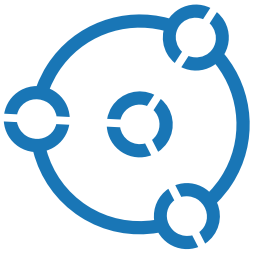
D4.3	Requirement tracing support in OpenModelica
Access[1]:	**PU**
Type[2]:	**Prototype**
Version:	**1.2**
Due Dates[3]:	**M24, M36**



openCPS

Open Cyber-Physical System Model-Driven Certified Development

Executive summary[4]:

This deliverable is concerned with support of requirement traceability in OpenModelica and graphical support of requirement modeling and verification in OMEdit

[1] Access classification as per definitions in PCA; PU = Public, CO = Confidential. Access classification per deliverable stated in FPP.
[2] Deliverable type according to FPP, note that all non-report deliverables must be accompanied by a deliverable report.
[3] Due month(s) according to FPP.
[4] It is mandatory to provide an executive summary for each deliverable.

Deliverable Contributors:

	Name	Organisation	Primary role in project	Main Author(s)[5]
Deliverable Leader[6]	Lena Buffoni	LiU	T4.3 leader	X
Contributing Author(s)[7]	Adrian Pop	SICSEast	T4.3 member	
Internal Reviewer(s)[8]	Martin Sjölund	LiU	WP4 leader	

Document History:

Version	Date	Reason for change	Status[9]
1.0	10/11/2017	First Issue	In Review
1.1	14/11/2017	Minor corrections after review	Released
1.2	30/11/2018	Update of progress	Released

[5] Indicate Main Author(s) with an "X" in this column.
[6] Deliverable leader according to FPP, role definition in PCA.
[7] Person(s) from contributing partners for the deliverable, expected contributing partners stated in FPP.
[8] Typically person(s) with appropriate expertise to assess deliverable structure and quality.
[9] Status = "Draft", "In Review", "Released".

Contents

Abbreviations

List of abbreviations/acronyms used in document:

Abbreviation	Definition
VVDR	Virtual Verification of Designs vs Requirements

1 Introduction

This is a deliverable report to accompany the prototype deliverable for requirement traceability and requirement modeling support in OpenModelica.

The work done in this deliverable is to automate the generation of verification models, proposing a fully Modelica based requirement verification support in OpenModelica and support for tracing requirement verification results.

2 Implementation

The basic concepts of the VVDR (Virtual Verification of Designs against Requirements) methodology used in this project are represented in the VVDR library[10].

VVDR library is a small Modelica library containing all the interfaces used by the OpenModelica algorithm to automatically generate the bindings. It is shipped by default with OpenModelica but can also be dowloaded separately.

It contains the following elements:

Requirement the requirement itself can be represented in different ways: by standard Modelica equations, state machines or dedicated libraries [OTB+15].

Scenario a scenario describes how the system is stimulated during the simulation

Design a design alternative is a possible implementation of the system. Several design alternatives can be included in the verification process (eg: different configurations of the system).

Verification Model this is the combination of a set of requirements, a design to be tested and a scenario. A verification model can be defined by the user or generated automatically.

Bindings package this contains the data structure for specifying the mediators used to generate the connections between the requirements, designs and scenarios.

The extensions in OpenModelica for requirement modeling are available in the latest version from openmodelica.org[11].

The example used in the paper and in the report can be found online[12].

In the current prototype support for batch simulation of the models and status report generation has been implemented.

3 Use Case

The first step is to load the VVDRlib, from File -> System libraries (Figure 1).

[10] https://github.com/lenaRB/VVDRlib
[11] https://openmodelica.org
[12] https://gitlab.ida.liu.se/olero90/RequirementsTutorial

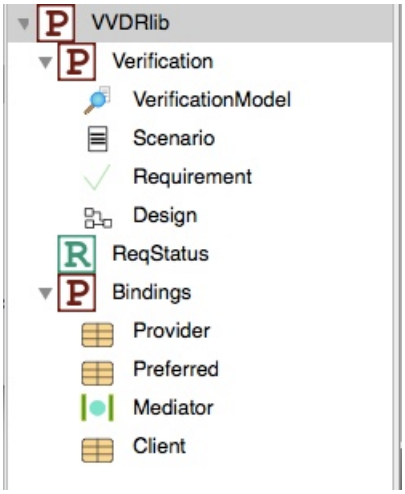


Figure 1: Structure of VVDR library

The next step is to load the example model, by loading the root package in TwoTanksExample, containing the two tank system model (in Figure 2).

