no according interface exists.

Example 2. *An interface definition for `Cytoplasm` is depicted in Figure 2. The interface exhibits two composite ports. Port "trans" is typed by role `TransReq` and port "en" and is of type `EnergyReq`. Whereas "trans" has to be connected exactly once, the number of connections to port "en" may range from one up to an arbitrary number (indicated by *). Parameter "mito" allows to configure a cytoplasm to the number of mitochondria it should be connected to. Formally, the interface is defined as $\mathscr{I} = (id, impl, Params, Ports)$, with id = "cyto:iface", impl = "cyto:impl", Params = $\{($"mito, "xsd:int"$, 1)\}$, and Ports = $\{($"trans", "base:TransReq"$, 1, 1)$, $($"en", "base:EnergyReq"$, 1, \infty)\}$. The namespace base is defined as above and cyto= "unihro/cbio/cyto".*

3 COMPOSITION

A composition is now defined such that it does not need to make assumptions about the implementation of components but solely refers to interfaces as introduced above. To use an interface as part of a composition, a qualified name of the interface has to be known and a concrete set of parameter values has to be provided.

Definition 7. *An **interface reference** is a tuple $iref = (name, iid, Params)$ with a $name \in String$, a reference to an interface definition $iid \in QName$, and a finite set of parameters Params. For $\mathscr{I} = drefI(iid)$ it must hold: (i) existing interface definition: $\mathscr{I} \neq \perp$, (ii) unique parameter names: $\forall p, p' \in Params.name_p = name_{p'} \Rightarrow p = p'$, and (iii) existent and consistent parameter $\forall p \in Params.\exists p' \in Params_{\mathscr{I}}.name_p = name_{p'} \wedge tid_p \sqsubseteq tid_{p'}$.*

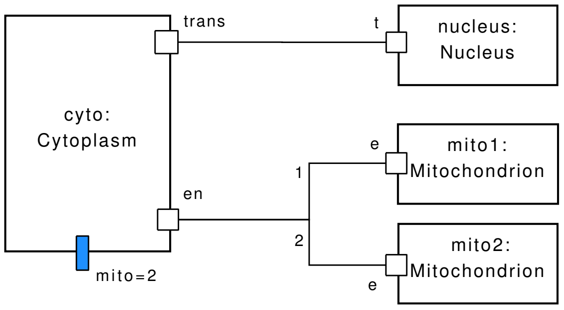


Figure 3: Composite structure of a cell

An interface may be connected via its published ports.

Definition 8. *A **connector** is a tuple $k = (if, port, pos)$, where $if \in String$ is the name of an interface, $port \in String$ is the name of a port, and $pos \in \mathbb{N}^+$ is a position.*

To equip connectors with a precise meaning, each connector needs to define to which position, in the range of multiplicities of a port, it refers to. A connection is made up of two connectors representing the start and the end point.

Definition 9. *A **composition connection** c is a pair of two connectors $(start, end)$, for which $if_{start} \neq if_{end}$.*

In compositions, all communication between components have to be expressed via composition connections, such that the knowledge of each sub component ends at its borders.

Definition 10. *A **composition** is a pair (Sub, Con), with a finite set of interface references Sub and a finite set of composition connections Con. Interface references must have unique names: $\forall s, s' \in Sub.name_s = name_{s'} \Rightarrow s = s'$.*

Example 3. *Figure 3 shows the composition and connection of `Cytoplasm`, `Nucleus`, and `Mitchondrion`. `Cytoplasm` and `Nucleus` are connected via their ports "trans" and "t". Both `Mitochondrion` instances are connected to the cytoplasm at a distinct position within the multiplicity of the port "en". The cytoplasm's parameter "mito" is set to 2. Formally, the composition is defined by Sub = $\{($"cyto", "cyt:iface", $($"mito", "xsd:int"$, 2))$, $($"nucleus", "nuc:iface"$, \emptyset)$, $($"mito1", "mit:iface"$, \emptyset)$, $($"mito2", "mit:iface"$, \emptyset)\}$ and Con = $\{(($"cyto", "trans"$, 1)$, $($"nucleus", "t"$, 1))$, $(($"cyto", "en"$, 1)$, $($"mito1", "e"$, 1))$, $(($"cyto", "en"$, 2)$, $($"mito2", "e"$, 1))\}$. with "nuc", "mit", and "cyto" beeing abbreviations for according namespaces.*

4 IMPLEMENTATIONS

Interfaces serve as contracts between model components. By referencing roles, an interface declares a set of directed event ports. An implementation is expected to provide ports that realize these interaction capabilities. Furthermore, the model is not allowed to circumvent its interface declaration, i.e. it must not have direct dependencies to other model implementations.

Declared interaction capabilities need to be unambiguously associated with concrete model ports. Explicit bindings resolve name clashes, which might occur if a port may be connected multiple times.

