Composing Variable Structure Models
A Revision of COMO

Alexander Steiniger and Adelinde M. Uhrmacher
Modeling and Simulation Chair, University of Rostock, Albert-Einstein-Str. 22, Rostock, Germany

Keywords: Component-based Modeling, Variable Structure Models, Variable Interfaces, Intensional Couplings.

Abstract: Component-based approaches aim at facilitating the storage, exchange, and reuse of components and their compositions. For this, components provide interfaces that formulate contracts composition can be based upon. Variable structure models imply the change of compositions, couplings, and even interfaces in terms of ports. Thus, combining variable structure models with a component-based approach poses specific challenges. We present a revision of the model composition framework COMO taking the specifics of variable structure models into account, e.g., by specifying interfaces as sets of parameters and supersets of ports, defining couplings intensionally, and introducing supersets of components as part of the compositional description. As target for generating executable simulation models the formalism ML-DEVS has been selected.

1 INTRODUCTION

Typically, complex systems, e.g., technical, social, or biological systems, comprise many different and often heterogeneous entities concurrently communicating and interacting with each other. Thus, many formalisms, aiming at modeling such systems, support constructing models by *parallel composition* of smaller model units (“components”) intrinsically, e.g., DEVS (Zeigler et al., 2000). or process algebras (Bergstra and Klop, 1989). Thereby, individual components advance in parallel (concurrently). Going one step further, component-based modeling approaches emphasize the notion of self-contained model components and foster their reuse (Verbraeck, 2004), also by third parties. However, to be usable in unforeseen contexts and for different purposes, a model component needs to announce its functionality by well-defined interfaces (Röhl and Uhrmacher, 2008).

Moreover, a variable structure characterizes many complex systems, i.e., their composition, the interaction between their components, and the components' behavior patterns change over time (Uhrmacher, 2001). Diverse modeling languages and formalisms have been developed to address this variability: either by extending existing modeling formalisms, e.g., variants of DEVS like (Barros, 1995), or by offering dynamic structure as salient feature from the outset, e.g., the π-calculus (Milner, 1999). All of those support a kind of *sequential composition*, as they provide structure in the temporal dimension, i.e., they determine which model incarnation replaces another under which circumstances and at which time.

Both types of composition seem to be at odd with each other. Model components are self-contained *building blocks* that have a well-defined interface and encapsulate certain aspects of the simulated system, whereas variable structure models can change their structure and behavior. Those changes occur during model execution (runtime), whereas assembling model components to simulation models is done beforehand (at configuration time) and separately from the execution of the composed simulation model (Petty and Weisel, 2003). To explore possibilities to support both, “classic” and sequential composition, we chose as starting points: (i) the interface and composition description that is the basis of the model composition framework COMO (Röhl and Uhrmacher, 2008) and (ii) the modular, hierarchical modeling formalism ML-DEVS (Steiniger et al., 2012).

2 EXAMPLE: RNA FOLDING

To illustrate our approach, we use the RNA folding model presented in (Maus, 2008) and specified in the formalism ML-DEVS (Steiniger et al., 2012). The model is a composition itself comprising a coupled model, which represents the RNA molecule, and

Steiniger A. and Uhrmacher A..
Composing Variable Structure Models - A Revision of COMO.
DOI: 10.5220/0004486302860293
In *Proceedings of the 3rd International Conference on Simulation and Modeling Methodologies, Technologies and Applications* (SIMULTECH-2013), pages 286-293
ISBN: 978-989-8565-69-3

the molecule's components, the nucleotides. Each nucleotide is an instance of the atomic model *Nucleotide*, which can be of type *A*, *C*, *G*, or *U* referring to its bound nucleobase. In the folding model, nucleotides are connected via ports according to their *primary structure* in a linear sequence (bond). Those ports represent the 3' and 5' carbon atoms, which are linked. In addition, a *secondary structure* will be established by dynamically adding connections between eligible nucleotides. For this, a nucleotide signalizes a binding request to the RNA molecule (macro model), which selects a potential partner according to different strategies, e.g., considering entropy. If the partner is able to bind, both nucleotides will announce additional ports for the connection and the macro model will connect those ports. The communication and interaction between the molecule and nucleotides is realized through *up- and downward causation* as provided by the formalism ML-DEVS.

3 COMPOSITION OF VARIABLE STRUCTURE MODELS

We chose the model composition framework COMO (Röhl and Uhrmacher, 2006) as starting point for a component-based model design. Its underlying description language allows specifying interfaces of model components and their compositions explicitly in a platform independent manner and separately from the actual implementation of the associated models. The description language is set-theoretically defined and represented by XML schema definitions (Röhl and Uhrmacher, 2008). Thus, composition specifications consist of XML documents, which can be stored in and retrieved from a repository. COMO can also analyze the syntactic correctness of compositions and transform them into executable simulation models in a certain target formalism. The execution of derived models (simulation) is not carried out by COMO, as it does not provide an execution engine. However, COMO can be added as an additional specification and analysis layer on top of a simulation system, which is then used for the execution. Figure 1 shows the relation between the description units of COMO and illustrates schematically the steps to derive an executable simulation model from a composition specification.