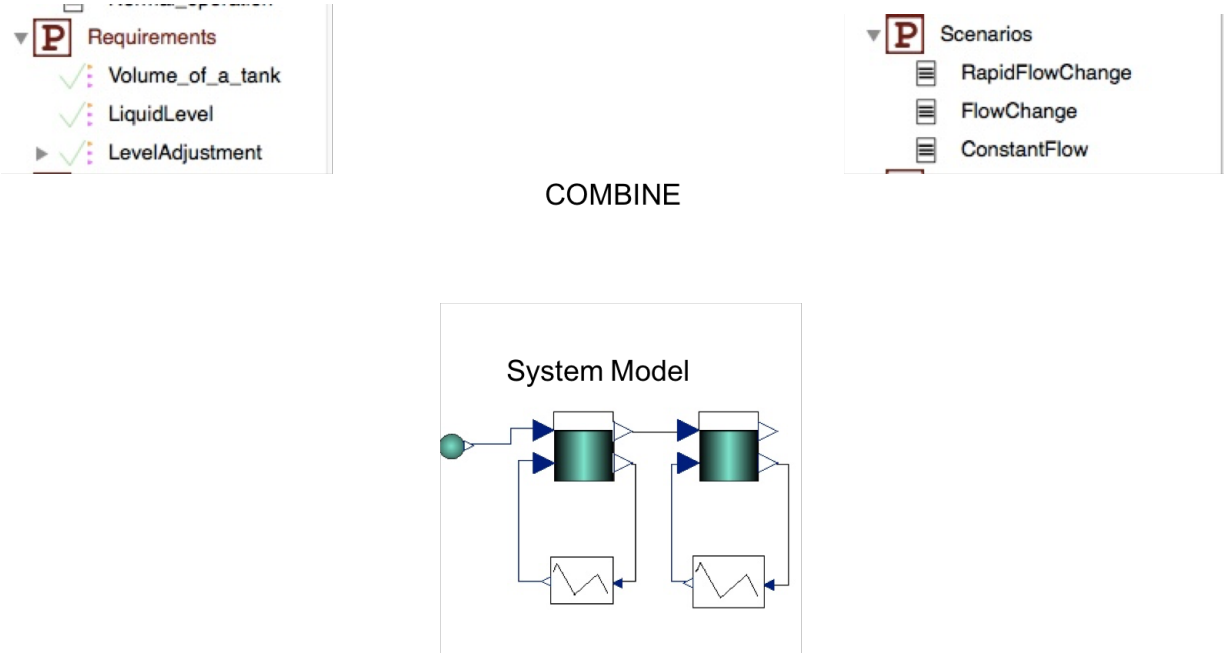


Figure 2: Scenarios, Requirements and Design Alternative

In the package, a set of requirements to verify the model against is also defined. It is implemented based on the following specification:

Req. 001: The volume of each tank shall be at least 2 m^3.

Req. 002: The level of liquid in a tank shall never exceed 80% of the tank height.

Req. 003: After each change of the tank input flow, the controller shall, within 20 seconds, ensure that the level of liquid in each tank is equal to the reference level with a tolerance of ± 1 0.05 m.

Requirements 1 and 2 are modelled as equations and requirement 3 is represented by a state machine.

A set of conditions under which the system will be tested can be found in the package `Scenarios`.

The VVDR methodology

A set of mediators to define how the requirements, scenarios and design alternatives map to each other [Sch13]. A binding is a causal relation which specifies that, at any simulated time, the value given to the referenced client instance shall be the same as the value computed by the right-hand expression. The connections between requirements and the system model are specified in mediators (Figure 3).

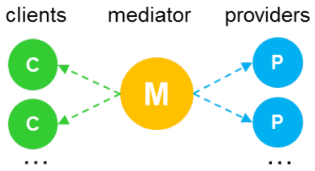


Figure 3: Mediators are used to connect together clients and providers

First the elements needed by the requirements from the system, or clients are defined by the requirement engineer:

```
record volumeLevel
  extends  Mediator(mType = "Real",
  clients = {Client(modelID = "TwoTanksExample.Requirements.Volume_of_a_tank",
    component = "tankVolume")});
end volumeLevel;
```

In a second step the system designer defines the way that information can be provided by the system, or providers:

```
record volumeLevel
  extends  Mediator(mType = "Real",
  clients = {Client(modelID = "TwoTanksExample.Requirements.Volume_of_a_tank",
    component = "tankVolume")},
  providers = {Provider(modelID = "TwoTanksExample.Design.Components.Tank",
    template = "%getPath.volume")});
end volumeLevel;
```

Based on these bindings, the possible combinations of requirements and scenarios together with the system model are generated and connected together in verification models. The result of scenario generation is a set of models that can be used to run tests.

Generating bindings for a single scenario

A user defined verification model can be created by dragging and dropping a set of requirements, a design alternative and a scenario:

```
model GraphicalModel
  extends VVDRlib.Verification.VerificationModel;
  Requirements.Volume_of_a_tank volume_of_a_tank1;
  Requirements.LiquidLevel liquidLevel1;
  Design.TwoTanksDesign;
end GraphicalModel;
```

In OMEdit the update bindings option can then be selected by right clicking the model and the correct number of requirements will be instantiated and connected with the model (Figure 4).