Definition 11. *A **binding** is a finite tuple set $\mathscr{B} = \{(port, decl, pos, impl)\}$, with $port \in String$ being the name of a composite port, $decl \in String$ the name of an event port, $pos \in \mathbb{N}^+$ a position, and $impl \in String$ the name of the assigned model port. Bindings must be unique, i.e. $\forall b, b' \in \mathscr{B}. b = b' \Leftrightarrow port_b = port_{b'} \wedge decl_b = decl_{b'} \wedge pos_b = pos_{b'}$.*

The meaning of bindings is given by the function *impl*, which takes the name of a composite port *pn*, the name of an event port *epn*, a position *pos*, and a binding $\mathscr{B}$ and provides the name of an implementation port, such that $impl(pn, epn, pos, \mathscr{B}) \overset{def}{=}$

$$\begin{cases} impl_b \text{ if } \exists b \in \mathscr{B}. port_b = pn \wedge decl_b = epn \wedge pos_b = pos \\ epn \text{ else.} \end{cases}$$

Example 4. *In Figure 4 the atomic ports of role `TransReq` are bound to a separate port of a model implementation (bindings are visualized with bold dashed lines). The two different positions of the event port `Glucose` of the composite port "en" are bound to different output ports of the implementation model, such that each mitochondrion can be addressed individually by the cytoplasm. As it is not relevant for the cytoplasm model from whom ATP is received, both `ATP` channels are bound to the same implementation port.*

An interface may declare a set of parameters. To change the internal structure of a component, according to a set of parameter values, is the task of a configurator function.

Definition 12. *A **configurator** κ is a function that maps a tuple $(Params, Sub, Con, \mathscr{M}, \mathscr{B})$ to a tuple $(Sub', Con', \mathscr{B}', \mathscr{M}')$, where Params is a set of parameters. (Sub, Con) and (Sub', Con') are both compositions. $\mathscr{M}, \mathscr{M}'$ are model definitions and $\mathscr{B}, \mathscr{B}'$ are bindings.*

A component comprises a configurator, bindings, and a composition. Thus, components may itself refer to other components and thereby facilitate hierarchical modeling. The last missing piece of a component becomes the part, which was the reason for introducing components in the first place: a model definition specified in a certain modeling formalism.

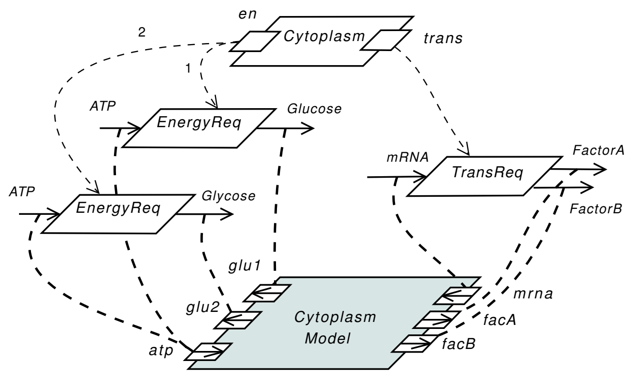


Figure 4: Binding of declared ports to an implementation

Definition 13. *A **component** is a tuple $\mathscr{C} = (id, if, \kappa, \mathscr{M}, \mathscr{B}, Sub, Con)$, with an $id \in QName$, an interface reference $if \in QName$, a configurator κ, a model $\mathscr{M}$, and a binding $\mathscr{B}$. (Sub, Con) is a composition. It must hold that $drefI(if) \neq \perp$ and $id = impl_{\mathscr{I}}$, where $\mathscr{I} = drefI(if)$.*

If $Sub_{\mathscr{C}} = \emptyset$, $\mathscr{C}$ is called an atomic component. If $Sub_{\mathscr{C}} \neq \emptyset$, $\mathscr{C}$ is called a composite component. A component $\mathscr{C}$ is instantiated by applying its configurator to the parameters defined by an interface reference $s = (name, iid, Params)$, such that $instC(s) \overset{def}{=} (id_{\mathscr{C}}, if_{\mathscr{C}}, \perp, \mathscr{M}', \mathscr{B}', Sub', Con')$ if $\exists \mathscr{C} \in Com. id_{\mathscr{C}} = impl_{\mathscr{I}}$, where $(Sub', Con', \mathscr{M}', \mathscr{B}') = \kappa_{\mathscr{C}}(Params_s, Sub_{\mathscr{C}}, Con_{\mathscr{C}}, \mathscr{M}_{\mathscr{C}}, \mathscr{B}_{\mathscr{C}})$ and $\mathscr{I} = drefI(iid_s)$.

Example 5. *A cell is now defined as a (composite) component. The interface of `Cell` provides two parameters and is equipped with a composite port to interact with other cells. The latter requires the definition of a new role and a different cytoplasm component, as the cytoplasm needs to interact according to the new role. Figure 5 depicts the hierarchical relation between the interface and the implementation of a cell. The parameter "mito" prescribes the number of mitochondrion components to use inside the cell and the parameter "nex" determines the number of connections to other cell components. κ is responsible for adding mitochondria instances and connections according to parameter "mito".*

5 SEMANTICS

Whereas the syntactical means for specifying compositions are not constrained to a particular modeling formalism, now the formalism PDEVS (Zeigler et al. 2000) is used as the semantic domain on which components are mapped.