So far, similar to other component-based approaches, such as COST (Chen and Szymanski, 2002) or CODES (Szabo and Teo, 2007), COMO is not addressing variable interfaces and structures, i.e., sequential composition (composition over time), but assumes static compositions instead. The question is now, how variability can be reflected at the specification and analysis level while maintaining the separation of composition, implementation, and execution. To cope with and support structural changes, we started to revise and extend COMO. Basically, our approach embraces the following three steps:

1. Adapt the interface and composition description, COMO is based on, to cope with the shift from static to variable interfaces and structures.
2. Choose an appropriate modeling formalism that allows expressing variable structures to show the applicability of the approach.
3. Revise and adapt the existing consistency checks of COMO.

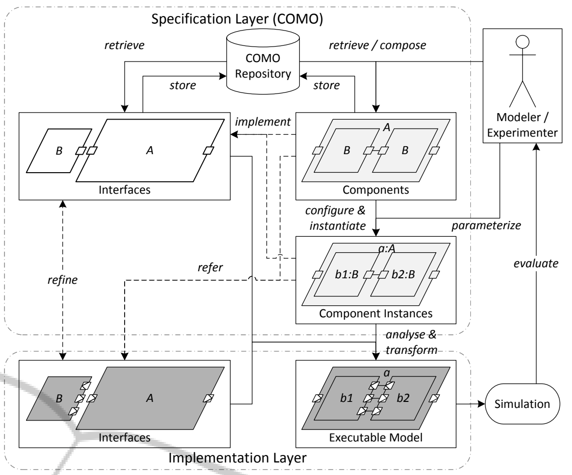

Figure 1: The derivation of an executable simulation model from a composition specification in COMO (specification layer) and model implementations in certain source formalisms (implementation layer). First, a given component/composition is configured and instantiated according to specified parameter values. Afterwards, the component instances are analyzed and transformed into an executable model that can then be executed.

3.1 Revision of COMO

Our first revision refers to the adaption and generalization of *interfaces*, the fundamental constructs in COMO. Although variable structures do not imply dynamic interfaces, some systems, as can be found in the biological domain, are characterized by the plasticity of their interfaces (Uhrmacher et al., 2007).

Specifying an interface with a static set of ports is straightforward. In case of variable interfaces, a concrete manifestation of ports depends on the component's state and external events, and may change during simulation. Thus, different incarnations of an interface can exist. Specifying all incarnations seems to be not practically. If we do not allow generating new names during execution, we can assume the superset of those variable ports, i.e., all potential ports a

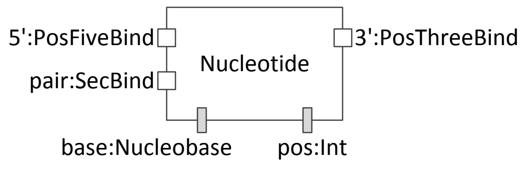

Figure 2: Interface of the component Nucleotide with three composite ports and two parameters.

model can exhibit, to be known before the execution.

Definition 1. *An **interface** is a tuple (id, Ports, Params) with a unique identifier id $\in$ QName, a finite superset of composite ports Ports, and a finite set of parameter declarations Params. Ports and parameters must be unambiguous.*

In contrast to (Röhl and Uhrmacher, 2008), an *interface definition* does not refer to a component that implements the interface. Several components can now implement the same interface. The superset of ports is encoded by declaring all potential *composite ports* in *Ports*. Composite ports have a name and *role* assigned to them[1]. Roles are defined by an *id* and a list of *event ports*. An event port consists of a name, message type, and direction (in- or output port). A *parameter declaration* has a unique name and a reference to a *type*. Figure 2 shows the interface of the component *Nucleotide* comprising three composite ports (white boxes) that reflect the three binding sites, which establish the primary and secondary structure, and two parameters (gray boxes) defining the nucleotide's base (*base*) and position in the primary structure (*pos*). All port and parameter labels consist of port and parameter names and references to roles and types, respectively, separated by ':'.

Interfaces serve as contracts between model components, whose implementations need to adhere to those contracts. Therefore, ports at the interface level must be related to concrete model ports in the implementations unambiguously. This can be done explicitly via *bindings*. If no binding is defined, COMO binds an event port to an implementation port whose names equals those of the event port.

Definition 2. *$\mathscr{B}$ is a finite set of **bindings** of a component c. Each binding $b \in \mathscr{B}$ is a tuple (port, decl, impl), where port $\in$ String is the name of a composite port, decl $\in$ String is the name of an event port (declared port), and impl $\in$ String is the the name of the assigned model port (implementation port). It holds that $\forall b, b' \in \mathscr{B}$: $((b.port = b'.port \wedge b.decl = b'.decl) \Rightarrow (b = b'))$.*

[1]We distinguish between identifiers and names. The former are given as qualified names and used to identify and refer to other description units unambiguously, whereas the latter are used within definitions as variable names.

