ModelicaML Value Bindings for Automated Model Composition

Wladimir Schamai[1], Peter Fritzson[2], Christiaan J.J. Paredis[3], and Philipp Helle[4]
[1, 4] EADS Innovation Works, Germany, {wladimir.schamai, philipp.helle}@eads.net
[2] Department of Computer and Information Science, Linköping University, Sweden, petfr@ida.liu.se
[3]Georgia Institute of Technology, Atlanta, USA, chris.paredis@me.gatech.edu

Keywords: ModelicaML, design verification, Modelica, model composition, model generation.

Abstract

vVDR is a method for model-based system design verification against requirements. This paper discusses enhancements to the vVDR method and its implementation in ModelicaML to further improve the support of system verification activities by automation. In the vVDR method there are different kinds of models that are created independently and that will become dependent and need to be related to each other in some concrete verification usage. The aim is to reduce modeling errors and modeling effort by automatically generating composite verification models from their constituting sub-models based on data dependencies that are defined using so-called mediators, which allow expressing data dependencies between models without affecting, i.e., changing, the models themselves.

1. INTRODUCTION

vVDR (Virtual Verification of Designs against Requirements) is a method that enables a model-based design verification against requirements. The first version of the vVDR method and an example of its application is illustrated in [3] using ModelicaML[4]. ModelicaML integrates a subset of the UML[2] and the Modelica[1] language. It supports all Modelica constructs and, in addition, supports an adopted version of the UML state machine and activity diagrams for behavior modeling as well as UML class composition diagrams for structure modeling. This enables engineers to use the simulation power (i.e., solving of hybrid differential algebraic equations) of Modelica combined with a standardized graphical notation for creating system models.

1.1. vVDR Method

The main vVDR method steps are:

1. **Select Requirements**
   This step explains how to determine which requirements can be verified using this method.
2. **Formalize Requirements**
   This step explicates how to formalize requirements for the design verification purpose.
3. **Select or Create Design Model to be Verified against Requirements**
   This step clarifies what properties a system design model needs to have for being suitable for this method.
4. **Select or Create Verification Scenarios**
   This step describes what the properties of a verification scenario are.
5. **Create Verification Models**
   This step explains what a verification model consists of.
6. **Execute Verification Models**
   This step puts requirements on the simulation output.
7. **Analyze Results**
   This step provides guidance for analyzing possible causes of inconsistencies found in the simulation results.

This paper explores the opportunities for automation in different vVDR steps. More precisely, it discusses the necessary enhancements of the vVDR method, and its implementation in ModelicaML, in order to facilitate automation and, hence, to reduce modeling errors, time and effort.

1.2. ModelicaML

ModelicaML is a UML profile and a language extension for Modelica. The main purpose of ModelicaML is to enable graphical system modeling using the standardized UML notation together with the modeling and simulation power of Modelica. This is in line with the Object Management Group's (OMG) efforts to define a formal transformation between Modelica and SysML [6, 8]. ModelicaML defines different views (e.g., composition, inheritance, behavior) on system models. It is based on a subset of UML and reuses some concepts from SysML. ModelicaML is designed to generate Modelica code from graphical models. Since the ModelicaML profile is an extension of the UML meta-model it can be used as an extension of both UML and SysML. A tool suite for modeling with ModelicaML and generating Modelica code can be downloaded from [4].

1.3. Problem Definition

A key challenge during any model-based systems engineering process is verifying that the prescribed objectives and requirements are met by a particular system design alternative[7]. vVDR answers to that and identifies artifacts that are required for a model-based design verification and defines how they should be formalized. It introduces the concept of a *requirement model*, a *design alternative model* and a *verification scenario*[1,2]. Each of these models is needed in order to create a *verification model.* In a scenario-based approach, a *verification model* will comprise one design alternative that is to be verified against a set of requirements by running one verification scenario. Moreover, some additional models may be required. For example, a dedicated calculation model might be needed when the required data cannot be provided by the design model if such calculation is not part of the design.

Let us generalize this approach by assuming that there are multiple models that depend on each other's data when they are used together. For example, requirement models, system design alternative models, test case or verification scenario models, configuration models, system cost models, system weight models, trade-off or measure of effectiveness models, etc. When used together, some of these models will require data from other models and may provide data for other models. For example, each *requirement model* needs data from the *design alternative model* which is verified in order to determine requirement violations. *Verification scenario models* provide stimuli for *design alternative models* and might require feedback data in order to complete the scenario.

Even though all models will potentially depend on other models in a given usage scenario, they may be created without the knowledge of these other models. For example, the formalized *requirement models* will probably be created before the system *design alternative models*. In turn, *design alternative models* will most likely be created before or in parallel to the *verification scenarios*.

