Modeling, Identification and Control, Vol. 35, No. 2, 2014, pp. 79–91, ISSN 1890–1328

An Approach to Automated Model Composition Illustrated in the Context of Design Verification

Wladimir Schamai [1] Lena Buffoni [2] Peter Fritzson [2]

[1] *Airbus Group Innovations, Hamburg, Germany E-mail:* wladimir.schamai@airbus.com

[2] *Linköping University, SE-581 83 Linköping, Sweden. E-mail:* {lena.buffoni, peter.fritzson}@liu.se

Abstract

Building complex systems form models that were developed separately without modifying existing code is a challenging task faced on a regular basis in multiple contexts, for instance, in design verification. To address this issue, this paper presents a new approach for automating the dynamic system model composition. The presented approach aims to maximise information reuse, by defining the minimum set of information that is necessary to the composition process, to maximise decoupling by removing the need for explicit interfaces and to present a methodology with a modular and structured approach to composition. Moreover the presented approach is illustrated in the context of system design verification against requirements using a Modelica environment, and an approach for expressing the information necessary for automating the composition is formalized.

Keywords: Bindings, model composition, requirement formalization, design verification

1. Introduction

With the increasing complexity of cyber-physical systems, determining whether a particular system design fulfills or violates requirements that are imposed on the system under development can no longer be done manually and requires formalizing a requirement into some computable form. For such a formalized requirement to be verified, it will need to obtain the necessary information from the system model that is being verified, that is, it needs to be combined together with the system. In complex systems with large numbers of requirements, there is a need for an automated approach for composing the requirements with a given system design for the purpose of verification. This task is further complicated by the fact that the requirement models and the physical models are developed separately, and can operate on different quantities and data-types therefore combining them together in order to enable design verification is far from trivial.

This paper builds upon a new approach that enables automated composition of models by expressing the minimum information necessary to compose the models in the form of *bindings* (Schamai, 2013a). It presents a proposal for implementing the *bindings* concept in a Modelica-based environment (Modelica Association, 2013). In contrast to an approach that is based on defining interfaces that models have to implement, our approach enables the integration or composition of models without the need for modifying those models.

Although the concepts proposed in this paper are designed to bind any kinds of components together, we illustrate them in the context of design verification (IEEE1220, 2005; NCOSE, 2006; Kapurch, 2010). More specifically, bindings are used in vVDR (Virtual Verification of Designs against Requirements), a method that enables model-based design verification against requirements. In our examples we wish to verify a particular system design, represented by a Modelica model, against requirements that are formalized as requirement violation monitors models in Modelica.

doi:10.4173/mic.2014.2.2

The need for the bindings concept, presented in this paper, was identified in Schamai (2013a) when trying to find a way to automatically compose simulation models that can be used for design verification. The approach in Schamai (2013a) shows how natural-language requirements (see Hull et al. (2005) for examples) are formalized in order to monitor their violations during simulations. The formalization approach of requirements into monitors is inspired by concepts from run-time verification (Leucker and Schallhart, 2009).

In order to compose simulation models [1] automatically, i.e., to combine the formalized requirements (i.e., violation monitor models), system design models and scenario models, the bindings concept is elaborated in Schamai (2013a) and prototyped in the ModelicaML language (Schamai, 2013b). ModelicaML enables using UML (OMG, 2013) diagram notations for modeling complex physical system and using Modelica for simulations. It supports the model-based design verification method proposed in Schamai (2013a).

The main contribution of this paper is a proposal for leveraging the bindings concept in Modelica directly. The new feature is expected to avoid modeling errors, to reduce the manual modeling effort for integrating models, to enable automated model composition of Modelica models, as well as to establish traceability between models.

In order to illustrate the use of bindings in Modelica, we use the simplified example of a cooling system containing a number of pumps. For the purpouse of this paper, the pumps are modeled in a very simplistic manner, as we are only interested in whether the pumps are turned on or off. However, even such a simple property can be modeled in different ways for different kinds of pumps. We will use this setup to illustrate how the requirement model can be decoupled from the way that a specific property is modeled in the physical representation of the system.

This paper is organized as follows: Section 2 presents the basic concepts. Sections 3 and 4 present two different alternatives for capturing bindings and explain through the use of examples how the model composition can be automated. Section 5 discusses use of bindings in the context of the vVDR methodology and finally Section 6 sums up the work presented and discusses further research directions.

2. Basic Concepts

In this section we give the definitions of the basec concepts used in our approach, and illustrate them by using simple examples of water pumps from a cooling system. To this end we introduce the notions of *clients* and *providers*. Clients require certain data; providers can provide the required data. However, clients and providers *do not know each other* a priori.

Moreover, there may be *multiple clients* that require the same information. On the other hand, data from *several providers* may be needed in order to compute data required by *one client*. This results in a many-to-many relation between clients and providers.

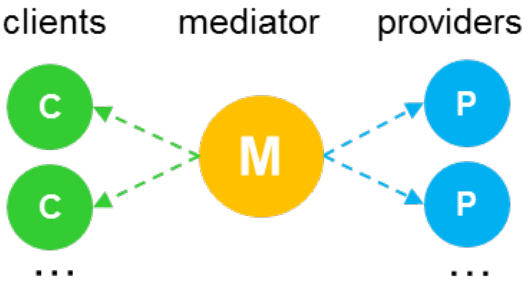

Figure 1: Concept of clients, mediator and providers

In order to associate the clients and the providers to each other we introduce the *mediator* [2] concept, which is an entity that can relate a number of clients to a number of providers, as illustrated in Figure 1. *References* to clients and providers are *stored in mediators* in order to avoid the need for modifying client or provider models.