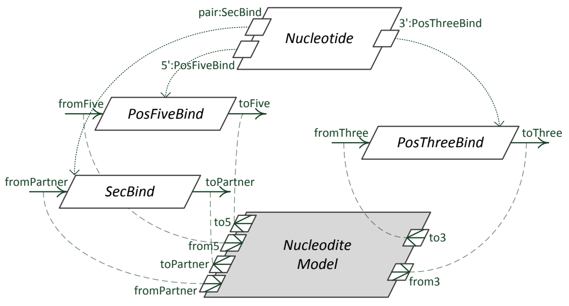

Figure 3: Relations between composite ports, roles, event ports, and implementation ports.

We write *b.port* etc. to access the corresponding elements of a given definition (tuple). The function *impl* returns the name of a model port for a given composite and event port name and a binding set:

$$impl(cpn, epn, \mathscr{B}) = \begin{cases} b.impl & \text{if } \exists b \in \mathscr{B}\colon\ cpn = b.port \wedge epn = b.decl, \\ epn & \text{otherwise.} \end{cases}$$

Figure 3 illustrates the relation between composite ports, roles, event ports, and implementation ports. The implemented model (gray rhomboid) exhibits ports (white rhombuses whit arrows) that are bound to event ports (dashed lines). Event ports are structured into roles, which describes complex interaction capabilities. Finally, roles are referenced by composite ports of the interface (dotted lines).

A *component definition* relates an interface to a given *model definition* (implementation) in a certain formalism (*source formalism*). We adapt the definition by introducing *loose connections* to deal with variable interfaces and structures.

Definition 3. *A **component** is a tuple (id, if, k, μ, $\mathscr{B}$, Sub, LCon) with a unique identifier id $\in$ QName, a reference to an interface definition if $\in$ QName, a configuration function k, a model definition μ (implementation), a finite binding set $\mathscr{B}$, a set of subcomponents Sub, and a set of loose connections LCon. Sub consists of tuples (name, iid, cid, Params) with a name $\in$ String, interface reference iid $\in$ QName, component reference cid $\in$ QName, and a set of parameter assignments Params. Each parameter assignment comprise a name and value. In case of an atomic component, Sub and LCon will be empty.*

Loose connections extend the idea of *multi-couplings* (Uhrmacher et al., 2007; Steiniger et al., 2012). They are defined at the level of interfaces, i.e., composite ports.

Definition 4. *A **loose connection** lc of a component c, i.e., lc $\in$ c.LCon, is defined as partial function: $Sub \times Sub \rightsquigarrow 2^{\{(start,end) \in String \times String\}}$, where Sub is* *defined as in Definition 3 and $start, end$ are names of the composite ports that shall be connected. The domain of lc can be specified by a relation $R_{lc} \subseteq Sub \times Sub$.*

Example 1. *Let lc be a loose connection that links the 3' and 5' carbon atoms of all pairs of consecutive nucleotides within a backbone bond, i.e., the primary structure: $lc(s,s')$ equals $\{(3',5')\}$ if $(s,s') \in R_{lc}$ or $\perp$ otherwise, with $R_{lc} = \{(s,s') \in Sub \times Sub \,|\, \exists p \in s.Params, p' \in s'.Params$: $p.name = p'.name =$ "pos" $\wedge\, p'.value = p.value + 1 \wedge s \neq s' \wedge s.iid = s'.iid =$ "$Nucleotide$"$\}$.*

For instantiating components, the configuration function or configurator k plays a crucial role. It allows changing the component's internal structure and determines the initially available sub-components, loose connections, and composite ports according to given parameter values.

Definition 5. *A* ***configurator*** *k is a function that maps the tuple (Params, Sub, LCon, $\mathscr{B}$) onto the tuple (Sub', LCon', $\mathscr{B}'$, Sub_i, $LCon_i$, $Ports_i$, Params').*

An implementation of k has to evaluate given parameter values and delegate them to sub-components if necessary. Given a configuration (*name*, *iid*, *cid*, *Params*) that refers to a concrete component and its context, we now adapt the central function $instC(.)$ to return a fully instantiated composition. For this, the compositional hierarchy is traversed, starting from the referenced component, and the specified configuration functions are applied successively. Algorithm 1 shows the instantiation, where $\mathscr{C}$ is the set of all component definitions.