1.4. Research Questions

Assuming that the models are created independently but become dependent when used together, the following questions arise:

- **What is necessary to determine if and what data a model needs from other models?** For example, how to determine what requirements are implemented in the design alternative model and thus are ready to be verified? Moreover, if a requirement is imposed on a component class that is instantiated multiple times in the design, is it possible to infer the number of needed requirement instantiations?
- **What is necessary to determine which other models can provide the data required by a model?** For example, how to determine if a verification scenario provides the mandatory stimuli to the system design model?
- **What is necessary to determine a valid combination of models in which all required binding data can be automatically deduced?** For example, how to determine which verification scenario can be used to verify which requirements by providing the necessary stimuli to the system design alternative model?
- The bottom line of the questions above is the question whether **it is possible to bind all models and their components automatically?** Applied to the vVDR method the question is: is it possible to determine a valid combination of one system design alternative model, a set of requirement models, one verification scenario, and other required models, and to automatically generate a verification model that contains all of these components and the code that binds them together correctly as depicted in Figure 1?

[1] This is only required if the verification is performed based on predefined scenarios (e.g. usage scenarios, misuse scenarios, stress scenarios, etc.).
[2] Scenarios are not needed when model-checking is used because in order to provide a formal proof all possible scenarios will be tested automatically.

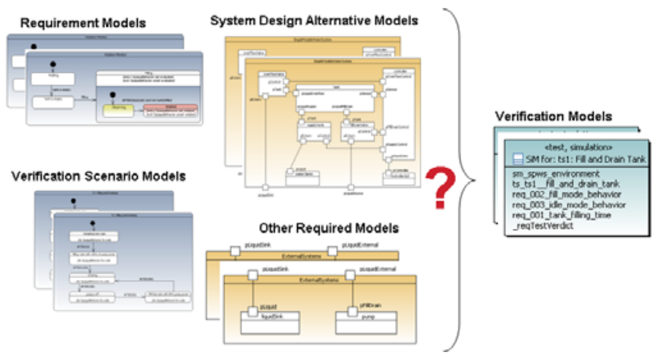

Requirement Models
System Design Alternative Models
Verification Scenario Models
Other Required Models
?
Verification Models
«test, simulation»
SIM for: ts1: Fill and Drain Tank
sm_spws_environment
ts_ts1__fill_and_drain_tank
req_002_fill_mode_behavior
req_003_idle_mode_behavior
req_001_tank_filling_time
_reqTestVerdict

Figure 1: Different models form a Verification Model

1.5. Running Example

We use a simplified potable water system, `spws`, as the running example in this paper. The system, as illustrated in Figure 2, is composed of a tank that can be filled from the bottom, several valves that are attached to the tank, a sensor to determine the level of liquid in the tank and a controller for controlling the valves.

Let us assume that the following requirements are imposed on this system:

- R001: *The time to fill an empty tank shall be 300 sec. max.*
- R002: *When the system is not used (e.g. is not being filled, drained or is not supplying water to clients) all valves shall be fully closed.*

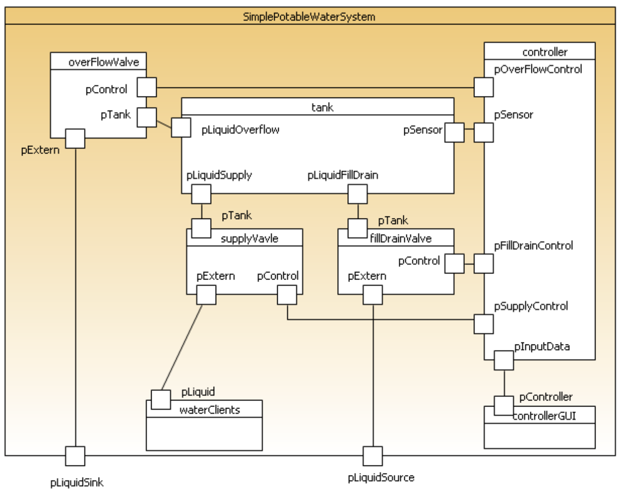


Figure 2: Simplified Potable Water System

When using vVDR, a requirement is formalized by identifying measurable properties and relating them in order to determine when this requirement is violated. For example, requirement R001 is formalized as follows: Figure 3 shows the formalized properties of the requirement that are addressed in its textual statement. Figure 4 shows how a violation of this requirement is determined. The violation monitor will start its evaluation by observing when the tank is empty. After that, it will wait until the filling of the tank starts. If these preconditions are met, the determination of the violation can be done by checking if the tank is full after the maximum allowed time of 300 sec.

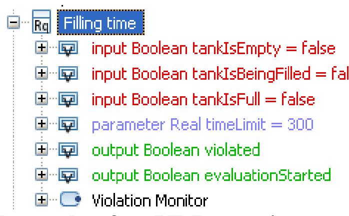

Rq Filling time
input Boolean tankIsEmpty = false
input Boolean tankIsBeingFilled = false
input Boolean tankIsFull = false
parameter Real timeLimit = 300
output Boolean violated
output Boolean evaluationStarted
Violation Monitor

Figure 3: Example of a vVDR requirement properties formalization

Please note that this requirement is independent of a specific system design model. It can be reused for any individual design, for example, with different numbers or types of tanks, valves and a different controlling strategy, assuming that the identified inputs are provided by the design model to be verified.