The purpose of the presented concepts is to capture all the information needed for inferring (i.e., generating) binding expressions for clients. We define a binding as follows:

```
client instance reference = binding expression
```

A binding is a causal relation, which specifies that, at any point in simulated time, the value of the referenced client instance shall be the same as the value computed by the *right-hand expression*. When composing models, the *right-hand side*, i.e., the binding expression for the client at hand, is *inferred* using the corresponding mediator and its referenced providers.

Mediator, client or provider references may have attached template. Templates are code snippets that contain placeholders for context sensitive information. These placeholders or macros are replaced or expanded when the binding expression is generated. For example,

[1] In Schamai (2013a), requirement violation monitor models and other models are ultimately translated into Modelica models for simulations.

[2] The idea of mediators is similar to mediator pattern from the software development domain (Gamma et al., 1995), which promotes the idea of a loose coupling of objects to enable their communication without them explicitly knowing each other's details.

a template attached to a mediator can contain macros for either reducing lists of providers to a single element (e.g. min(:), max(:), sum(:), etc.) or to return arrays (e.g. toArray(:)) or information about the array (e.g. size(:)). Templates attached to client or provider references can contain code snippets for unit or type conversion or references to sub-components.

3. Alternative 1: Binding Specification in Modelica

This section presents the approach for capturing information required for an automated model composition using an extended version of the Modelica language.

Assume that we have a system that contains several pumps, and an informal requirement, such as, *"At least 2 pumps shall be in operation at any time"*. Independent of a specific system design (i.e., the number and type of pumps contained in the system model), this requirement can be formalized into a requirement violation monitor Modelica model as follows:

```
package Requirements
model Req
input Integer numberOfOpPumps = 0 "
   number of operating pumps";
constant Integer minNumberOfOpPumps
   = 2 "min. number of operating
   pumps";
output Integer status(start=0,fixed=
   true) "indication of requirement
    violation, 0 = not evaluated";
  equation
    if numberOfOpPumps <
       minNumberOfOpPumps then
      status = 2 "2 means violated";
    else
      status = 1 "1 means not
         violated";
    end if;
  end Req;
end Requirements;
```

This requirement violation monitor model has as an input `numberOfOpPumps` (that is, the number of operating pumps). When this requirement violation monitor model will be used (i.e., instantiated within another model) in simulations, this input component will need to be bound to some expression that calculates the number of operating pumps within the system for a given system design. Simulating the requirement violation monitor model is pointless without binding it to an expression that calculates this information during simulation.

Further, assume we have the following system design model that contains 3 pumps of 2 different types.

```
package Design
  model System
    PA pump1;
    PB pump2;
    PB pump3;
  end System;

  model PA
input Boolean on = false
"to turn the pump on or off";
  end PA;

  model PB
input Boolean switchedOn = false
"to switch the pump on or off";
Real volFlowRate
"volume flow rate of the pump";
  equation
    if switchedOn then volFlowRate =
        1;
    else volFlowRate = 0;
    end if;
  end PB;
end Design;
```

Additionally, there is a scenario[3] model that can be used for stimulating the system model, e.g., it will turn on and off different pumps during simulation.

```
package Scenarios
  model Scenario
output Boolean pump1active "turn on
   pump1 = true, turn off pump1 =
   false";
output Boolean pump2active "turn on
   pump2 = true, turn off pump2 =
   false";
output Boolean pump3active "turn on
   pump3 = true,  turn off pump3 =
   false";
  algorithm
    if time > 0 and time <= 10 then
          pump3active := false;
    elseif time > 0 and time <= 20
       then pump2active := false;
    elseif time > 0 and time <= 30
       then pump2active := true;
    else
      pump1active := true;
      pump2active := true;
```

[3] In Schamai (2013a) such models are referred to as *verification scenarios*. They are used to stimulate the system model such that a set of requirement violation monitors can be evaluated.

```
      pump3active := true;
    end if;
  end Scenario;
end Scenarios;
```

We create a new model (e.g. `AnalysisModel`[4]) that contains instances of the requirement monitor model (`req1`), the system model (`sys`), and the scenario model (`scen`). Now we wish to automatically find all components (i.e., clients) that require data from other components (i.e., providers) and *bind* them.

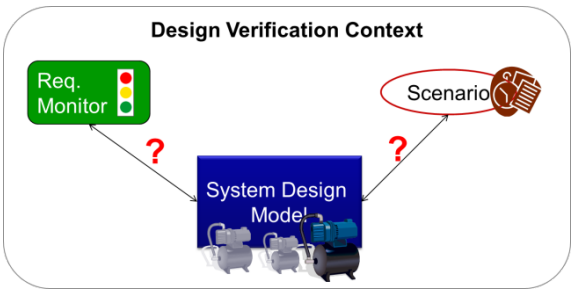

Figure 2: Integrating models for design verification

In Modelica there are two possible ways to integrate models: *acausal* and *causal*. *Acausal* connections (Modelica connect equations) are typically used for modeling physical energy or material flow. In our setting there is no need for acausal connections because requirement monitors are observers (i.e., they must not impact the system model) and scenarios stimulate the system and may observe it. *Causal* connections (Modelica connect equations, component modifiers) are sufficient for connecting requirement monitors, system design and scenarios.

Another question that we need to answer, is how we want to connect models in Modelica: using equations (that require ports of predefined compatible types to be connected) or using component modifiers (i.e., by replacing the declaration equations for input components)? Connections, i.e., connected ports with predefined interfaces, would require an extension or modification of the involved models. This approach may clutter up models with instrumentation code that is not part of the system design, requirement or scenarios. In contrast to pre-defined interfaces, component modifiers can be added when needed and adapted to the context. In the following we will apply the later approach: we will integrate models in a causal way using Modelica component modifiers. The reasons for this decision are twofold. First of all a physical model needs to be designed separately as a standalone model, and therefore having unconnected connectors that are only used in the presence of the requirement model, which would be an issue. Moreover, the binding expressions are generated automatically; therefore their complexity does not impact the user, however if we use connectors, which are in turn used to automatically generate connection equations we add an extra level of complexity to the computation model.