Algorithm 1: Instantiation of a component definition.

name: *instC*
input: *name*, interface reference *cid*, component reference *iid*, parameter values *Params*
output: instantiated composition *ci*

```
//get component definition
if $\exists c' \in \mathscr{C}$ with $c'.id = cid$ then $c = c'$ else $c = \perp$
//check if c valid
if $c \neq \perp$ and $c.if = iid$ then
 //apply configurator of current component
 $Sub, LCon, \mathscr{B}, Sub_i, LCon_i, Ports_i, Params'$
   $= c.k(Params, c.Sub, c.LCon, c.\mathscr{B})$
 //instantiate recursively
 $ci = (name, c.id, c.if, c.\mu, \bigcup_{s \in Sub}\{instC(s)\}, LCon,$
   $\mathscr{B}, Com_i, LCon_i, Ports_i, Params')$
   with $ci.Com_i = \bigcup_{s \in Sub_i}\{ci' \in ci.Com | ci'.name = s.name\}$
 //return instantiated composition
 return $ci$
else
 return $\perp$
```

In our revision, a component can have several instances sharing the same component definition. We introduce the definition of a *component instance* representing a component in a concrete context, i.e., with a concrete parameterization and initial structure.

Definition 6. *A* ***component instance*** *is a tuple (name, cid, iid, μ, Com, LCon, $\mathscr{B}$, Com_i, $LCon_i$, $Ports_i$, Params) with a unique name $\in$ String, a reference to a component definition cid $\in$ QName, a reference to a interface definition iid $\in$ QName, a model definition μ, a set of (sub-)component instances Com, a set of loose connections LCon, a set of bindings $\mathscr{B}$, and the sets $Com_i \subseteq Com$, $LCon_i \subseteq LCon$, and $Ports_i \subseteq String$ that reflect the initial structure. Params is a set of parameter assignments.*

Figure 4 shows the instantiation and initialization of a composite component *mRNA* (ride side) according to the given parameter values, i.e., the RNA molecule should contain 38 nucleotides with varying nucleobases. On the left side, only the initial structure and ports of the model and its components are depicted.

3.2 Incorporation of Multi-Level-DEVS

The modeling formalism ML-DEVS supports variable ports and structures, intensional couplings, and provides additional mechanisms for up- and downward causation between different levels of behavior (Steiniger et al., 2012). Moreover, ML-DEVS has been and is currently used in different application domains, from demography (Zinn, 2011) to smart environments (Krüger et al., 2012). Therefore, we chose ML-DEVS as target formalism. For a thorough introduction to ML-DEVS please refer to (Steiniger et al., 2012).

In COMO, the function $model(.)$ transforms component instances and their implementations into executable simulation models. For this purpose, *metamodels* of the source formalism(s) and target formalism are needed that describe how models are specified in those formalisms. To use ML-DEVS as target, the transformation function, returning ML-DEVS models, is defined as follows:

Definition 7. *Let ci be a component instance, mm_{source} be the meta-model of an arbitrary source formalism, mm_{mldevs} be the meta-model of ML-DEVS, and the function $metamodel(\mu)$ return the meta-model of the formalism in which the given model μ is defined. Then, the transformation is defined by:*

$$model(cid, mm_{source}, mm_{mldevs}) = \begin{cases} micro(cid, metamodel(ci.\mu)) & \textit{if } ci.Com = \varnothing, \\ macro(cid, metamodel(ci.\mu)) & \textit{otherwise}. \end{cases}$$

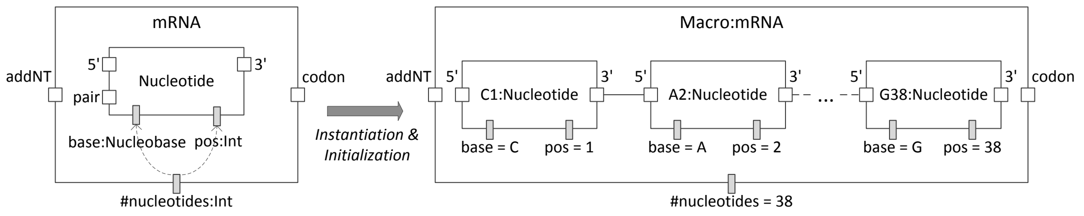

Figure 4: Instantiation and initialization of the component mRNA and its sub-components, i.e., nucleotides.

So, the transformation into ML-DEVS consists of two different functions: $micro(.)$ and $macro(.)$. The former maps atomic components onto MICRO-DEVS models, whereas the latter maps composite components onto MACRO-DEVS models. Both functions share similar functionality, as MACRO-DEVS models have also a state and behavior. Additionally, $macro(.)$ is responsible for transforming the sub-components of a composite component and synthesizes the communication structure from the given descriptions:

$$\begin{aligned} C &= \bigcup_{c \in ci.Com} \{model(c, mm_{source}, mm_{mldevs})\}, \\ &\quad \text{with } mm_{source} = metamodel(c.\mu) \\ MC &= multiCouplings(ci). \end{aligned}$$

For our prototype we also chose ML-DEVS as source formalism, which makes the implementation of the transformation straightforward. If formalisms different from the actual target formalism shall be used to implement components, a similar approach as in (de Lara and Vangheluwe, 2002) can be exploited.