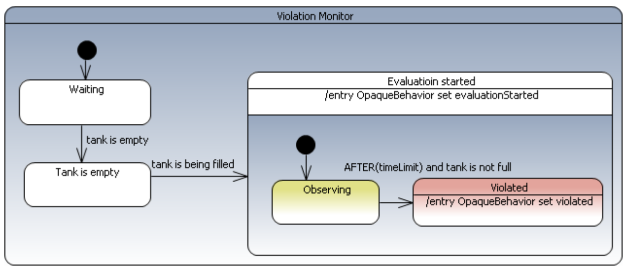

Violation Monitor
Waiting
tank is empty
Tank is empty
tank is being filled
Evaluatioin started
/entry OpaqueBehavior set evaluationStarted
AFTER(timeLimit) and tank is not full
Observing
Violated
/entry OpaqueBehavior set violated

Figure 4: Example of a vVDR requirement violation monitor

The tester will define verification scenarios for the verification of requirements. In the scenario depicted in Figure 5 the system will be filled, then drained, then the pump will be turned off, then the system will be filled again, etc.

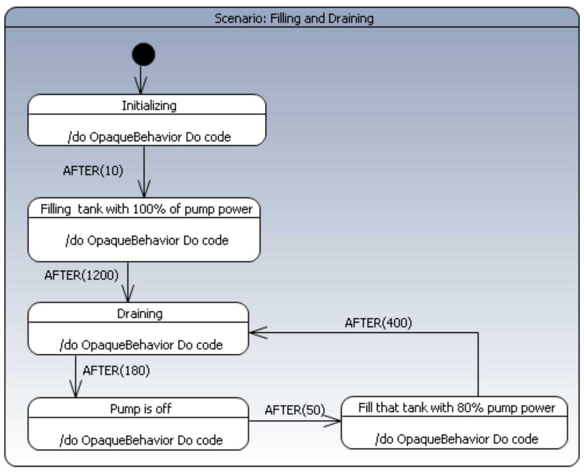

Scenario: Filling and Draining
Initializing
/do OpaqueBehavior Do code
AFTER(10)
Filling tank with 100% of pump power
/do OpaqueBehavior Do code
AFTER(1200)
Draining
/do OpaqueBehavior Do code
AFTER(400)
AFTER(180)
Pump is off
/do OpaqueBehavior Do code
AFTER(50)
Fill that tank with 80% pump power
/do OpaqueBehavior Do code

Figure 5: Example of a verification scenario

Each of the scenario steps contains action code to stimulate the system model. Figure 6 lists the properties of this verification scenario that are used to configure or stimulate the system model when this scenario should be used to verify it.

Usually, verification scenarios are created based on a given set of requirements with the intention to verify a design against these requirements. One scenario can be used to verify multiple requirements and one requirement is usually verified using multiple scenarios to increase the confidence in the verification results due to the independence of the scenarios.

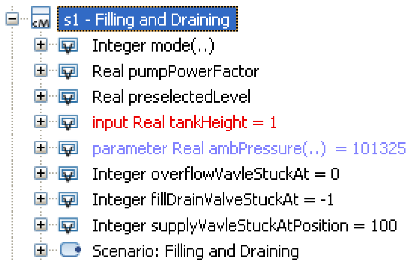

s1 - Filling and Draining
Integer mode(..)
Real pumpPowerFactor
Real preselectedLevel
input Real tankHeight = 1
parameter Real ambPressure(..) = 101325
Integer overflowVavleStuckAt = 0
Integer fillDrainValveStuckAt = -1
Integer supplyVavleStuckAtPosition = 100
Scenario: Filling and Draining

Figure 6: Example of scenario properties

The information about which scenarios are appropriate to verify which requirements is captured by setting dependencies between them as illustrated in Figure 7. Complementary to this information, when new requirements are linked to existing scenarios it is recommended to also capture why a specific scenario is not appropriate to verify a specific requirement (not illustrated here).

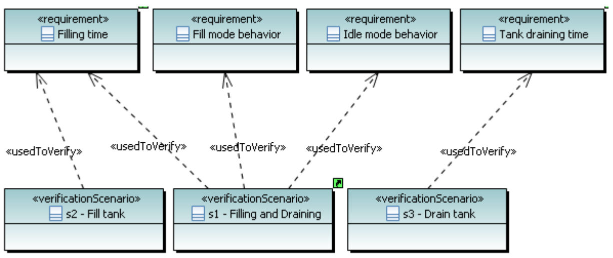

«requirement» Filling time
«requirement» Fill mode behavior
«requirement» Idle mode behavior
«requirement» Tank draining time
«usedToVerify»
«usedToVerify»
«usedToVerify»
«usedToVerify»
«usedToVerify»
«verificationScenario» s2 - Fill tank
«verificationScenario» s1 - Filling and Draining
«verificationScenario» s3 - Drain tank

Figure 7: Dependencies between verification scenarios and requirements

1.6. Paper Structure

In the remaining part of this paper we first introduce the Value Bindings Concepts, describe the enablers for the automated binding code derivation and verification models generation, and finally end with a conclusion.