In our example, binding expressions shall be generated for the input component `Requirements.Req.numberOfOpPumps`, and for the pump components within the system. What we want to achieve is shown below:

```
model AnalysisModel
  Requirements.Req req1(
    numberOfOpPumps = sum({
      (if sys.pump1.on then 1 else
         0),
      (if sys.pump2.volFlowRate > 0
         then 1 else 0),
      (if sys.pump3.volFlowRate > 0
         then 1 else 0)}));
  Design.System sys(
    pump1(on = scen.pump1active),
    pump2(switchedOn =
       scen.pump2active),
    pump3(switchedOn =
       scen.pump3active));
  Scenarios.Scenario scen;
end AnalysisModel;
```

In `AnalysisModel` the component `numberOfOpPumps` of the instantiated requirement model is now *bound* to an expression that will calculate the number of operating pumps within the system during simulations by obtaining data from the system model. The system model components `pump1`, `pump2` and `pump3` are bound to the scenario model instance.

In order to do this automatically, the following information will need to be captured by experts because it cannot be deduced automatically. Experts (i.e., people that are familiar with either clients or provider models) will need to specify:

- which models or components are clients,
- which models or components are the corresponding providers,
- and, in case there is no 1:1 compatible mapping of client and provider components, what template should be used to generate the appropriate binding expressions?

In the following section we present a proposal for a Modelica concrete syntax for capturing this *minimum set of information* required to enable such automated generation of binding expressions.

[4] In Schamai (2013a) such models are referred to as verification models

3.1. Modelica Extension for Defining Mediators

As mentioned in Section 2, we suggest storing all information, required to enable inferring binding expressions, in a so called *mediator*. This concept does not exist in Modelica. All Modelica code hereafter is a new proposed extension to the Modelica concrete syntax.

In order to modularize the process, the information captured by mediators can be defined in multiple mediators that use inheritance (i.e., `extends` relation in Modelica).

Assume that in our process we have the role of a *requirement analyst*, a person in charge of elicitation, negotiation and formalization of requirements. When formalizing the requirements, the requirement analyst will define mediators in order to expose the information needed by the clients. For example, the requirement analyst would create the mediator `NumberOfOperatingPumps_C` and associate the component `numberOfOpPumps` (the input component of the requirement violation monitor model) to it as follows:

```
package PartialMediators
  mediator NumberOfOperatingPumps_C
     requiredType Integer;
     clients
      mandatory Requirements.
        Req.numberOfOpPumps;
  end NumberOfOperatingPumps_C;
end PartialMediators;
```

Note, that the only way to reference a client is to use its *model qualified name* (e.g., `Requirements.Req.numberOfOpPumps`). The model `Requirements.Req` may be used in different context, i.e., instantiated in another model and will be given an *instance path* (e.g., req1.numberOfOpPumps in the context of `AnalysisModel`). However, we cannot know the instance path a priori.

So far, the mediator only contains references to *clients*, i.e., models or components that require the information, however, no description of how to get this information yet. In that sense this mediator is *incomplete*.

For the sake of simplicity, there is only one client model that requires the number of operating pumps within the system in our current example. In a larger example there are likely to be more models (e.g. other requirement violation monitors) that will also need the same information. In that case there will still be only one mediator that would then contain references to more clients in the `clients` section. This way, mediators allow a *grouping* of models or components *that require the same information* and enable a concise definition of bindings.

This mediator also indicates the type of data to be provided to the clients, the `requiredType` reference, which must be compatible with the type of all associated clients. This way, the mediator reflects what is needed by clients. There is no need to analyze each client anymore, it is sufficient to only look at mediators. This is especially useful if a different person, with no knowledge of the requirement model is in a charge of defining the providers in the model.

Moreover, the client reference can have the prefix `mandatory`, to indicate that this client must be bound[5]. In our example it is the input of the requirement violation monitor model that has to receive the corresponding value during simulation.

At some point in time, another person, e.g., *system designer* will specify which models can provide the information required by clients and how to compute it from a particular *provider model*. As mentioned above, for doing so, system designer will only need to look at mediators. There is no need to analyze the referenced clients (which may be many in larger models) because the mediator unambiguously reflects the content and the type of required data.

To this end, system designer creates a new mediator `NumberOfOperatingPumps_P`, that extends the mediator `NumberOfOperatingPumps_C`. By using inheritance, the new mediator obtains all client references. Now system designer can add references to provider models and specify how the binding expression should be generated in case there is no 1:1 mapping between clients and providers.

```
mediator NumberOfOperatingPumps_P
  extends PartialMediators.
      NumberOfCaviatingPumps_C;
  template sum(:) end template;
  providers
     Design.PA.on
        template if getPath() then
            1 else 0 end template
           ;
     Design.PB.volFlowRate
        template  if getPath() > 0
            then 1 else 0 end
           template;
end NumberOfOperatingPumps_P;
```

If there is no 1:1 mapping, in addition to the client or provider references, templates can be used to specify how the expression code should look like and where to insert context-sensitive data.

For example, a template can be attached to the mediator in order to either specify how to aggregate in-

[5] An example for a not mandatory client is shown at the end of this section.

formation in case several providers are in place, or to specify that it only should contain constant data.