The mapping from loose connections onto *multi-couplings* of the target ML-DEVS is done by an auxiliary function, called $multiCouplings(.)$. Please note that the generic multi-couplings in ML-DEVS are defined intensionally based on port names. If ports with those names exist during execution, an extensional coupling, as known from other DEVS variants, is installed (Steiniger et al., 2012). A reference implementation of the mapping is shown in Algorithm 2. For each loose connection lc it is checked whether or not a combination of sub-components (including the composite component) is in the domain of lc (line 15). If so, all values returned by lc, i.e., pairs of composite port names, are iterated and the referenced roles[2] (lines 15 to 24) are *matched* (lines 26 to 29) similarly as done when resolving composition connections (cf. (Röhl and Uhrmacher, 2008)). For this, $matches(r, r')$ is defined as follows:

Definition 8. *Let r, r' be role definitions, then* $matches(r, r') \Leftrightarrow \{(f, t) \in String \times String \mid \exists e \in r.EP, e' \in r'.EP\colon f = e.name \wedge t = e'.name \wedge e.inp \neq e'.inp \wedge (e.inp \Rightarrow drefT(e'.tid) \sqsubseteq drefT(e.tid)) \wedge (e'.inp \Rightarrow drefT(e.tid) \sqsubseteq drefT(e'.tid))\}$.

[2]The functions $drefI(.)$, $drefR(.)$, and $drefT(.)$ retrieve corresponding definitions based on a given identifier.

Algorithm 2: Construction of multi-couplings.

name: *multiCouplings*
input: instance of composite component *ci*
output: set of multi couplings *MC*

```
1   $MC = \varnothing$
2   for each $lc \in ci.LCon$ do
3     //iterate all component pairings
4     for each $s \in ci.Com \cup \{ci\}$ do
5       for each $s' \in ci.Com \cup \{ci\}$ do
6         //prevent direct feedback loops
7         if $s = s'$ then continue with next pair
8         //get and check interfaces
9         if $(i = drefI(s.if)) = \perp$ then //error
10        if $(i' = drefI(s'.if)) = \perp$ then //error
11        //build interface instances
12        $sub = (s.name, i.id, s.cid, s.Params)$
13        $sub' = (s'.name, i'.id, s'.cid, s'.Params)$
14        //iterate all potential port name pairs
15        for each $(start, end) \in lc(sub, sub')$ with
              $lc(sub, sub') \neq \perp$ do
16          //get ports and roles
17          if $\exists port \in s.Ports$ with $port.name = start$
18            then $p = port$
19            else continue with next port pair
20          if $\exists port \in s'.Ports$ with $port.name = end$
21            then $p' = port$
22            else continue with next port pair
23          if $(r = drefR(p.rid)) = \perp$ then \\error
24          if $(r' = drefR(p'.rid)) = \perp$ then \\error
25          //match event ports of connected roles
26          $M = \varnothing$
27          if $s = c$ then $M = matches(\overline{r}, r')$
28          else if $s' = c$ then $M = matches(r, \overline{r'})$
29          else $M = matches(r, r')$
30          //iterate all event port matches
31          for each $(from, to) \in M$ do
32            //get names of implementation ports
33            $ip = impl(p.name, from, s.\mathscr{B})$
34            $ip' = impl(p'.name, to, s'.\mathscr{B})$
35            //check orientation of model coupling
36            $e = dp$ with $dp \in r.EP.\ dp.name = from$
37            if $(s = c \wedge e.inp)$ or $(s \neq c \wedge \neg e.inp)$
38              then $MC \mathrel{+}= (ip, ip')$ //keep orientation
39            else if $(s \neq c \wedge e.inp)$ or $(s = c \wedge \neg e.inp)$
40              then $MC \mathrel{+}= (ip', ip)$ //invert orientation
41            else do nothing
42  return $MC$
```

The relation $t \sqsubseteq t'$ indicates that type t is a sub-type of t'. Furthermore, $\bar{r}$ indicates that the direction of all event ports declared in role r are inverted, so $\bar{r} := (\bot, r.\overline{EP})$ with $r.\overline{EP} = \{(e.name, e.tid, \neg e.inp) \mid (e.name, e.tid, e.inp) \in r.EP\}$. Finally, for each match, the names of the bound implementation ports are retrieved (lines 33 to 34) and a multi-coupling between those names is added to the returning coupling set, considering the event port's direction (lines 36 to 41). The algorithm ignores all matches that result in inconsistent multi-couplings, such as direct feedback loops by connecting ports of the same component (line 7).

One difference between the previous composition connections and the novel loose connections is: the former are resolved into concrete, extensional model couplings, whereas the latter are resolved into intensional multi-couplings. Those intensional couplings have to be resolved into concrete, extensional model couplings in a second step during the execution of the derived simulation model. This resolution cannot be done by COMO, but has to be done by the simulator of ML-DEVS. The impact of using intensional couplings on consistency checked are discussed in the next section.