2. VALUE BINDINGS

When models are not created collaboratively but shall be used together in different combinations the following additional questions arise:

- How to preserve the independence of models in order to facilitate their reuse and not affect them by enhancements that are necessary for a specific purpose (e.g. verification models generation)?
- How to ensure that enhancements made, for example defined bindings definitions, can still be reused when models evolve? That is, how to make sure that the defined binding definitions are not affected by the change of model element names or their structure?

The following sections propose answers to these questions. They introduce the concepts of clients, mediators and providers and show how binding code can be derived in order to enable automated model generation.

2.1. Basic Concepts

The central idea of the ModelicaML Value Bindings concept is that the data dependencies between models should be expressed in a way that does not affect, i.e., change, the models themselves. This was mainly done for two reasons: Firstly, it facilitates reuse and avoids cluttering up models with information that is only useful in certain usage situations. Secondly, it might not be possible to change the models, i.e., if they are from a third party or from a library. This is similar to the mediator pattern from the software development domain [5] which promotes the idea of a loose coupling of objects to enable communication without knowing each other explicitly.

If we translate this idea to the needs of the vVDR method then we can state that there are models that are created independently that will become dependent and need to be related to each other in some concrete usage.

For that, we introduce a notation of clients and providers that do not know each other explicitly. There may be multiple clients with the same need that can be satisfied by one provider. In turn, data from several providers may be needed by one client. This results in a many-to-many relation between clients and providers. The mediator concept is now introduced as a means to relate a number of clients to a number of providers as illustrated in Figure 8.

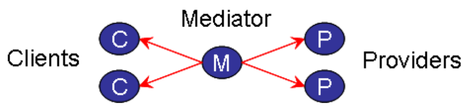


Figure 8: Basic concepts of client, mediator and provider

Note that the information about these relations is stored in the mediators so that the client or provider models are not modified.

The goal of this concept is to enable automated binding code generation for clients by finding the appropriate mediator. This mediator contains the information which providers need to be accessed and how the binding code can be derived.

The implementation of Value Bindings in ModelicaML is as follows:

- Each class property can be a client, a provider, or both.
- Mediators, clients and providers are created as properties of a class with dedicated UML stereotypes.
- UML dependencies are used for relating clients and providers. The dependencies point from the mediator to the clients and providers. This way the client and provider models are not modified. Mediators can be loaded (i.e., imported) when binding code needs to be generated.

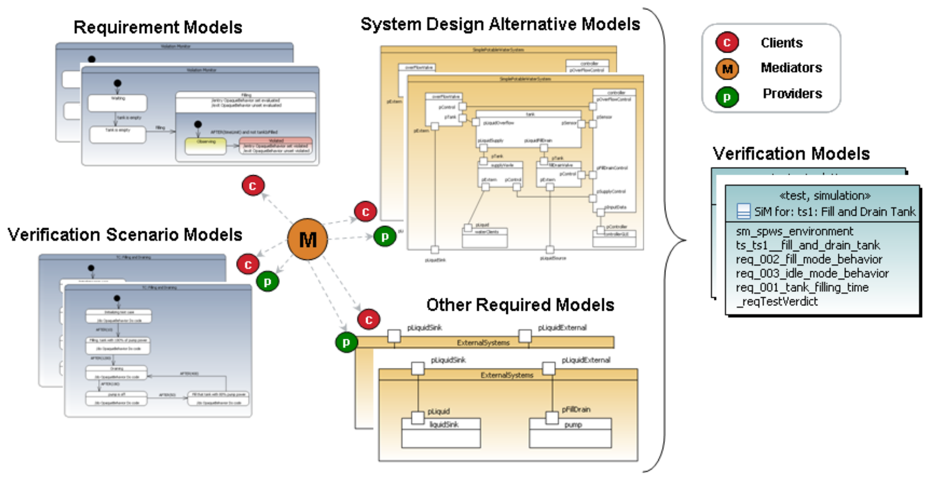
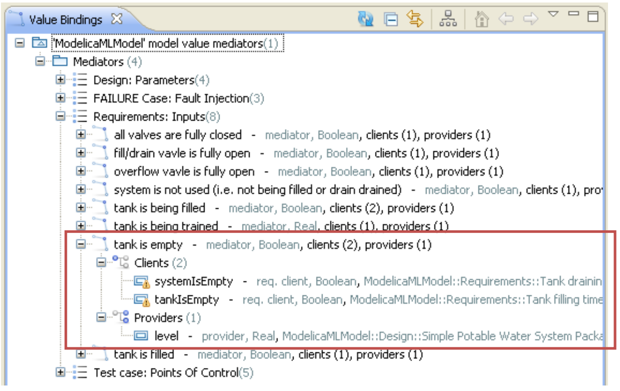