In our example, the mediator will be used to generate an expression that will calculate the number of operating pumps for any design, i.e., for any number and type of pumps contained in a particular system design model. The strategy for computing this information in this example is the following. Each provider model (i.e., a model of a pump) shall indicate whether the pump is in operation by returning 1 if it does and 0 otherwise. Then the *sum*(:) operator, specified in the mediator `"template` *sum*(:) `end template;"`, will be turned into an expression to calculate the total number of operating pumps in the system during simulations.

The colon in *sum*(:) indicates that this operator expects an array of unordered items. In our example *sum*(:) will be mapped to the Modelica array reduction function *sum*(A) where A is an array of expressions each calculating whether a pump instance is in operation. Note that such expressions may be different for different pump model instances within the system model at hand. In general, the mediator templates are allowed to include any template function that accepts as input an unordered list.

Now, for each referenced provider model of a pump (i.e., in our example there are `Design.PA` and `Design.PB`), we specify how to determine whether the pump is in operation using the provider template. Based on the strategy of the mediator, the following template for the provider model component Design.PA.on `"if getPath() then 1 else 0"` specifies that 1 is returned if the pump is in operation and 0 otherwise. This is different for the second pump type. Here we will be using the component `Design.PB.volFlowRate` and the template `if getPath() > 0 then 1 else 0`.

For referencing provider model sub-components, the `getPath()` operator is used. It is a placeholder that will be replaced[6] by the instance path of the provider model in a particular context (i.e., in our example in the context of `AnalysisModel`, see below).

The mediator `NumberOfOperatingPumps_P` is *complete*. It contains all the information needed to automatically generate the binding expression for all referenced clients[7]. Note that this mediator is independent from the number of pump instances in a given design. As long as the referenced provider models (i.e., `Design.PA` and `Design.PB`) are used in a system design model, this mediator can cope with any number of pump instances (e.g., 20 pumps instead of 3 pumps in our example model `Design.System`).

The inferred binding expression will then be passed[8] to each client that requires the information about the number of pumps that are in operation during a simulation run.

Also, note that the purpose of creating another mediator is to show that this approach can be used for separating concerns and making definitions reusable. In the same way, we could use the first mediator `NumberOfOperatingPumps_C` and add the missing information to it. Separating concerns may be necessary because different people will be the owners of different mediator models, or because the same mediator, that contains client references, may be reused for different designs with provider models specific to the design at hand. Figure 3 summarizes the information that needs to be captured in abstract syntax.

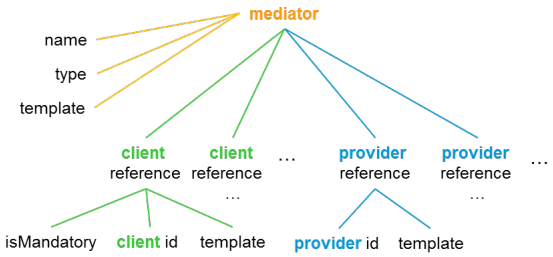

Figure 3: The information that needs to be captured in abstract syntax.

The mediator name[9] reflects what is needed by clients. The mediator `requiredType` must be compatible to each of its clients[10].

Client or provider id is the *qualified name* of the client or provider model or component (e.g. `Requirements.Model1.comp1`). The attribute `mandatory` (`true` by default) indicates whether the client must be bound. If not, the client component must have a default value.

All templates are optional. For example, if there is only one provider that returns exactly what is needed by clients, there will be no need for neither the mediator template nor for the provider template.

In contrast, if the binding expression will refer to several provider models for which the code for accessing the right data is different (like in our running example), then probably a mediator template and a provider template will be necessary. A client template will be

[6] The `getPath()` operator will be replaced with the instance paths of providers when the binding expression will be generated for a particular client.

[7] Recall, in our example we associated clients in the `NumberOfOperatingPumps_C` mediator that is extended by `NumberOfOperatingPumps_P`

[8] In our proposal the generated binding expression will be passed via the Modelica component modifier to the client component.

[9] Comments may be included like for any Modelica model.

[10] In addition, the mediator should indicate the lowest variability of its clients (this is not discussed in this paper).

needed as soon as the actual client is not the referenced client model but a sub-component of it.

Client or provider templates are expressions that can contain instance paths (e.g. in Modelica using the dot-notation) for referencing sub-components within the client or provider models. A mediator template can only contain predefined macros or built-in functions (e.g. $sum(:)$, $toArray(:)$, $card(:)$, $min(:)$, $max(:)$, etc.) or constant data.

A client template is needed in order to enable pointing to sub-components. For example, in our system model there are several pumps that can be turned on or off. System designer can expose the potential stimuli of the system by creating a new mediator and associating the clients as shown below. The provider references will be added by the tester who will be creating the scenario models. Now, the client template in the mediator `Pump1IsOn` is necessary because we need to point) a particular pump instance within the system model, in this case the pump1 component within `Design.System`.

```
package Mediators
...
mediator Pump1IsOn
      requiredType Boolean;
   clients
      Design.System.pump1
         template getPath().on =
            getBinding();
         end template;
    providers
      Scenarios.Scenario.pump1active
         ;
end Pump1IsOn;
...
end Mediators;
```

If there will be more scenario models that turn on and off the pump1, the mediator will still be the same and will only need to include the additional scenario model references in the `providers` section. Similarly, if there is a new design alternative, say one that contains 20 pumps of the same type instead of 3 like in the example above, the bindings will still be generated correctly. If there will be another type of pump (i.e., other than the two types from the example above), then we will merely need to add a new provider reference to the mediator.

To sum up, in order to integrate the binding concept into the Modelica language, we define a new type of class, mediator, which has a particular structure. However the scope of these extensions is limited, as all the extensions are confined to within the mediator, and moreover, once the bindings are generated the models only contains standard Modelica, which means that they will be compatible with any Modelica tools.