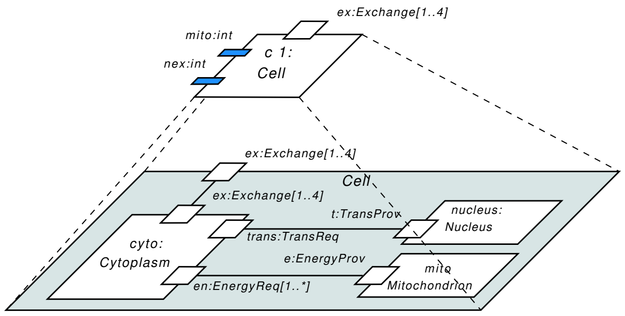


Figure 5: Interface and implementation of a cell component

Definition 14. *Let $\mathscr{C}$ be a component defined by $(id, if, \kappa, \mathscr{M}, \mathscr{B}, Sub, Con)$. The PDEVS model for $\mathscr{C}$ is constructed by*

$$model(\mathscr{C}) = \begin{cases} \mathscr{M}' & if\, Sub = \emptyset \\ (X, Y, D, \{M_d\}, EIC, IC, EOC) & if\, Sub \neq \emptyset \end{cases},$$

where $\mathscr{M}' = pdevs(\mathscr{M})$, $X = X_{\mathscr{M}'}$, $Y = Y_{\mathscr{M}'}$, $D = \bigcup_{s \in Sub} name_s$,

$\{M_d\} = \bigcup_{s \in Sub} M_{name_s}$, *with* $M_{name_s} = model(\mathscr{C}_{name_s})$, $\mathscr{C}_{name_s} = instC(s)$,

$EIC = \{(fm, fp, tm, tp) \in couplings(Sub', Con) | fm = \text{“this”} \wedge tm \neq \text{“this”}\}$,

$EOC = \{(fm, fp, tm, tp) \in couplings(Sub', Con) | fm \neq \text{“this”} \wedge tm = \text{“this”}\}$,

$IC = \{(fm, fp, tm, tp) \in couplings(Sub', Con) | fm \neq \text{“this”} \wedge tm \neq \text{“this”}\}$,

with $Sub' = Sub \cup \{(\text{“this”}, if, \emptyset)\}$.

The function *model* recursively maps sub components to sub models and composition connections to model couplings. To constrain the semantic mapping to *one* target formalism, the universality of PDEVS (Zeigler et al. 2000, pp. 391f) is exploited. PDEVS is suited to work as a target formalism for many source formalisms (Vangheluwe 2000). Here, we abstract from the concrete realization of model transformation and assume that a function *pdevs* exists, which transforms models to PDEVS.

With respect to composition structures, the mapping of composition connections to model couplings is done by another help function, called *couplings*. Algorithm 1 lists its definition. Composition connections are interpreted to relate event ports of referenced roles. For each connection between two composite ports *couplings()* instantiates the roles that type both ports. According to the hierarchical relation between the two connected subcomponents (whether between a component and one of its sub components or between two sub components) the atomic port declarations contained within both roles are matched against each other. The resulting set of port name pairs

Algorithm 1 Construction of model couplings

name: *couplings()*
input: interface references *Sub*, connections *Con*
output: set of model couplings *MC*

```
$MC = \emptyset$
for each $(start, end) \in Con$ do
  // dereference interfaces and roles according to connectors
  $\mathscr{I} = drefI(iid_s)$ with $s \in Sub \wedge name_s = if_{start}$
  $\mathscr{I}' = drefI(iid_{s'})$ with $s' \in Sub \wedge name_{s'} = if_{end}$
  $\mathscr{R} = drefR(rid_p)$ with $p \in Ports_{\mathscr{I}} \wedge name_p = port_{start}$
      $\wedge min_p \leq pos_{start} \leq max_p$
  $\mathscr{R}' = drefR(rid_{p'})$ with $p' \in Ports_{\mathscr{I}'} \wedge name_{p'} = port_{end}$
      $\wedge min_{p'} \leq pos_{end} \leq max_{p'}$
  // match event ports of connected roles
  if $if_{start} = \text{“this”}$ then $M = matches(\overline{\mathscr{R}}, \mathscr{R}')$
  else if $if_{end} = \text{“this”}$ then $M = matches(\mathscr{R}, \overline{\mathscr{R}'})$
  else $M = matches(\mathscr{R}, \mathscr{R}')$
  for each $(f, t) \in M$ do
    // apply bindings to get ports of the implementation
    $ip = impl(port_{start}, f, pos_{start}, \mathscr{B})$
    $ip' = impl(port_{end}, t, pos_{end}, \mathscr{B}')$
    // get first event port and compare its direction to the
        orientation of the composition connection
    $e = dp$, with $dp \in EP_{\mathscr{R}}.name_{dp} = f$
    if $if_{start} = \text{“this”} \wedge inp_e$ or $if_{start} \neq \text{“this”} \wedge \neg inp_e$
      then $MC += (if_{start}, ip, if_{end}, ip')$ // keep orientation
    else if $if_{start} \neq \text{“this”} \wedge inp_e$ or $if_{start} = \text{“this”} \wedge \neg inp_e$
      then $MC += (if_{end}, ip', if_{start}, ip)$ // invert orientation
    else $MC += \perp$
return $MC$
```

