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 (Pettie 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
Composing Variable Structure Models - A Revision of COMO

the molecule’s components, the nucleotides. Each nucleotide is an instance of the atomic model \textit{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 \textit{primary structure} in a linear sequence (bond). Those ports represent the 3’ and 5’ carbon atoms, which are linked. In addition, a \textit{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 \textit{up- and downward causation} as provided by the formalism ML-DEVS.

3 COMPOSITION OF VARIABLE STRUCTURE MODELS

We chose the model composition framework COMO (Rohl 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 (Rohl 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 specifi-

![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.](image)

3.1 Revision of COMO

Our first revision refers to the adaption and generalization of \textit{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
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, Sub, LCon) with a unique identifier id ∈ QName, a reference to an interface definition if ∈ QName, a configuration function k, a model definition μ (implementation), a finite binding set B, a set of sub-components Sub, and a set of loose connections LCon. Sub consists of tuples (name, iid, cid, Params) with a name ∈ String, interface reference iid ∈ QName, component reference cid ∈ QName, and a set of parameter assignments Params. Each parameter assignment comprises 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 ∈ c.LCon, is defined as partial function: Sub × Sub → 2((start, end) ∈ String × 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 $R_{lc} \subseteq \text{Sub} \times \text{Sub}$.

**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: $\text{lc}(s, s') = \{3', 5'\}$ if $(s, s') \in R_{lc}$ or $\perp$ otherwise, with $R_{lc} = \{(s, s') \in \text{Sub} \times \text{Sub} \mid \exists p \in s.\text{Params}, p' \in s'.\text{Params}: p'.\text{name} = p'.\text{value} = p.\text{value} + 1 \land s \neq s' \land s.\text{iid} = s'.\text{iid} = \text{"Nucleotide"}\}$.

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 $(\text{Params}, \text{Sub}, \text{LCOn}, \varnothing)$ onto the tuple $(\text{Sub}', \text{LCOn}', \varnothing, \text{Sub}', \text{LCOn}', \text{Ports}, \text{Params}')$.

An implementation of $k$ has to evaluate given parameter values and delegate them to sub-components if necessary. Given a configuration $(\text{name}, \text{lid}, \text{cid}, \text{Params})$ that refers to a concrete component and its context, we now adapt the central function $\text{instC}(\cdot)$ 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 $\mathcal{C}$ is the set of all component definitions.