3.2. Generating Binding Expressions

In this section we illustrate how we use the information contained in the mediators to automatically generate binding expressions.

3.2.1. Templates

Let us first come back to the question why we need client, mediator, and provider *templates*. A template defines the form of the binding expression to be generated. It specifies where to insert context-sensitive information, such as the instance path of components (i.e., which will replace the placeholder `getPath()`) in order to enable pointing to particular client or provider subcomponents.

The purpose of the mediator template is to specify how to reduce arrays to a single value, or to specify that constant data should be passed to the associated clients. Client and provider templates are primarily used to enable pointing to sub-components. Client templates are also used for overwriting binding definitions. A consideration of possible cases and general validation rules for templates can be found in Schamai (2013a). They are used for generating valid bindings. A *binding* is said to be *valid* if the binding expression can be inferred for client ci_k and the resulting type of the right-hand binding expression is compatible with the left-hand-side expression.

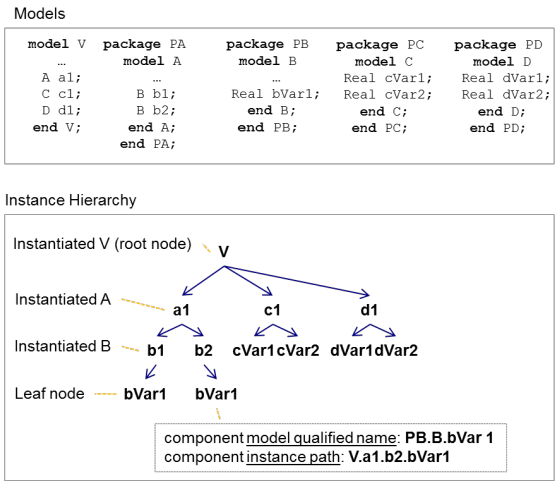

Figure 4: The information that needs to be captured in abstract syntax.

3.2.2. Instantiation Tree

Let us first now introduce a structure, called *instantiation tree*, which is used for inferring binding expressions. It is a tree that starts with the root node representing the model being instantiated (see Figure 4).

Each child node represents a component of the parent model. The recursive tree construction stops at leaf nodes. Leaf nodes represent components of primitive types, which do not have any further internal structure. Figure 4 shows an example of Modelica models[11] and the correspondent instance hierarchy.

Model qualified name is the path (structured name) of a model element - e.g. a class or an attribute of a class - that identifies the element within the structure used to organize the model (e.g., by means of packages or nested classes). Instance path identifies a component within an instantiated model (for example, in our setting, within `AnalysisModel`).

Each tree node contains all the relevant information about the element (e.g. the component model qualified name, component instance path etc., see Figure 4). They are necessary in order to match clients and provider instances based on the model qualified names within mediators.

3.2.3. Algorithm

In `AnalysisModel`, we first import mediators that should be used for generating the binding expressions (i.e., we add an import clause import `Mediators.*` to import the mediators `Mediators.NumberOfOperatingPumps_P`, `Mediators.Pump1IsOn`, etc.).

Now, in order to generate binding expressions for each client in `AnalysisModel` we trigger the new tool feature "Update bindings". Figure 5 below shows how such an invocation could look in a Modelica tool[12].

The algorithm will first create an instantiation tree[13] and collect all client components, mediators to be used for inferring binding expressions and all the referenced provider components contained[14] in `AnalysisModel`.

While creating the instantiation tree (see Figure 6) the algorithm takes the imported mediators into account in order to identify (i.e., match) nodes (i.e., components) that are *clients* or *providers*. Clients and

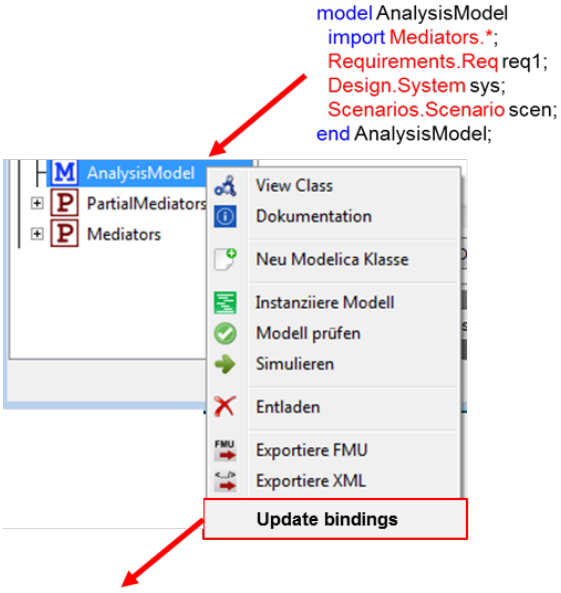

Figure 5: The information that needs to be captured in abstract syntax.

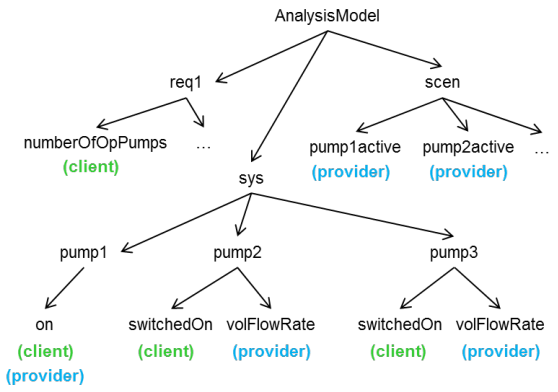

Figure 6: The information that needs to be captured in abstract syntax.

[11] In Modelica the primitive types, such as Real, Integer, String, and Boolean, still have one level of internal structure of predefined properties, see Modelica Association (2013).