is called $matches(\mathscr{R}, \mathscr{R}')$, with $matches(\mathscr{R}, \mathscr{R}') \overset{def}{=} \{(f, t) \in String \times String | \exists e \in EP_{\mathscr{R}}, e' \in EP_{\mathscr{R}'}.name_e = f \wedge name_{e'} = t \wedge inp_e \neq inp_{e'} \wedge (inp_e \Rightarrow drefT(tid_{e'}) \sqsubseteq drefT(tid_e)) \wedge (inp_{e'} \Rightarrow drefT(tid_e) \sqsubseteq drefT(tid_{e'}))\}$. In case of a vertical connection (connection between component and one of its sub component) the directions of event ports are inverted for matching purposes. Formally, $\overline{\mathscr{R}} \overset{def}{=} (\perp, EP_{\overline{\mathscr{R}}})$, with $EP_{\overline{\mathscr{R}}} = \{(name_e, tid_e, \neg inp_e) | (name_e, tid_e, inp_e) \in EP_{\mathscr{R}}\}$. Finally, for each pair of matching event ports – which may be seen as atomic connections at the declaration level – bindings are applied to get the implementation ports to be connected.

Example 6. *Figure 6 shows the model structure derived from the cell component (cf. Figure 5) with parameters set to “mito”=10 and “nex”=1. A sample model definition is depicted in Figure 7. The mitochondrion is modeled as a statechart. Function pdevs interprets events receivable by the statechart as input ports and emit-able events as output ports.*

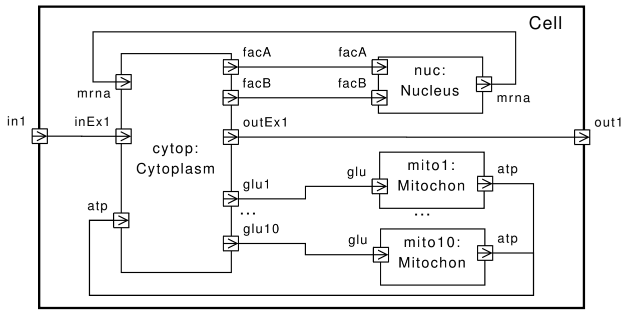


Figure 6: Derived model structure

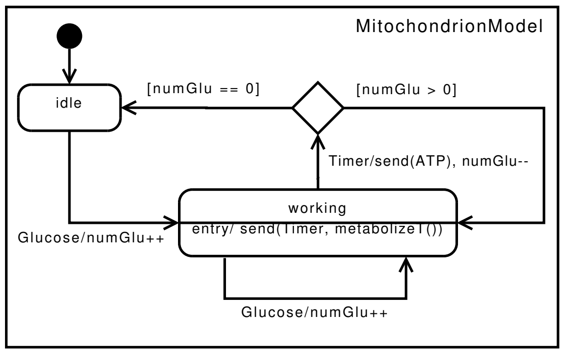


Figure 7: Mitochondrion modeled as a statechart

6 ANALYSIS

An important motivation for compositional approaches roots in the promise to save time and costs by ensuring correctness by construction. Properties of a composition should be derivable from property descriptions of the parts and the rules to combine them. In the following, requirements on a component $\mathscr{C}$ are formulated, such that $model(\mathscr{C})$ constructs a proper simulation model.

To decide whether components can be connected, one has to compare the types of events that are declared by their interfaces. Only if these types are sufficiently similar, models may be composed. Coupled system specifications require the set of events that a source port may sent to form a subset of the events that the target port is able to receive (Zeigler et al. 2000, p. 130). This directly corresponds to the sub type relation $\sqsubseteq$.

Definition 15. *Two roles $\mathscr{R}$ and $\mathscr{R}'$ are called* ***compatible****, denoted $\mathscr{R} \sim \mathscr{R}'$, iff $\forall e \in EP_{\mathscr{R}}.\exists e' \in EP_{\mathscr{R}'}.compatible(e,e')$ and $\forall e' \in EP_{\mathscr{R}'}.\exists e \in EP_{\mathscr{R}}.compatible(e',e)$, where for two event ports e,e', $compatible(e,e') \Leftrightarrow (inp_e \Rightarrow \neg inp_{e'} \wedge dref(tid_e) \sqsupseteq dref(tid_{e'})) \wedge (\neg inp_e \Rightarrow inp_{e'} \wedge dref(tid_e) \sqsubseteq dref(tid_{e'}))$.*

Connections fall into two different classes. On the one hand, it is possible to connect ports of sub components. On the other hand, connections may delegate ports between a component and one of its sub components.