**Algorithm 1: Instantiation of a component definition.**

```plaintext
name: \text{instC}
input: name, interface reference cid, component reference iid, parameter values \text{Params}
output: instantiated composition \text{ci}

1 //get component definition
2 if \text{ci} = \text{cid} then \text{ci}' = \text{ci} else \text{ci}' = \perp
3 //check if ci valid
4 if \text{ci}' \neq \perp and \text{ci}' = \text{iid} then
5 //apply configurator of current component
6 \text{Sub}, \text{LCOn}, \varnothing, \text{Sub}, \text{LCOn}, \text{Ports}, \text{Params}'
7 = \text{k}(\text{Params}, \text{Sub}, \text{cid}, \text{LCOn}, \varnothing)
8 // instantiate recursively
9 \text{ci} = (\text{name}, \text{cid}, \text{ci}', \text{cid}, \cup_{\text{LCOn}}(\text{instC}(\text{ci})), \text{LCOn},
10 \text{B}, \text{Com}, \text{LCon}, \text{Ports}, \text{Params}')
11 with \text{ci}.\text{Com} = \cup_{\text{LCOn}}(\text{ci}'[\text{ci}.\text{name} = s.\text{name}])
12 //return instantiated composition
13 return \text{ci}
14 else
15 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 $(\text{name}, \text{cid}, \text{iid}, \mu, \text{Com}, \text{LCon}, \varnothing, \text{Com}_i, \text{LCon}_i)$ with a unique name $\in \text{QName}$, a reference to a component definition $\text{cid} \in \text{QName}$, a model definition $\mu$, a set of (sub-)component instances Com, a set of loose connections LCon, a set of bindings $\varnothing$, and the sets $\text{Com}_i \subseteq \text{Com}, \text{LCon}_i \subseteq \text{LCon}, \text{ports}_i \subseteq \text{String}$ that reflect the initial structure. $\text{Params}$ is a set of parameter assignments.

Figure 4 shows the instantiation and initialization of a composite component mRNA (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 $\text{model}(\cdot)$ transforms component instances and their implementations into executable simulation models. For this purpose, meta-models 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, $\text{mm_source}$ be the meta-model of an arbitrary source formalism, $\text{mm_mldevs}$ be the meta-model of ML-DEVS, and the function $\text{metamodel}(\mu)$ return the meta-model of the formalism in which the given model $\mu$ is defined. Then, the transformation is defined by:

$$\text{model}(\text{cid}, \text{mm_source}, \text{mm_mldevs}) =$$

$$\begin{cases} 
\text{micro}(\text{cid}, \text{metamodel}(\text{ci}.\mu)) & \text{if } \text{ci}.\text{Com} = \varnothing, \\
\text{macro}(\text{cid}, \text{metamodel}(\text{ci}.\mu)) & \text{otherwise.}
\end{cases}$$
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:

\[
C = \bigcup_{ci \in Com} \{model(c, mm_{source}, mm_{adddevs})\},
\]

with \( mm_{source} = \text{metamodel}(c, \mu) \)

\[
MC = \text{multiCouplings}(ci).
\]

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 \text{multiCouplings}(·). Please note that the generic multi-couplings in ML-DEVS are defined intensively 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, \text{matches}(r, r')\) is defined as follows:

**Definition 8.** Let \( r, r' \) be role definitions, then

\[
\text{matches}(r, r') \Leftrightarrow \{(f, t) \in \text{String} \times \text{String} | \exists e \in r.EP, e' \in r'.EP: f = e.name \land t = e'.name \land e.inp \neq e'.inp \land (e.inp \Rightarrow \text{drefT}(e'.tid) \sqsubseteq \text{drefT}(e.tid)) \land (e'.inp \Rightarrow \text{drefT}(e.tid) \sqsubseteq \text{drefT}(e'.tid))\}.
\]

\(^2\)The functions \text{drefT}(·), \text{drefR}(·), and \text{drefT}(·) retrieve corresponding definitions based on a given identifier.

**Algorithm 2:** Construction of multi-couplings.

\[
\begin{align*}
\text{name: } & \text{multiCouplings} \\
\text{input: } & \text{instance of composite component } c_i \\
\text{output: } & \text{set of multi couplings } MC
\end{align*}
\]

1. \( MC = \emptyset \)
2. for each \( lc \in Ci \) Com do
   
3.   //iterate all component pairings
4.   for each \( r' \in Ci \) Com do
5.     for each \( s' \in Ci \) Com \( r \) do
6.       //prevent direct feedback loops
7.       if \( s' = r' \) then continue with next pair
8.       //get and check interfaces
9.       if \( (i = \text{drefR}(r')) = \perp \) then //error
10.      if \( (l = \text{drefT}(r')) = \perp \) then //error
11.     //build interface instances
12.     sub = \{ name, id, uid, s.Params\}
13.    sub = \{ if name, r.id, s'.uid, s.Params\}
14.   //iterate all potential port name pairs
15. for each \( (start, end) \in lc(\text{sub.sub}) \) with \( lc(\text{sub.sub}) \neq \perp \) do
16.   //get ports and roles
17.   if \( \exists porta \in s\text{.Ports} \) with \( \text{port name} = \text{start} \) then \( p = \text{port} \)
18.   else continue with next port pair
19.   if \( \exists porta \in s'\text{.Ports} \) with \( \text{port name} = \text{end} \) then \( p' = \text{port} \)