[12] Such as the OpenModelica graphical editor OMEdit [4]

[13] The algorithm for constructing the instantiation tree is not shown in this paper. It is a traversal that is straight forward to implement.

[14] Note that a mediator may contain much more client or provider references. However, now, in a specific context, the algorithm will only consider those that are contained in the model at hand.

providers are identified by comparing the *model qualified name* of each component node in the instantiation tree with client and provider references of the imported mediators.

For example, `req1.numberOfOpPumps` is a client because there is a mediator (`Mediators.NumberOfOperatingPumps_P1`) that references this component using its model qualified name (`mandatory Requirements.Req.numberOfOpPumps`). Moreover, it is a mandatory client. This client must be bound to some expression. This means that if no binding expression can be inferred for that client, an error should be reported.

Now the algorithm for inferring the binding expression can be triggered. It is described in pseudo-code in the Appendix [15]. The algorithm requires as input an instantiation tree node which is a client, and the set of mediators to be used for inferring binding expressions.

Mediator that contains client references:

```
mediator NumberOfOperatingPumps_C
  requiredType Integer;
  clients
    mandatory Pack1.Req.numberOfOpPumps;
  ...
```

Mediator that also contains provider references:

```
mediator NumberOfOperatinggPumps_P1
  extends NumberOfOperatingPumps_C;
  template sum(:) end template;
  providers
    Pack2.PA.on
      template if getPath() then 1 else 0 end template;
    ...
```

Model with updated bindings:

```
model AnalysisModel
  import Mediators.*;
  Pack1.Req req1(
    numberOfOpPumps = sum({
      (if sys.pump1.on then 1 else 0),
      (if sys.pump2.volFlowRate > 0 then 1 else 0),
      (if sys.pump3.volFlowRate > 0 then 1 else 0)}));
  ...
end AnalysisModel;
```

Figure 7: The information that needs to be captured in abstract syntax.

The algorithm first identifies the actual client[16] and finds the corresponding mediator. Then, based on the mediator template, the associated providers (e.g. `sys.pump1.on`, `sys.pump2.volFlowRate`, etc.), and the provider templates (e.g. `template if getPath() then 1 else 0 end template` for provider model component `Design.PA.on`) the algorithm tries to generate the binding expression.

For the requirement model component client `Requirements.Req.numberOfOpPumps` the binding expression can be inferred. Figure 6 explains which parts of the mediator specification were used to generate which parts of the binding expression that is passed to the client via Modelica component modifier (see component modification for `numberOfOpPumps` in `req1`).

Note that all placeholder occurrences of `getPath()` are now replaced by the corresponding instance path (e.g. `"sys.pump1."`) of the corresponding components within `AnalysisModel`. The mediator template template `sum(:) end template`; is mapped to the Modelica built-in function $sum(A)$ where A is the array of values, which in our example results from providers and expressions generated based on their templates.

Finally, the inferred binding expression is passed to the client (e.g. to the instance of `Requirements.Req.numberOfOpPumps`) via the Modelica component modifier in its first-level component (i.e., `req1`). The updated `AnalysisModel` is shown in Figure 5.

4. Alternative 2: Binding Specification in XML

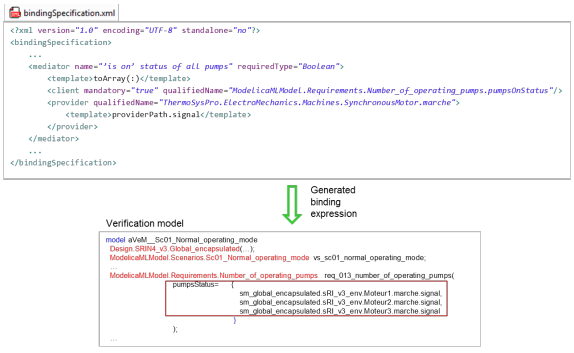

Figure 8: Bindings specification is used to generate binding expressions.

In this Section we present an alternative approach for capturing binding specification. In contrast to the approach presented in Section 3, instead of writing the bindings specification (i.e., the mediator classes) in an extended version of Modelica, we capture this information using XML (W3C, 2013).

[15] In Schamai (2013a)] templates are referred to as operations.

[16] When looking for the client c there may be component at a higher hierarchy level that uses its client template to point to it . This is the actual client because, similar to Modelica *component modification* mechanism, the higher-level components overwrite definitions for lower-level components.

Assume we have the following requirement *"013 - When the system is in operation, there should be no less than two pumps in operation for more than 2 seconds"*. The Modelica requirement violation monitor may have the following inputs:

```
model Number_of_operating_pumps
 input Boolean isNormalOperatingMode
    ;
 input  parameter Integer
    numberOfPumps=0;
 input Boolean pumpsOnStatus[
    numberOfPumps];
...
```

Consider the input `pumpsOnStatus[:]`. It is an array of type Boolean. It's size is equal to the number of pumps within the system. Each array entry indicates whether the pump is in operation (i.e., = true) or not (i.e., = false). This array will need to be constructed based on the system model at hand, i.e., based on the pumps within the system, and passed to the instance of the requirement violation monitor model above.

Figure 8 shows an example of an XML file that captures the bindings specification (only for the part relevant for our example). Inferring of binding expressions would work same as for the Modelica-based version described in Section 3. However, instead of parsing the extended version of Modelica here the tool will need to parse XML. The advantage of this approach is that there is no need for extending the Modelica language. The only thing to be agreed on is the XML Schema, i.e., the structure of such XML files.

5. Further Applications for Bindings

This section mentions one possible application for using the presented bindings concept. Assume that we wish to combine a given system design model with a scenario model and a set of requirements the design shall be verified against. Such a combination is a new model, called *verification model*.