Definition 16. *A composition connection $c = (start, end)$ is* ***well-formed*** *for two interface references s, s', denoted $wellformed(c,s,s')$, if for $\mathscr{R} = role(drefI(iid_s), start)$ and $\mathscr{R}' = role(drefI(iid_{s'}), end)$ it holds that $\mathscr{R} \neq \perp \wedge \mathscr{R}' \neq \perp$ and if$_{start}$ = "this" $\Rightarrow \overline{\mathscr{R}} \sim \mathscr{R}'$, if$_{end}$ = "this" $\Rightarrow \mathscr{R} \sim \overline{\mathscr{R}'}$, and $assembly(c) \Rightarrow \mathscr{R} \sim \mathscr{R}'$; where $role(\mathscr{I},k) \overset{def}{=} drefR(rid_p)$ if $\exists p \in Ports_{\mathscr{I}}.name_p = port_k \wedge min_p \leq pos_k \leq max_p$ or $\perp$ if not and $assembly((start,end)) \overset{def}{=} name_{start} \neq$ "this" $\wedge\, name_{end} \neq$ "this".*

Composition connections demand compatibility of connected port's roles. Compositionality requires from the parts to be combined to adhere to certain rules too: implementations have to refine their respective interfaces. Refinement depends on bindings.

Definition 17. *Given an interface $\mathscr{I}$, a model $\mathscr{M}$, and a binding $\mathscr{B}$. The pair $(\mathscr{M},\mathscr{B})$ is a* ***preserving refinement*** *for $\mathscr{I}$, denoted $\mathscr{I} \succ_p (\mathscr{M},\mathscr{B})$, if $\forall p \in Ports_{\mathscr{I}}.\forall pos \in [1, max_p].\forall e \in EP_{\mathscr{R}}.inp_e \Rightarrow ip \in InPorts_{\mathscr{M}'} \wedge car_{dref(tid_e)} = X_{ip,\mathscr{M}'}$ or $\neg inp_e \Rightarrow ip \in OutPorts_{\mathscr{M}'} \wedge car_{dref(tid_e)} = Y_{ip,\mathscr{M}'}$, with $\mathscr{M}' = pdevs(\mathscr{M})$, $ip = impl(name_p, name_e, pos, \mathscr{B})$,and $\mathscr{R} = drefR(rid_p)$.*

A model refines an interface if all multiplicities of all declared interaction capabilities are bound to an existing implementation port. Preserving refinement ensures that all properties declared in an interface are realized by an implementation. This is an important relation, which however captures only one side of the refinement coin (Schröter 2004). A model implementation may require some ports to be connected during execution of a simulation, in the following referred to as $required(\mathscr{M})$. Whereas preserving refinement ensures that "good" properties are fulfilled by the implementation, reflecting refinement forces an implementation to not extend the interface with further, maybe "bad", properties.

Definition 18. *A model implementation $\mathscr{M}$ and a binding $\mathscr{B}$ are a* ***reflecting refinement*** *for an interface $\mathscr{I}$, denoted by $\mathscr{I} \succ_r (\mathscr{M},\mathscr{B})$, if $\forall ip \in required(\mathscr{M}').\exists p \in Ports_{\mathscr{I}}.\exists e \in EP_{drefR(rid_p)}.\exists pos \in [1, min_p]$, such that $ip = impl(p,e,pos,\mathscr{B})$, where $\mathscr{M}' = pdevs(\mathscr{M})$.*

Reflecting refinement ensures that each model port requiring a connection is represented in the interface. Combining both refinement relations yields full refinement: $\mathscr{I} \succ (\mathscr{M},\mathscr{B}) \overset{def}{=} \mathscr{I} \succ_r (\mathscr{M},\mathscr{B}) \wedge \mathscr{I} \succ_p (\mathscr{M},\mathscr{B})$. Full refinement ensures that an interface reflects all requirements of an implementation and that the implementation fulfills all declarations of the interface. Eventually, in a composition it comes to the question, whether these properties mutually fit together.

Definition 19. *A composition* (Sub, Con) *is* ***complete*** *if* $\forall c \in Con.\exists s, s' \in Sub.wellformed(c, s, s')$ *and* $\forall s \in Sub$ *with* $\mathscr{I} = drefI(iid_s)$ *and* $\forall p \in Ports_{\mathscr{I}}$ *and* $\forall m \in [1, min_p]$ $\exists (start, end) \in Con$ *such that* $if_{start} = name_s \wedge port_{start} = name_p \wedge pos_{start} = m$ *or such that* $if_{end} = name_s \wedge port_{end} = name_p \wedge pos_{end} = m$.

Refinement and completeness form the basic requirements on components to be correct and thereby deploy-able.

Definition 20. *A component* $\mathscr{C}$ *is* ***correct*** *if*

1. $\mathscr{I} \succ (\mathscr{M}_{\mathscr{C}}, \mathscr{B}_{\mathscr{C}})$, *with* $\mathscr{I} = drefI(if_{\mathscr{C}})$
2. $complete((Sub_{\mathscr{C}} \cup \{(\text{"`this"'}, if, \emptyset)\}, Con_{\mathscr{C}}))$
3. $\forall s \in Sub_{\mathscr{C}}.correct(instC(s))$
4. *Acyclic compositions*

An atomic component is correct if it refines its interface. For composite components correctness is defined recursively. Analysis of a composite component requires its full instantiation down to the leaves.