3.3 Analysis

Assuring *correctness by construction* by analyzing components and compositions merely based on conceptual specifications, such that the function *model*(.) construct a proper simulation model, is another motivation of COMO (Röhl and Uhrmacher, 2008). However, the components' implementations have to be considered as well for checking some properties. So far, correctness is checked at the syntactic level, in terms of *syntactic composability* as defined in (Petty and Weisel, 2003; Szabo and Teo, 2007).

When checking the *consistency* of (parallel) compositions, the *compatibility* of interfaces and in particular the compatibility of the connected composite ports is of interest. Composite ports are compatible if their roles are compatible, denoted by $r \sim r'$, i.e., if for each event port declared in r a counterpart with the opposite direction and a suitable message type (subtype relation must hold) in r' exists and vice versa. Although we are now dealing with variable ports and structures, knowing the supersets of ports and subcomponents allows us to check whether or not a loose connection is *well-defined* in general, similarly as it was done for composite connections.

Definition 9. *A loose connection lc is* ***well-defined*** *for a component instance ci, denoted by* $welldefined_{ci}(lc)$*, if there exist* $s, s' \in Sub$*, with* $s \neq s'$ *and* $lc(s,s') \neq \bot$*, for which hold* $\forall (start, end) \in lc(s,s') \exists p \in i.Ports, p' \in i'.Ports$: $p.name = start \wedge p'.name = end \wedge ((s = ci \Rightarrow \bar{r} \sim r') \wedge (s' = ci \Rightarrow r \sim \bar{r'}) \wedge ((s \neq ci \wedge s' \neq ci) \Rightarrow r \sim r'))$*, where* $Sub = \bigcup_{c \in ci.Com \cup \{ci\}} \{(c.name, c.iid, c.cid, c.Params)\}$*,* $i = drefI(s.iid)$*,* $i' = drefI(s'.iid)$*,* $r = drefR(p.rid)$*, and* $r' = drefR(p'.rid)$*.*

However, except for the initial state, we cannot make assumptions at the specification layer on the actual existence of concrete model couplings, into which loose connections are resolved during execution. Please note, due to the intensional definition some inconsistencies can no longer occur, as explicitly (current) properties of interfaces are taken into account. In contrast, before it could have happened that connections had been defined based on ports that did not exist during execution.

Based on the above definition, we can check the *completeness* of a given component instance, which may be the root of a composition, i.e., we analyze the consistency of the specification:

Definition 10. *A component instance ci is* ***complete****, denoted by* $complete(ci)$*, if:*

1. *all referenced types, interfaces, roles, and components exist,*
2. $\forall p \in ci.Params\ \exists p' \in i.Params: p.name = p'.name \wedge p.value \in t.Car$ *with* $t = drefT(p'.tid)$*,*
3. $\forall lc \in ci.LCon$: $wellformed_{ci}(lc)$*,*
4. *the initial structure state is valid, i.e.,* $ci.Com_i \subseteq ci.Com$*,* $ci.LCon_i \subseteq ci.LCon$*, and* $ci.Ports_i \subseteq \bigcup_{p \in i.Ports} \{p.name\}$ *with* $i = drefI(ci.if)$*,*
5. *all names of sub-components are unique,*
6. $\forall c \in ci.Com$: $complete(c)$*.*

As we cannot decide whether or not ports will be connected during execution and a component may use ports to signalize state changes to its parent, we, in contrast to (Röhl and Uhrmacher, 2008), do not require that composite ports have to be connected.

Although the implementation of a model component (model definition) is decoupled from its interface definition, both have to relate to each other. Hence, COMO checks whether the model definition *refines* the specified interface and vice versa based on bindings. As composite ports have no longer a connectivity range, we adapt the definition given in (Röhl and Uhrmacher, 2008) as follows:

Definition 11. *A model definition* μ *and a set of bindings* $\mathscr{B}$ *are a* ***preserving refinement*** *for an interface i, denoted by* $i \succ_p (\mu, \mathscr{B})$*, if* $\forall p \in i.Ports \forall e \in r.EP$: $ip \in \mu'.Ports \wedge t.Car = range_{ip}(\mu'.XY)$ *with* $r = drefR(p.rid)$*,* $ip = impl(p.name, e.name, \mathscr{B})$*,* $\mu' = model(\mu)$*, and* $t = drefT(e.tid)$*.*

In COMO we distinguish between in- and output ports, but during the transformation into executable ML-DEVS models the direction of ports will be ignored, as ML-DEVS does not make this distinction.

Complementing the *preserving refinement*, the *reflecting refinement* assures that all requirements of the model definition, i.e., required ports, are reflected in the interface and an analysis of a composition solely based on interfaces is possible in the first place. Therefore, all required ports, denoted by $required(\mu)$, that a model may use to communicate messages or signalize state changes to its surroundings should be reflected in the model's interface.