To do so, first, we need to find combinations of one *system design model*, one *scenario model*, and a *set of requirements*. Thereby, we want to ensure that only those scenarios should be considered which can stimulate the system model. In addition, we want only to consider requirements that are already addressed in the design.

Figure 9 illustrates[17] how bindings can be leveraged to create such verification models. It starts with the one design alternative selected by the user. Then it iterates over all found scenarios[18].

For each of the scenarios it checks whether all mandatory clients of the design and this particular scenario model can be satisfied. If yes, the scenario is selected because we can assume that it will stimulate the system design model and it will receive all necessary feedback from the design model. If not all mandatory clients can be satisfied, then this scenario is discarded.

Satisfying a client means that based on the given set of mediators and providers in place, for this client it is possible to infer a valid binding expression.

Further, having selected a scenario, the algorithm iterates over all requirements that are referenced by this particular scenario with a relation indicating that this scenario can be used to verify designs against this particular requirement. For each requirement, again the algorithm checks whether all mandatory requirement violation monitor clients can be satisfied. If yes, the requirement is added to the combination. If no, then we assume that this requirement is not completely or not correctly addressed in the given design and cannot be evaluated yet.

An outer-loop iteration terminates with a selected scenario and one or more requirements. This combination can be translated into a verification model by instantiating the design, selected scenario, and requirements and passing the inferred bindings to clients.

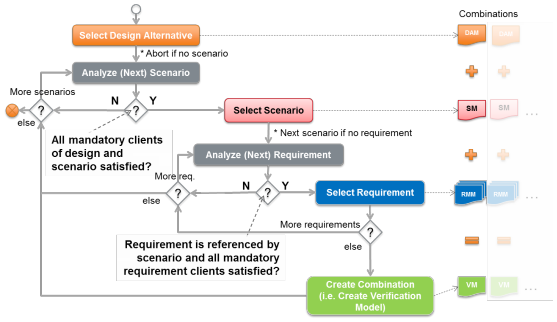

Figure 9: : Illustration of the verification models generation algorithm.

6. Conclusion

In this paper we have presented a new concept that enables the automation of Modelica model composition. We have proposed two different approach to support the process of expressing the information necessary to

[17] Pseudo code for this algorithm can be found in Schamai (2013a)

[18] Scenarios are models that are annotated to be a scenario. Such, for example, Modelica models stimulate the system model during simulation in order to enable evaluation of requirement violation monitors.

enable the automated generation of binding expressions: one based on an extension version of the Modelica language and one based on XML. We illustrated the concept on examples in the context of design verification. In particular, we have shown how bindings of components can be generated for a given Modelica model.

This approach does not rely on an interface mechanism and therefore increases the decoupling of the models as it does not require prior knowledge of the interfaces by the models. Just as with classic interfaces, the binding concept respects the encapsulation principle, and only the information that is displayed publicly by the model can be bound. If a private attribute of a system model is required in order to obtain the information required for the composition of the models, then the model should be rethought, and eventually the required information made public. Furthermore, the possibility to define mediators in several steps means that the information can be provided by different people at different stages of the design process resulting in a more flexible and modular approach.

The advantages of using such an automated generation of binding expressions are the following:

1. Exposing and grouping of information about what data is needed by clients will reduce analysis work because the number of mediators will be smaller than the number of clients. The more clients will require the same information the greater will be the gain in terms of information reuse.

2. Automated generation of binding expressions will reduce modeling errors and the manual modeling effort. This is in particular true for models with highly interrelated components, or for complex binding expressions (e.g., Modelica component modifiers) too complicated to be written manually.

3. The binding concept enables a number of applications. For example, it enables automated composition of verification models from Schamai (2013a). Furthermore, it enables a formal traceability between client and provider models. For example, to determine which requirements are implemented in the system design model at hand, can be achieved by looking at the bindings for mandatory requirement clients.

References

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

Hull, E., Jackson, K., and Dick, J. *Requirements Engineering*. Springer, 2005.

IEEE1220. Ieee standard for application and management of the systems engineering process. IEEE, 2005.

Kapurch, S. *NASA Systems Engineering Handbook*. DIANE Publishing Company, 2010. URL http://books.google.se/books?id=2CDrawe5AvEC.

Leucker, M. and Schallhart, C. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 2009. 78(5):293 – 303. doi:10.1016/j.jlap.2008.08.004. The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS07).

Modelica Association. Modelica 3.2 revision 2 specification. 2013. URL www.modelica.org.

NCOSE. *Systems Engineering Handbook (Version 3 ed.)*. INCOSE., 2006.

OMG. Object Management Group (OMG). 2013. URL www.omg.org.

Schamai, W. *Model-Based Verification of Dynamic System Behavior against Requirements*. Ph.D. thesis, Method, Language, and Tool Linköping: Linköping University Electronic, PressDissertations, 1547, 2013a.

Schamai, W. ModelicaML - UML Profile for Modelica. 2013b. URL www.openmodelica.org/modelicaml.

W3C. Extensible Markup Language (XML). 2013. URL www.w3.org/XML.