Theorem 1. *Let* $\mathscr{C}$ *be a component. If* $\mathscr{C}$ *is correct,* $model(\mathscr{C})$ *is a well-formed PDEVS model in which all required ports of its sub models are connected.*

Proof. Req. 2 of Def. 20 ensures that the composition (Sub, Con) passed to $couplings()$ is complete. Consequently there exist $s, s' \in Sub$ with $name_s = if_{start}$ and $name_{s'} = if_{end}$. Furthermore, req. 2 ensures that for s, s' there exists a $c = (start, end) \in Con$ such that $wellformed(c, s, s')$. Thus, $\mathscr{R} \neq \perp$ and $\mathscr{R}' \neq \perp$. Construction of matches M is done according to the three cases of possible relations between $\mathscr{R}$ and $\mathscr{R}'$, such that $\forall (f, t) \in M.\exists e \in EP_{\mathscr{R}} \exists e' \in EP_{\mathscr{R}'}.name_e = f \wedge name_{e'} = t$. From req. 3 of Def. 20 follows in combination with req. 1 that $\mathscr{I} \succ_p (\mathscr{M}, \mathscr{B})$ and $\mathscr{I}' \succ_p (\mathscr{M}', \mathscr{B}')$. From Def. 17 we derive that $ip \in InPorts_{\mathscr{M}}$ or $ip \in OutPorts_{\mathscr{M}}$ with according orientation. $\succ_r$ ensures that all required ports of each sub model is declared in the respective interface. Combined with $complete(Sub, Con)$ we can conclude that all required composite ports are connected and thereby an appropriate model coupling was constructed. Req. 4 ensures that $model(\mathscr{C})$ will not go to an infinite recursion.

□

The main challenge for component definitions is to show correctness for all possible combinations of allowed parametrization. Thereby, an increased flexibility induces higher efforts.

Definitions of formalisms and languages often neglect concrete syntax (Kleppe 2007). To become usable in practice, interfaces and composition structures need a concrete representation, preferably a platform-independent format that eases database integration.

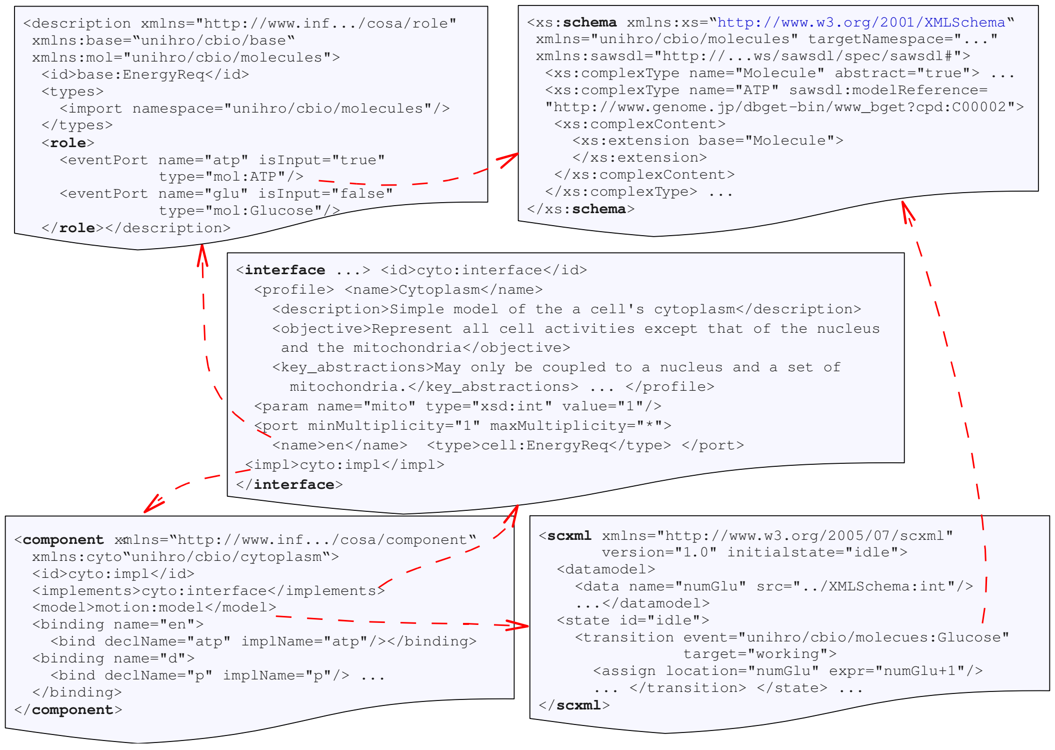
7 REPRESENTATION IN XML

In modeling and simulation tools, types of model components are usually defined in programming languages (Zeigler and Sarjoughian 2005, Brooks et al. 2007) and thereby bound to a particular type system, e.g. that of Java. Attempts to increase interoperability of systems, such as the web service architecture (W3C 2004a), utilize XML to abstract from tool-specific and programming language specific representations. Data descriptions based on XML are generally considered to be robust, extensible, and well suited to represent complex data structures (Harold 2002).