20.   else continue with next port pair
21.   if \( (r = \text{drefR}(p\text{.rid}) = \perp \) then //error
22.   if \( (r' = \text{drefT}(p'\text{.rid}) = \perp \) then //error
23.   //match event ports of connected roles
24.   \( M = \emptyset \)
25.   if \( s = r \) then \( M = \text{matches}(r, r') \)
26.   else if \( s = r \) then \( M = \text{matches}(r, r') \)
27.   else \( M = \text{matches}(r, r') \)
28.   //iterate all event port matches
29. for each \( (from, to) \in M \) do
30.   //get names of implementation ports
31.   ip = imp[p.name from s, M]
31.   ip' = imp[p' name from s', M]
32.   //check orientation of model coupling
33.   if \( \text{drefT}(i) \neq \text{drefR}(i') \) or \( \text{drefT}(e) \neq \text{drefR}(e) \) then \( \text{MC} = (ip, ip') //keep orientation \)
34.   else if \( (s = \text{drefT}(e)) \) or \( (s = \text{drefR}(e)) \) then \( \text{MC} = (ip, ip') //invert orientation \)
35.   else do nothing
36.   return \( MC \)
The relation \( \tau \subset \tau' \) indicates that type \( \tau \) is a sub-type of \( \tau' \). Furthermore, \( \tilde{\tau} \) indicates that the direction of all event ports declared in role \( r \) are inverted, so \( \tilde{\tau} := (\tilde{\tau}, r.\delta P) \) with \( r.\delta P = \{ (e.ename, e.tid, e.inp), (e.ename, e.tid, e.inp) \} \in r.\delta P \). 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 (Pettty 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 superset of ports and sub-components 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 \( lc(s, s') \neq \perp \), for which hold \((\text{start}, \text{end}) \in lc(s, s') \exists p \in i.Ports, p' \in i'.Ports : p.\text{name} = \text{start} \land p'.\text{name} = \text{end} \land (s = ci \Rightarrow \tilde{\tau} \sim r') \land (s' = ci \Rightarrow r \sim \tilde{\tau}) \land (s \neq ci \land s' \neq ci) \Rightarrow r \sim r' \), where \( Sub = \bigcup_{ci \in Com.(ci)} \{(e.ename, e.tid, c.ci, c.Params) \}, i = dref\{i.\text{ iid} \}, i' = dref\{s.\text{ iid} \}, r = drefR(p.\text{ rid}) \), and \( r' = drefR(p'.\text{ rid}) \).

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. \( \forall p \in ci.\text{ Params} \exists p' \in i.\text{ Params} : p.\text{name} = p'.\text{name} \land p.\text{ value} = t.\text{ Car} \) with \( t = drefT(p'.\text{ tid}) \),
3. \( \forall lc \in ci.\text{ LCon} \land wellformed_{ci}(lc) \),
4. the initial structure state is valid, i.e., \( \forall ci.\text{ Com}_{i} \subseteq ci.\text{ Com}, ci.\text{ LCon} \subseteq ci.\text{ LCon} \), and \( ci.\text{ Ports}_{i} \subseteq \bigcup_{p \in i.\text{ Ports}} \{ p.\text{name} \} \) with \( i = dref\{ci.\text{ if} \} \),
5. all names of sub-components are unique,
6. \( \forall ic \in ci.\text{ Com} : complete(ic) \).

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 \( \mu \) and a set of bindings \( \mathcal{B} \) are a preserving refinement for an interface \( \mathcal{I} \), denoted by \( i \triangleright_{p} (\mu, \mathcal{B}) \), if \( \forall p \in i.\text{ Ports} \in e.\delta P : ep \in \mu'.\text{ Ports} \land t.\text{ Car} = \text{range}_{ep}(\mu'.XY) \) with \( r = drefR(p.\text{ rid}) \), \( ep = \text{ impl}(p.\text{name}, e.ename, \mathcal{B}) \), \( \mu' = model(\mu) \), and \( t = drefT(e.\text{ tid}) \).

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 \( \text{required}(\mu) \), 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 \( \mu \) and a set of bindings \( \mathcal{B} \) are a reflecting refinement for an interface \( \iota \), denoted by \( \iota \succ (\mu, \mathcal{B}) \), if \( \forall \iota p \in \text{required}(\mu) \exists \rho \in \iota.\text{Ports} \cap \rho.\text{EP} : \iota p = \text{impl}(p.\text{name}, e.\text{name}, \mathcal{B}) \) with \( \mu \equiv \text{mldevs}(\mu) \) and \( \rho = \text{drefR}(p.\text{rid}) \).

However, the question remains, how to retrieve the set \( \text{required}(\mu) \) from a given model definition. So far, the modeler has to define the set. Combining both refinements yields full refinement, denoted by \( \iota \succ (\mu, \mathcal{B}) \equiv \iota \succ (\mu, \mathcal{B}) \land \iota \succ (\mu, \mathcal{B}) \), assuring that an interface definition captures all requirements of a model definition and vice versa (Röhlf and Uhrmacher, 2008).

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

**Definition 13.** A component instance \( c_i \) is correct, denoted by \( \text{correct}(c_i) \), if

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

A correct component instance can be deployed and transformed into a proper simulation model. In contrast to (Röhlf 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 multilevelness (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