Appendices

A. Algorithm for Generating Binding Expressions

Algorithm: $infer Binding(ci, M\_S)$: Infer binding for a client

input : Client ci (node in instantiation I) for which binding is to be inferred
Set of mediators M_S that should be used

output: *bindingExpression* (binding expression, i.e., the right-side expression)
isBindingPossible (is possible to infer binding including manual decisions?)
ci_a (actual client used, i.e., ci or one of its parents in I)
m_a actual mediator used
$p_0 ... p_n$ set of providers referenced by m_c and contained in I

```
1  begin
       // Set all to undefined or false
2      isBindingPossible = false;
3      bindingExpression, ci_a, m = null ;                    // Set all to null (meaning they are undefined)
4      dataCollection = empty ;                                                          // Empty list
       // Get the instantiation (starting from root node) that contains ci
5      I = getInstantiation(ci) ;
       // Find the actual client (ether ci itself or an upper level node in I). Note that upper level client
          templates may over-write bindings from lower levels.
6      {n_0...n_n = ci} = getParentsTopDown(ci) ;                  // Get parents top-down including ci
7      for n_k in {n_0...n_n} do
8          m_0...m_n = getMediators(n_k, M_S) ;                  // Ordered list of mediators that reference n_k
           // For each n_k find those mediators with client references containing specifications for ci.
9          for m_k in {mk_0...mk_n} do
10             nco_k = getClientTemplate(m_k, n_k) ;                          // Get client template for n_k
               // If the left-side expression (placeholder replaced with instancePathOf(n_k)) is equal to
                  instancePathOf(ci) → then select. If there are there multiple matches, select only the
                  topmost entry because this overwrites lower entries. If we reached ci then select ci to be
                  the actual client.
11             if specifiesBindingFor(nco_k, instancePathOf(ci)) or n_k == ci then
                   // Check whether the client template and client reference are valid. A client template
                      must contain the placeholder (e.g. getPath()) to be replaced by a concrete client
                      instance path in I.
                   // If n_k is ci the client template is discarded because it can only specify lower-level
                      clients, which are not of interest at this point.
12                 assert (isValidClientTemplate(nco_k)) → abort otherwise
13                     "INVALID: Client template nco_k, attached to n_k specifying ci is not valid."
                   // Ensure that there is only one reference from mediator to client.
14                 assert (getClientReferences(m_k, n_k) ==1) → abort otherwise
15                     "INVALID: m_k references client n_k several times."
                   // Continue if asserts are true.
16                 ci_a = n_k ;                                                  // Actual client found.
                   // Get providers (and their templates) that are referenced by m_k and contained in I.
17                 {p_0...p_n}, {o_p0...o_pn} = getProviders(m_k, I) O_m_k = getMediatorTemplate(m_k)
                   // Mediator template can be empty or can contain reduction function macros.
18                 assert (isValidMediatorTemplate(O_m_k, sizeOf({p_0...p_n}))) → abort otherwise
19                     "INVALID: Mediator template O_m_k, of m_k is not valid for inferring binding for ci."
                   // Continue if asserts are true.
20                 dataCollection.add(ci, ci_a, o_ci_a, m_k, o_m_k, {{p_0...p_n}, {o_p0...o_pn}}) ;   // Add a raw to
                   dataCollection.
                   // It will be possible to infer binding assuming manual selection.
21                 isBindingPossible = true;
22             end if
23         end for
           // End of iteration over mediators.
24     end for
90     // Top-down iteration looking for c_ia.
```

25

// Check that there is only one entry, i.e., only one valid mediator.

26 **assert** *(sizeOf(dataCollection) == 1)* → **abort otherwise**

27 or **ask for manual decision** "Select a mediator from {...} for inferring binding for ci."

// Check whether the mediator template can handle multiple providers or if providers should be selected manually.

28 **if** *sizeOf*($\{p_0...p_n\}$) > 1 ***and not*** *(isMultiProviderMediatorTemplate(*O_{m_a}*) or preferredBindingExists(ci,* $\{p_0...p_n\}$*))* **then**

29 **abort** or **ask for manual decision** "Select provider from $\{p_0...p_n\}$ for inferring binding for *ci*." ;

30 **optionally** store the selection in **preferred bindings** by using instancePathOf(*ci*), modelQualifiedNameOf(*ci*) and instancePathOf(p_k), modelQualifiedNameOf(p_k)

31 **end if**

// Continue if assertions are true.

// If there is only one mediator, or it was selected manually, the first row contains all relevant data.

32 $ci_a, o_{ci_a}, m_a, om_a, \{\{p_0...p_n\}, \{o_{p0}...o_{pn}\}\} \leftarrow$ the first row in dataCollection

// A provider template may be empty or must contain placeholder (e.g. getPath()) for future provider instance path otherwise.

33 **assert** *(areAllValidProviderTemplates(*$\{\{p_0...p_n\}, \{o_{p0}...o_{pn}\}\}$*)* → **abort otherwise**

34 "Selected providers ... have invalid templates ..."

// If the actual client and the mediator are found and it is possible to infer binding.

35 **assert** *(isDefined(*ci_a*) and isDefined (*m_a*) and isBindingPossible)* → **abort otherwise**

36 "No binding could be inferred for *ci*."

// Continue if assertions are true.

// At this point we have found the actual client, mediator, providers, and all templates. Now we can generate the binding expression.

// In all provider templates, replace the placeholder with the concrete provider instance path in I.

37 $\{ot_{p0}...ot_{pn}\}$ = translateProviderTemplates($\{p_0...p_n\}$)

// Expand reduction function macros in the mediator template.

38 ot_m = translateMediatorTemplate($m_k, \{ot_{p0}...ot_{pn}\}$)

// Replace the placeholder with the client instance path in *I*; replace the placeholder with the inferred binding expres-sion.

39 *bindingExpression* = translateClientTemplate(ci_a, o_{ci_a}, ot_m)

40 **return** *bindingExpression, isBindingPossible,* ci_a*,* m_a*,* $\{p_0...p_n\}$

41 **end**

Any `abort` in the algorithm above means that the function returns the current status of the outputs. Functions such as `isValidClientTemplate(...)` test the validity of the template as described in Schamai (2013a). The `preferredBindingExists(...)` function (not explained in this paper, see Schamai (2013a) for more details) determines whether, in the given set of providers, there is one provider that should be used for the given client.