Figure 9: Clients, mediators and providers relations example

- UML dependencies are used to reference UML properties by using their unique identifiers, not their names. This way binding definitions will not be affected by changes of property names or owners, i.e., change in model structure.



```
Value Bindings
'ModelicaMLModel' model value mediators(1)
  Mediators (4)
    Design: Parameters(4)
    FAILURE Case: Fault Injection(3)
    Requirements: Inputs(8)
      all valves are fully closed  -  mediator, Boolean, clients (1), providers (1)
      fill/drain vavle is fully open  -  mediator, Boolean, clients (1), providers (1)
      overflow vavle is fully open  -  mediator, Boolean, clients (1), providers (1)
      system is not used (i.e. not being filled or drain drained)  -  mediator, Boolean, clients (1), pro
      tank is being filled  -  mediator, Boolean, clients (2), providers (1)
      tank is being trained  -  mediator, Real, clients (1), providers (1)
      tank is empty  -  mediator, Boolean, clients (2), providers (1)
        Clients (2)
          systemIsEmpty  -  req. client, Boolean, ModelicaMLModel::Requirements::Tank drainin
          tankIsEmpty  -  req. client, Boolean, ModelicaMLModel::Requirements::Tank filling time
        Providers (1)
          level  -  provider, Real, ModelicaMLModel::Design::Simple Potable Water System Packa
      tank is filled  -  mediator, Boolean, clients (1), providers (1)
    Test case: Points Of Control(5)
```

Figure 10: Example of Value Bindings

Figure 10 shows an example of ModelicaML value bindings. Consider the mediator `tank is empty` of type Boolean. There are two clients in two different requirement models that need this information. This mediator was created when the first client was identified.

The second client simply subscribed to the same mediator later on. At this point it was not necessarily clear how the design would look like and therefore how this information could be obtained from a design model. It was not until the creation of the design model, that the provider was defined.

In the example the component `level` of the class `Tank` in one specific design alternative model is the right place to obtain the information if the tank is empty. However, this provider is of type Real. Obviously, a conversion is needed as the clients expect a Boolean input. This can be achieved by means of binding operations as explained in chapter 3.

2.2. Enhancements of the vVDR method

Figure 9 illustrates the application of the basic concepts described in the previous section and implies the following enhancements of the vVDR method steps:

- Each model that requires data from other models should express this need by creating a new mediator or by subscribing to an already existing one.
- Each mediator must have defined[3] providers so that the correct binding code for its clients can be determined.

For example, in a collaborative work environment a requirements analyst, who formalizes requirements, will express the need that a requirement needs specific data in order to evaluate its violation by creating a new mediator or by subscribing to an existing one. The system designer, who creates the design model, will answer to this need by defining one or more providers for this mediator. In turn, a design model will require stimuli from the verification scenario, and the system designer will create new or reuse existing mediators to express this need. That will be answered by the tester, who will create the verification scenario and define the appropriate providers for the mediators.

If clients and providers are defined, then the binding code can be automatically derived for each client and an automated generation of verification models will be possible. The vVDR method steps (see section 1.1) are now enhanced as follows (changes in bold):

[3] Except if this mediator holds constant data. This is a special case for using mediators for example as means for defining different configurations. The configuration can then be determined based on which mediators are loaded in the model on hand.

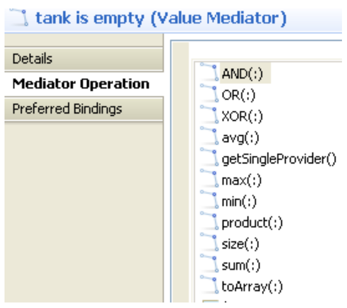
1. …
2. Formalize Requirements
   a. **Define clients**
3. Select or Create Design Model to be Verified against Requirements
   a. **Define clients and providers**
4. Select or Create Verification Scenarios
   a. **Define clients and providers**
5. **Generate** Verification Models
6. …

3. BINDING OPERATIONS

The relation between clients and providers can convey further definitions called binding operations. The action language that is used for the definition of such an operation is an enhanced version of the Modelica expression syntax [1]. The following sections explain the purpose of binding operations and provide examples of their usage.

3.1. Mediator Binding Operation

The mediator binding operation is mainly used as an array reduction function to reduce a selection to one item, for example, by taking the minimum or the maximum value, sum values, build a product, or to use the logic operators such as AND, OR, or XOR. Figure 11 lists the predefined binding functions that can be used in expressions. These functions are only used if multiple providers are anticipated for this mediator. For example, assume that the mediator is `system weight` of type Real. Now each of the sub-systems may reference providers to contribute its own weight. All of them will be summed up by the mediator by using the function `sum(:)` so that the mediator effectively conveys the system total weight.



```
tank is empty (Value Mediator)

Details                AND(:)
Mediator Operation     OR(:)
Preferred Bindings     XOR(:)
                       avg(:)
                       getSingleProvider()
                       max(:)
                       min(:)
                       product(:)
                       size(:)
                       sum(:)
                       toArray(:)
```

Figure 11: Predefined Mediator Binding Operation functions

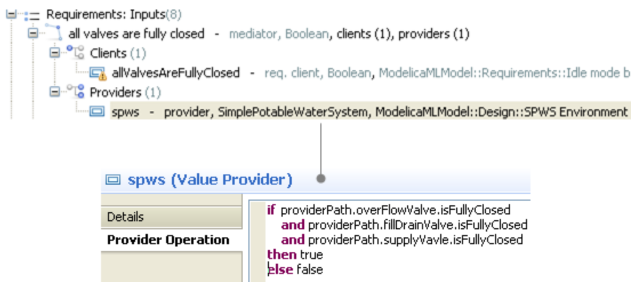
In contrast, the function `getSingleProvider()` is used to ensure that there must be only one provider for this mediator. If multiple providers can be found, then the user will need to check if it is an incorrect definition or if a choice is needed.