Definition 12. *A model definition* μ *and a set of bindings* $\mathscr{B}$ *are a* ***reflecting refinement*** *for an interface i, denoted by* $i \succ_r (\mu,\mathscr{B})$*, if* $\forall ip \in required(\mu')\exists p \in i.Ports \exists e \in r.EP$: $ip = impl(p.name, e.name, \mathscr{B})$ *with* $\mu' = mldevs(\mu)$ *and* $r = drefR(p.rid)$*.*

However, the question remains, how to retrieve the set $required(\mu)$ from a given model definition. So far, the modeler has to define the set. Combining both refinements yields *full refinement*, denoted by $i \succ (\mu,\mathscr{B}) \Leftrightarrow i \succ_p (\mu,\mathscr{B}) \wedge i \succ_r (\mu,\mathscr{B})$, assuring that an interface definition captures all requirements of a model definition and vice versa (Röhl and Uhrmacher, 2008).

Finally, we adapt the definition of the *correctness* of a component instance as follows:

Definition 13. *A component instance ci is* ***correct****, denoted by* $correct(ci)$*, if*

1. $i \succ (ci.\mu, ci.\mathscr{B})$ *with* $i = drefI(ci.if)$*,*
2. $complete(ci)$*,*
3. $\forall c \in ci.Com$: $correct(c)$*,*
4. *no sub-component of ci refers to ci (acyclic compositions).*

A correct component instance can be deployed and transformed into a proper simulation model. In contrast to (Röhl and Uhrmacher, 2008), we check completeness and correctness at the level of component instances.

4 DISCUSSION AND CONCLUSIONS

The overall promise of a component-based design with a strict separation of interface and model is that models can be stored and composed more easily and compositions can be analyzed mostly independent of implementation details. To exploit these capabilities for models with variable interfaces and structures, we revised the existing model composition framework COMO.

We adapted how components and their compositions are specified in COMO. Now we distinguish more clearly between components and their instances. We generalized interfaces so that different components can refer to the same interface, which is more in line with the usual meaning of interfaces (see also (Tolk and Muguira, 2003) for the role of interfaces). This also implies that the modeler is responsible to assure that a component implements a specific interface. To take variable ports into account, the interface definition includes supersets of ports. We replaced the existing extensional couplings by a loose coupling scheme, which defines and constrains couplings intensionally based on properties of the subcomponents' interfaces and their concrete manifestations during execution. For this, a highly flexible and yet expressive scheme for defining couplings has been realized. In addition, this revision allowed us to get rid of some burden that COMO had been carrying around.

In the analysis phase, a compositional description is checked by COMO for being complete and correct. Therefore, different properties of the composition are of interest: (i) the direction and consistency of types of ports that should be connected according to the intensional couplings, (ii) the initial structure of the model (its components and couplings) that should relate to the overall definition, and (iii) the refinement relation between implemented model and component instance. COMO is independent of modeling formalisms. However, it requires meta models of modeling formalisms and transformation functions to translate models from a source formalism into a target formalism. For our prototypical realization we chose the same source and target formalism for our models, ML-DEVS. The formalism supports variable ports and structures and employs intensional couplings. However, the revised COMO would also work for models with a static structure and the old target formalism, P-DEVS (Chow and Zeigler, 1994).

Several avenues for further research exist: For instance, it would be interesting to see how COMO can be mapped into other suitable target formalisms, e.g., Beta binders (Priami and Quaglia, 2005). Also the transformation of models into the target formalism based on suitable meta-models offers an entire field of research. So far, we also have not exploited one fundamental characteristic of ML-DEVS, its mulitlevelness (relating macro and micro levels). Here, it would be interesting to examine the demands of multi-level modeling beyond classic hierarchical decomposition on component interfaces and coupling schemes. However, to come up with a not too ML-DEVS-specific solution, other multi-level modeling approaches need to be inspected as well, such as (Oury and Plotkin, 2011).

ACKNOWLEDGEMENTS

This research is supported by the German Research Foundation (DFG) within the context of the project MuSAMA (Multimodal Smart Appliance Ensembles for Mobile Applications).

REFERENCES

Barros, F. J. (1995). Dynamic Structure Discrete Event System Specification: A New Formalism for Dynamic Structure Modeling and Simulation. In Alexopoulos, C., Kang, K., Lilegdon, W. R., and Goldsman, D., editors, *Proceedings of the 1995 Winter Simulation Conference*, pages 781–785. IEEE Computer Society.

Bergstra, J. A. and Klop, J. W. (1989). ACT$_\tau$: A Universal Axiom System for Process Specification. In Wirsing, M. and Bergstra, J. A., editors, *Algebraic Methods: Theory, Tools and Applications*, volume 394 of *Lecture Notes in Computer Science*, pages 447–463. Springer-Verlag.