XML Schema Definition (XSD) allows to constrain the content of elements and attributes by type and value range assignments (W3C 2004b). Based on type definitions in XSD, roles may be separated from model definitions and reside in own XML documents – similar to interfaces defined in the Web Service Description Language, abbreviated WSDL, (W3C 2006b). Furthermore, type definitions become independent of modeling formalisms and simulation tools, general schema matching approaches may be used to check the compatibility of types (Röhl and Morgenstern 2007), and compatible to semantic annotations, e.g. according to SAWSDL (W3C 2006a).

XML, XSD, SAWSDL form the base for the concrete syntax of types, roles, interfaces, and components. As XML documents they may be stored in databases as separate units of definition. To increase the usability of interfaces, they are equipped with semi-structured data, which are not part of the set-theoretic formalism introduced above. However, if a model should be reused the concepts underlying a model definition need to be considered (Tolk and Muguira 2003). Assumptions, simplifications, and constraints made in the model are usually described in semi-structured data. Profile data may defined in an XML document of an interface similar to identification table of federates in the HLA (IEEE 2003).

Example 7. *Figure 8 shows XML documents defining the component* `Cytoplasm`*. References based on URIs are visualized by dashed arrows. The interface document forms the pivotal part of a component. The interface references a role definition to announce a composite port. The role document declares two event ports and imports type definitions made in XSD.* `ATP` *and* `Glucose` *are both derived from the abstract type molecule. In* `ATP`*, the attribute* `modelReference` *holds a reference to the ATP entry of KEGG (not displayed). The interface points to a component document containing the implementation for the interface. The component component references a model definition by a qualified name. The model itself is defined in SCXML (W3C 2007). A configurator may be defined in XSLT (W3C 1999),*



```
<description xmlns="http://www.inf.../cosa/role"
 xmlns:base="unihro/cbio/base"
 xmlns:mol="unihro/cbio/molecules">
  <id>base:EnergyReq</id>
  <types>
    <import namespace="unihro/cbio/molecules"/>
  </types>
  <role>
    <eventPort name="atp" isInput="true"
               type="mol:ATP"/>
    <eventPort name="glu" isInput="false"
               type="mol:Glucose"/>
  </role></description>
```

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
 xmlns="unihro/cbio/molecules" targetNamespace="..."
 xmlns:sawsdl="http://...ws/sawsdl/spec/sawsdl#">
  <xs:complexType name="Molecule" abstract="true"> ...
  <xs:complexType name="ATP" sawsdl:modelReference=
  "http://www.genome.jp/dbget-bin/www_bget?cpd:C00002">
   <xs:complexContent>
     <xs:extension base="Molecule">
     </xs:extension>
   </xs:complexContent>
  </xs:complexType> ...
</xs:schema>
```

```
<interface ...> <id>cyto:interface</id>
  <profile> <name>Cytoplasm</name>
    <description>Simple model of the a cell's cytoplasm</description>
    <objective>Represent all cell activities except that of the nucleus
     and the mitochondria</objective>
    <key_abstractions>May only be coupled to a nucleus and a set of
      mitochondria.</key_abstractions> ... </profile>
  <param name="mito" type="xsd:int" value="1"/>
  <port minMultiplicity="1" maxMultiplicity="*">
    <name>en</name>  <type>cell:EnergyReq</type> </port>
 <impl>cyto:impl</impl>
</interface>
```

```
<component xmlns="http://www.inf.../cosa/component"
  xmlns:cyto"unihro/cbio/cytoplasm">
  <id>cyto:impl</id>
  <implements>cyto:interface</implements>
  <model>motion:model</model>
  <binding name="en">
    <bind declName="atp" implName="atp"/></binding>
  <binding name="d">
    <bind declName="p" implName="p"/> ...
  </binding>
</component>
```