3.2. Provider Binding Operation

The provider binding operation is typically used when the actual provider is a sub-component. For example, imagine a mediator `all valves are fully closed` for the requirement R002. Which valves exist in the system and how to determine if all of them are closed can be expressed by referencing the next upper component that contains all valves (e.g. the overall system `spws`) and by defining the provider operation as illustrated in Figure 12:

Note, `providerPath` is a reserved key word that is replaced by the actual instance name when the binding code is derived.

Another example for using provider binding operations is type conversion of values. Consider the example from section 2.1 where a binding operation `providerPath < 0.001` would convert a Real type value to a Boolean.



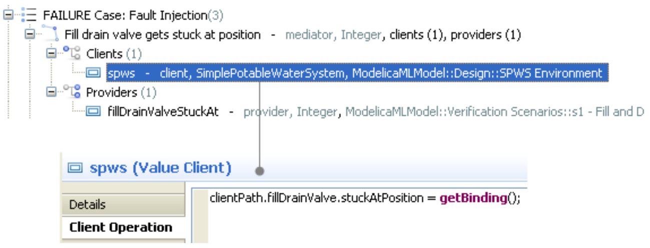
```
Requirements: Inputs(8)
  all valves are fully closed - mediator, Boolean, clients (1), providers (1)
    Clients (1)
      allValvesAreFullyClosed - req. client, Boolean, ModelicaMLModel::Requirements::Idle mode b
    Providers (1)
      spws - provider, SimplePotableWaterSystem, ModelicaMLModel::Design::SPWS Environment

spws (Value Provider)

Details
Provider Operation

if providerPath.overFlowValve.isFullyClosed
  and providerPath.fillDrainValve.isFullyClosed
  and providerPath.supplyVavle.isFullyClosed
then true
else false
```

Figure 12: Example of a provider binding operation referencing sub-components

3.3. Client Binding Operation

Similarly to the provider binding operation, client binding operations are typically used when a single instance must be referenced explicitly, i.e., when the actual client is a sub-component. Consider the example depicted in Figure 13. The client that is referenced in the binding definition is again the overall system `spws` that is instantiated in the class `SPWS Environment`. However, the actual binding is for a sub-component `fillDrainValve.stuckAtPosition`. It would be possible to reference the class attribute `stuckAtPosition` directly if the class `Valve` was instantiated exactly one time in the design. This is not the case in this design model because all valves are instances of the same type. Therefore, an explicit instance, the `fillDrainValve`, has to be referenced. Another example: assume the system contains two tanks of the same type `Tank` and there is a requirement that is imposed on the second tank, the instance `tank2`, explicitly. Then the actual client will again be the overall system component `spws` and the binding code could look like: `clientPath.tank2.level = getBinding();`[4].

[4] There may be multiple sub-components that are referenced this way. Individual entries are separated by ";".



```
FAILURE Case: Fault Injection(3)
  Fill drain valve gets stuck at position  -  mediator, Integer, clients (1), providers (1)
    Clients (1)
      spws  -  client, SimplePotableWaterSystem, ModelicaMLModel::Design::SPWS Environment
    Providers (1)
      fillDrainValveStuckAt  -  provider, Integer, ModelicaMLModel::Verification Scenarios::s1 - Fill and D
```

spws (Value Client)

Details | Client Operation

```
clientPath.fillDrainValve.stuckAtPosition = getBinding();
```

Figure 13: Example of a client binding operation referencing sub-components

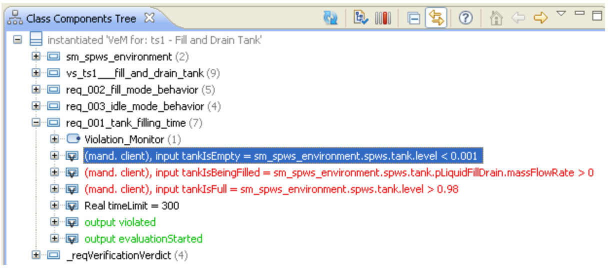
Another usage example for client binding operations is when the client is subscribed to an existing mediator that does not provide the same type (e.g. Boolean). Then the binding operation can be used for value conversions in the same way as the provider binding operation (see section 3.2).

Yet another example is the overwriting of binding definitions. Any upper binding definition in the components tree overwrites lower level bindings.

Like `providerPath`, `clientPath` is a reserved keyword that is replaced with the instance's dot-notation name when the binding code is derived.