Chen, G. and Szymanski, B. K. (2002). COST: A Component-Oriented Discrete Event Simulator. In Yücesan, E., Chen, C.-H., Snowdon, J. L., and Charnes, J. M., editors, *Proceedings of the 2002 Winter Simulation Conference*, pages 776–782. IEEE Computer Society.

Chow, A. C. and Zeigler, B. P. (1994). Parallel DEVS: A parallel, hierarchical, modular modeling formalism. In Tew, J. D., S., M., Sadowski, D. A., and Seila, A., editors, *Proceedings of the 1994 Winter Simulation Conference*, pages 716–722. IEEE Computer Society.

de Lara, J. and Vangheluwe, H. L. M. (2002). AToM3: A Tool for Multi-formalism and Meta-modelling. In Kutsche, R.-D. and Weber, H., editors, *Fundamental Approaches to Software Engineering*, volume 2306 of *Lecture Notes in Computer Science*, pages 174–188. Springer-Verlag.

Krüger, F., Steiniger, A., Bader, S., and Kirste, T. (2012). Evaluating the robustness of activity recognition using computational causal behavior models. In Dey, A. K., Chu, H.-H., and Hayes, G., editors, *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, pages 1066–1074. ACM Press.

Maus, C. (2008). Component-Based Modelling of RNA Structure Folding. In Heiner, M. and Uhrmacher, A. M., editors, *Computational Methods in Systems Biology*, volume 5307 of *Lecture Notes in Computer Science*, pages 44–62. Springer-Verlag.

Milner, R. (1999). *Communicating and Mobile Systems: the π-calculus*. Cambridge University Press, 1st edition.

Oury, N. and Plotkin, G. D. (2011). Coloured Stochastic Multilevel Multiset Rewriting. In Fages, F., editor, *Computational Methods in Systems Biology*, pages 171–181. ACM Press.

Petty, M. D. and Weisel, E. W. (2003). A Composability Lexicon. In *Proceedings of the Spring 2003 Simulation Interoperability Workshop*, pages 181–187.

Priami, C. and Quaglia, P. (2005). Beta Binders for Biological Interactions. In Danos, V. and Schachter, V., editors, *Computational Methods in Systems Biology*, volume 3082 of *Lecture Notes in Computer Science*, pages 20–33. Springer-Verlag.

Röhl, M. and Uhrmacher, A. M. (2006). Composing Simulations from XML-Specified Model Components. In Perrone, L. F., Wieland, F. P., Liu, J., Lawson, B. G., Nicol, D. M., and Fujimoto, R. M., editors, *Proceedings of the 2006 Winter Simulation Conference*, pages 1083–1090. IEEE Computer Society.

Röhl, M. and Uhrmacher, A. M. (2008). Definition and Analysis of Composition Structures for Discrete-Event Models. In Mason, S. J., Hill, R. R., Mönch, L., Rose, O., Jefferson, T., and Fowler, J. W., editors, *Proceedings of the 2008 Winter Simulation Conference*, pages 942–950. IEEE Computer Society.

Steiniger, A., Krüger, F., and Uhrmacher, A. M. (2012). Modeling Agents and their Environment in Multi-Level-DEVS. In Laroque, C., Himmelspach, J., Pasupathy, R., Rose, O., and Uhrmacher, A. M., editors, *Proceedings of the 2012 Winter Simulation Conference*. IEEE Computer Society. Article No. 233.

Szabo, C. and Teo, Y. M. (2007). On Syntactic Composability and Model Reuse. In Al-Dabass, D., Zobel, R., Abraham, A., and Turner, S., editors, *Proceedings of the First Asia International Conference on Modelling & Simulation*, pages 230–237. IEEE.

Tolk, A. and Muguira, J. A. (2003). The Level of Conceptual Interoperability Model. In *Fall Simulation Interoperability Workshop*.

Uhrmacher, A. M. (2001). Dynamic Structures in Modeling and Simulation: A Reflective Approach. *ACM Transactions on Modeling and Computer Simulation*, 11(2):206–232.

Uhrmacher, A. M., Ewald, R., John, M., Maus, C., Jeschke, M., and Biermann, S. (2007). Combining Micro and Macro-Modeling in DEVS for Computational Biology. In Henderson, S. G., Biller, B., Hsieh, M.-H., Shortle, J., Tew, J. D., and Barton, R. R., editors, *Proceedings of the 2007 Winter Simulation Conference*, pages 871–880. IEEE Computer Society.

Verbraeck, A. (2004). Component-based Distributed Simulations. The Way Forward? In *Proceedings of the 18th Workshop on Parallel and Distributed Simulation*, pages 141–148. IEEE Computer Society.

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

Zinn, S. (2011). *A Continuous-Time Microsimulation and First Steps Towards a Multi-Level Approach in Demography*. Dissertation, University of Rostock, Rostock, Germany.