```
<scxml xmlns="http://www.w3.org/2005/07/scxml"
       version="1.0" initialstate="idle">
  <datamodel>
    <data name="numGlu" src="../XMLSchema:int"/>
    ...</datamodel>
  <state id="idle">
    <transition event="unihro/cbio/molecues:Glucose"
                target="working">
      <assign location="numGlu" expr="numGlu+1"/>
      ... </transition> </state> ...
</scxml>
```

Figure 8: Definition of a component by a set of XML documents

which can be specified itself in XML and thereby preserves platform independence of component specifications.

8 CONCLUSION

The introduced descriptions combine two advantages of existing composition approaches. Modular-hierarchical modeling formalisms separate between model definition and simulator implementation. Here, interface definitions became separated from their model implementations. Thereby, compositions can be analyzed solely based on interface definitions. As XML documents, interfaces may be stored in databases and can be analyzed for compatibility based on platform-independent type definitions in XSD. XSD provides compatibility to related standards like SAWSDL, which facilitates semantic annotations.

Composition structures are based on UML concepts but are tailored to the specific needs of discrete-event simulation. Formal semantics is provided by mapping compositions to the existing modeling formalism PDEVS. Compositions can be analyzed formally to check the compatibility of interfaces, the refinement between interfaces and implementations, the completeness of compositions, and the correctness of components. Correct components fulfill all requirements to be deployed and ensure that a proper simulation model can be derived.

REFERENCES

Brooks, C., E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng. 2007, Jan. Heterogeneous concurrent modeling and design in java (volume 1: Introduction to ptolemy ii). Technical Report UCB/EECS-2007-7, EECS Department, University of California, Berkeley.

Broy, M., M. L. Crane, J. Dingel, A. Hartman, B. Rumpe, and B. Selic. 2006. 2nd UML 2 semantics symposium: Formal semantics for UML. In *MoDELS 2006 Workshops*, ed. T. Kühne, Volume 4364 of *LNCS*, 318–323: Springer.

de Alfaro, L., and T. A. Henzinger. 2005. Interface-based design. In *Engineering Theories of Software-intensive Systems*, Volume 195 of *NATO Science Series: Mathematics, Physics, and Chemistry*, 83–104. Springer: M. Broy, J. Gruenbauer, D. Harel, and C.A.R. Hoare.

Elmqvist, H., S. E. Mattsson, and M. Otter. 2001. Object-oriented and hybrid modeling in modelica. *Journal Européen des systèmes automatisés* 35 (1): 1–10.

Harel, D., and B. Rumpe. 2004. Meaningful modeling: What's the semantics of semantics? *Computer* 37 (10): 64–72.

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

IEEE 2003, March. Recommended practice for high level architecture (HLA) federation development and execution process (FEDEP). Document 1516.3.

Janssen, T. M. V. 1997. Compositionality (with an appendix by B. Partee). In *Handbook of Logic and Language*, ed. J. van Benthem and A. ter Meulen, 417–473. Amsterdam: Elsevier.

Kleppe, A. 2007. A language description is more than a metamodel. In *Proceedings of the 4th International Workshop on Software Language Engineering (ateM 2007), Nashville, TN, USA, October 2007*, 15–23. Informatik-Bericht Nr. 4/2007, Johannes-Gutenberg-Universität Mainz, October 2007.

O'Keefe, G. 2006. Improving the definition of UML. In *MoDELS 2006, Genova, Italy, October 1-6, 2006*, ed. O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, Volume 4199 of *LNCS*, 42–56: Springer.

Oliver, I., and V. Luukkala. 2006, May 31 – June 2. On UML's composite structure diagram. In *SAM'06*. Kaiserslautern, Germany.

OMG 2007a, March. Systems modeling language (OMG SysML^{TM}) 1.0 specification. Proposed Available Specification ptc/2007-02-04.

OMG 2007b, November. Unified Modeling Language: Superstructure, v2.1.2. Document Number: formal/2007-11-02, available specification http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF.

Pierce, B. C. 2002. *Types and programming languages*. Cambridge, MA, USA: MIT Press.

Röhl, M., and S. Morgenstern. 2007. Composing simulation models using interface definitions based on web service descriptions. In *Proceedings of the 2007 Winter Simulation Conferenec*, ed. S. G. Henderson, B. Biller, M.-H. Hsieh, J. Shortle, J. D. Tew, and R. R. Barton, 815–822: Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Schröter, G. 2004. *Characterization and comparison of formal refinement and development relations for software modeling techniques*. Ph. D. thesis, Technical University of Berlin, School of Electrical Engineering and Computer Sciences.

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 level of conceptual interoperability model. Fall Simulation Interoperability Workshop (SISO), Orlando.

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.

Verbraeck, A. 2004. Component-based distributed simulations: the way forward? In *PADS '04: Proceedings of the eighteenth workshop on Parallel and distributed simulation*, 141–148. New York, NY, USA: ACM Press.

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

W3C 2004a. Web services architecture. http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/. W3C Working Group Note 11 February 2004.

W3C 2004b, October. XML Schema part 0: Primer second edition. http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/. W3C Recommendation 28 October 2004.

W3C 2006a. Semantic annotations for WSDL. http://www.w3.org/TR/2006/WD-sawsdl-20060928/. W3C Working Draft 28 September 2006.

W3C 2006b. Web services description language (WSDL) version 2.0 part 1: Core language. http://www.w3.org/TR/2006/CR-wsdl20-20060327. W3C Candidate Recommendation 27 March 2006.

W3C 2007. State chart XML (SCXML): State machine notation for control abstraction. http://www.w3.org/TR/2007/WD-scxml-20070221. W3C Working Draft 21 February 2007.

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. 2005, January. Introduction to DEVS modeling and simulation with JAVA: Developing component-based simulation models. Arizona Center for Integrative Modeling and Simulation, University of Arizona and Arizona State University, Tucson, Arizona, USA.

AUTHOR BIOGRAPHIES

MATHIAS RÖHL holds a MSc in Computer Science from the University of Rostock. 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>. Web address of his homepage 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 e-mail address is <lin@informatik.uni-rostock.de> and her Web page is <www.informatik.uni-rostock.de/~lin>.