4. BINDING CODE DERIVATION

Based on the model references and the binding operation for clients, mediators and providers, the binding code can be derived automatically when the clients and the providers are in the same components tree. Figure 14 shows an example of the deduced components tree of a class that contains a system design alternative, a verification scenario and a set of requirements. Based on this information and the information contained in the currently loaded mediators, it is now possible to automatically derive the binding code for each client. For example, the binding code for the client `tankIsEmpty` of the requirement `R001` instance is `tankIsEmpty = sm_spws_environment.spws.tank.level < 0.001`.



Class Components Tree

```
instantiated 'VeM for: ts1 - Fill and Drain Tank'
  sm_spws_environment (2)
  vs_ts1___fill_and_drain_tank (9)
  req_002_fill_mode_behavior (5)
  req_003_idle_mode_behavior (4)
  req_001_tank_filling_time (7)
    Violation_Monitor (1)
    (mand. client), input tankIsEmpty = sm_spws_environment.spws.tank.level < 0.001
    (mand. client), input tankIsBeingFilled = sm_spws_environment.spws.tank.pLiquidFillDrain.massFlowRate > 0
    (mand. client), input tankIsFull = sm_spws_environment.spws.tank.level > 0.98
    Real timeLimit = 300
    output violated
    output evaluationStarted
  _reqVerificationVerdict (4)
```

Figure 14: ModelicaML components tree

In case there is more than one mediator for a client for which the binding code should be derived, or the selection of providers that are available in the components tree cannot be done unambiguously, the tool will ask the user to make a decision. For selected providers, such decisions can be stored as preferred providers and reused for future binding code derivations.

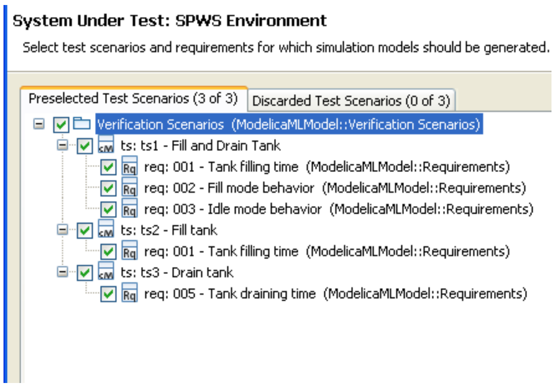
5. VERIFICATION MODELS GENERATION

Along with the automated binding of components, the binding code derivation can answer two questions for a given set of one design model, one scenario and a set of requirements.

- Is this requirement implemented in the design model? This can be answered by determining if for all clients[5] contained in the requirement the binding code to the design model can be derived.
- Does the test scenario provide stimuli for the system model? Again, this can be answered by determining if for all system model clients the binding code can be derived.

Furthermore, following the relations between the verification scenario and the requirements enables the identification of all requirements that can be verified using a given scenario.

All this information can now be used to determine valid combinations of scenarios and requirements for a selected system design alternative model as illustrated in Figure 15. Now, verification models can be generated automatically comprising components that are bound correctly (see and example in Figure 14).



System Under Test: SPWS Environment

Select test scenarios and requirements for which simulation models should be generated.

Preselected Test Scenarios (3 of 3) | Discarded Test Scenarios (0 of 3)

```
[✓] Verification Scenarios  (ModelicaMLModel::Verification Scenarios)
  [✓] ts: ts1 - Fill and Drain Tank
    [✓] req: 001 - Tank filling time  (ModelicaMLModel::Requirements)
    [✓] req: 002 - Fill mode behavior  (ModelicaMLModel::Requirements)
    [✓] req: 003 - Idle mode behavior  (ModelicaMLModel::Requirements)
  [✓] ts: ts2 - Fill tank
    [✓] req: 001 - Tank filling time  (ModelicaMLModel::Requirements)
  [✓] ts: ts3 - Drain tank
    [✓] req: 005 - Tank draining time  (ModelicaMLModel::Requirements)
```

Figure 15: Dialog for verification models generation

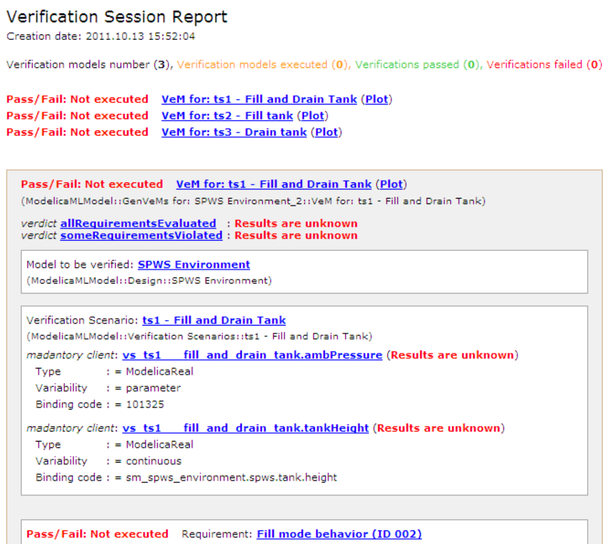
The generated verification models can be executed automatically and a report can be generated that shows a summary of the verification session. This summary includes the simulation configuration (i.e., what was the design

[5] In ModelicaML Value Bindings all clients are optional (provided they have default values) if they are not marked as mandatory. I.e., here the question is rather: can the binding code be derived for all mandatory clients?

model, which scenario was used, against which requirements was the system model verified), and the results of requirement violation observations as illustrated in Figure 16.

6. CONCLUSIONS

The approach presented in this paper enables the definition and reuse of bindings for the on-demand binding code generation and an automated model composition. ModelicaML bindings are defined at class attribute level and use the ModelicaML unique identifiers for storing the relation between clients, mediators and providers. Binding definitions do not modify client or provider models. Moreover, the binding definitions are not affected by the change of client and provider names or model structure except when binding operations are used that include references to subcomponents with concrete names. The binding code can be generated on-demand when instantiated models need to be bound, for example in a verification model.



```
Verification Session Report
Creation date: 2011.10.13 15:52:04

Verification models number (3), Verification models executed (0), Verifications passed (0), Verifications failed (0)

Pass/Fail: Not executed   VeM for: ts1 - Fill and Drain Tank (Plot)
Pass/Fail: Not executed   VeM for: ts2 - Fill tank (Plot)
Pass/Fail: Not executed   VeM for: ts3 - Drain tank (Plot)

Pass/Fail: Not executed   VeM for: ts1 - Fill and Drain Tank (Plot)
(ModelicaMLModel::GenVeMs for: SPWS Environment_2::VeM for: ts1 - Fill and Drain Tank)

verdict allRequirementsEvaluated  : Results are unknown
verdict someRequirementsViolated : Results are unknown

Model to be verified: SPWS Environment
(ModelicaMLModel::Design::SPWS Environment)

Verification Scenario: ts1 - Fill and Drain Tank
(ModelicaMLModel::Verification Scenarios::ts1 - Fill and Drain Tank)
madantory client: vs_ts1___fill_and_drain_tank.ambPressure (Results are unknown)
  Type         : = ModelicaReal
  Variability  : = parameter
  Binding code : = 101325

madantory client: vs_ts1___fill_and_drain_tank.tankHeight (Results are unknown)
  Type         : = ModelicaReal
  Variability  : = continuous
  Binding code : = sm_spws_environment.spws.tank.height

Pass/Fail: Not executed   Requirement: Fill mode behavior (ID 002)
```

Figure 16: Example of a verification session report (this figure will be updated for the final paper version)

However, when generated, the binding code contains explicit instance names that were derived from the components tree. When binding operations are updated or when they include names of subcomponents that have changed, the binding code is not valid anymore and the generated bindings must be updated. In general, in order to make sure that the binding code is correct it is recommended to regenerate the verification models for each verification session. The generated verification models thereby become artifacts that are created on-demand and are not needed anymore after the verification is finished. This way, each of the involved roles (e.g. requirement analyst, system designer or tester) can concentrate on the main artifacts and does not need to maintain the verification models[6]. This paradigm is in line with the overall ModelicaML approach in which Modelica code is generated from the ModelicaML model only when it is needed for simulations. A ModelicaML model contains all the information that is required to re-generate code at any time. This is advantageous because it reduces the number of models that need to be maintained. However, it might become a disadvantage for a verification session when the generation of a large number of models would take a long time and therefore block the session to start. This possible issue is subject to empirical observations that will be addressed in our future work when large models will be used for design verification following this approach.

References

[1] Modelica Association. Modelica: A Unified Object-Oriented Language for Physical Systems Modeling: Language Specification Version 3.2, March 24, 2010. http://www.modelica.org.

[2] UML. OMG Unified Modeling Language ™ (OMG UML). http://www.uml.org/

[3] Wladimir Schamai, Philipp Helle, Peter Fritzson, Christian Paredis. "Virtual Verification of System Designs against System Requirements", In: *Proc. of 3rd International Workshop on Model Based Architecting and Construction of Embedded Systems* (ACES 2010), 53-66.

[4] ModelicaML – A UML Profile for Modelica. www.openmodelica.org/modelicaml

[5] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994

[6] Paredis, C. et. al. "An Overview of the SysML-Modelica Transformation Specification". In: *Proceedings of the 2010 INCOSE International Symposium*, Chicago, IL, 2010.

[7] Aleksandr A. Kerzhner, Christiaan J.J. Paredis. "Model-Based System Verification: A Formal Framework for Relating Analyses, Requirements, and Tests". In: *Proceedings of the 4th International Workshop on Multi-Paradigm Modeling (MPM 2010)*.

[8] OMG SE DSIG SysML-Modelica Working Group. *SysML-Modelica transformation specification*. http://www.omgwiki.org/OMGSysML/doku.php?id=sysml-modelica:sysml_and_modelica_integration.

[6] Except if there is a need to enable the reproduction of simulation results. In this case, it will be necessary to keep the verification models and the exact binding code that was used.