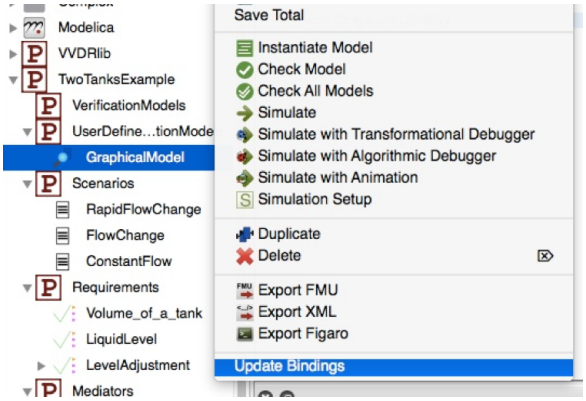


Figure 4: Binding generation in OMEdit

The resulting model contains the right number of requirement instances (in green), one requirement of each type for each tank and the correct connection code (in yellow).

```
model GraphicalModel
  extends VVDRlib.Verification.VerificationModel;
  TwoTanksExample.Requirements.Volume_of_a_tank volume_of_a_tank1_autogen_bind_0
      (tankVolume = twoTanksDesign1.tank1.volume);
  TwoTanksExample.Requirements.Volume_of_a_tank volume_of_a_tank1_autogen_bind_1
      (tankVolume = twoTanksDesign1.tank2.volume);
  TwoTanksExample.Requirements.LiquidLevel liquidLevel1_autogen_bind_0
      (waterLevel = twoTanksDesign1.tank1.levelOfLiquid);
  TwoTanksExample.Requirements.LiquidLevel liquidLevel1_autogen_bind_1
      (waterLevel = twoTanksDesign1.tank2.levelOfLiquid);
  TwoTanksExample.Design.TwoTanksDesign twoTanksDesign1;
end GraphicalModel;
```

This model can then be simulated and used to verify the requirements. Figure 5 shows the status for requirement one for tank one.

We can see that as the level of water goes above the 80% of the tank volume, the requirement status changes to violated (-1). Once the controller gets the water volume below that value, the requirement status switches back to not violated (-1).

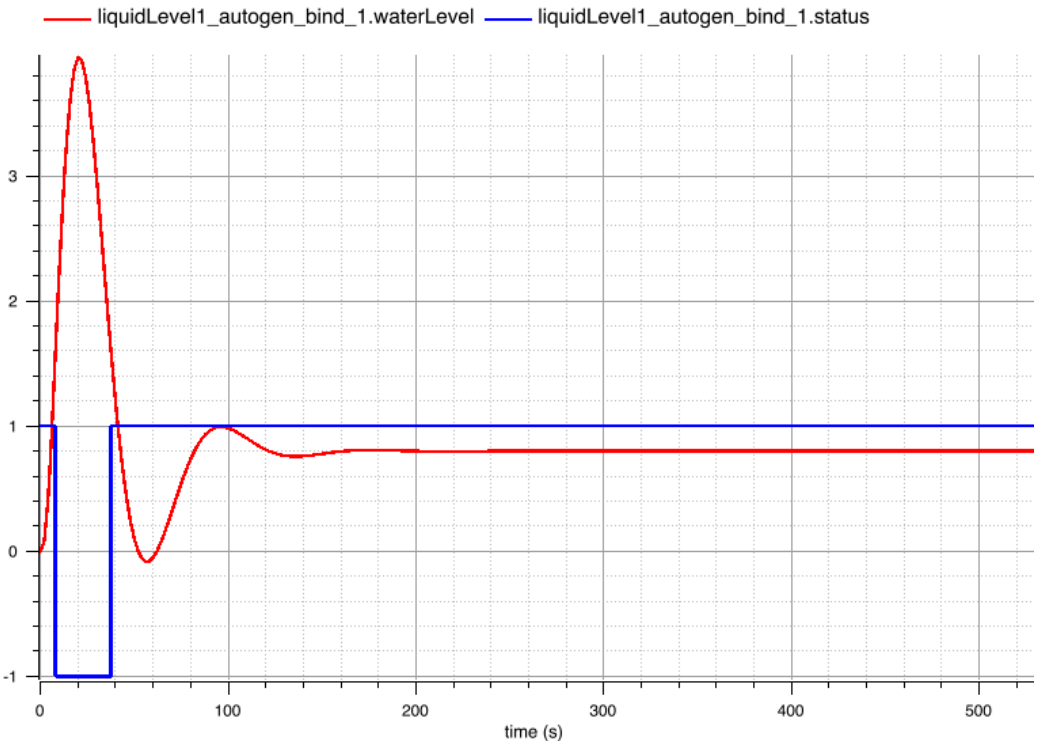


Figure 5: Requirement Req001 status for tank1

Automatic generation of scenarios

All the possible combinations of verification models can be generated by selecting an empty package, right clicking it and selecting the generate verification scenarios (Figure 6).

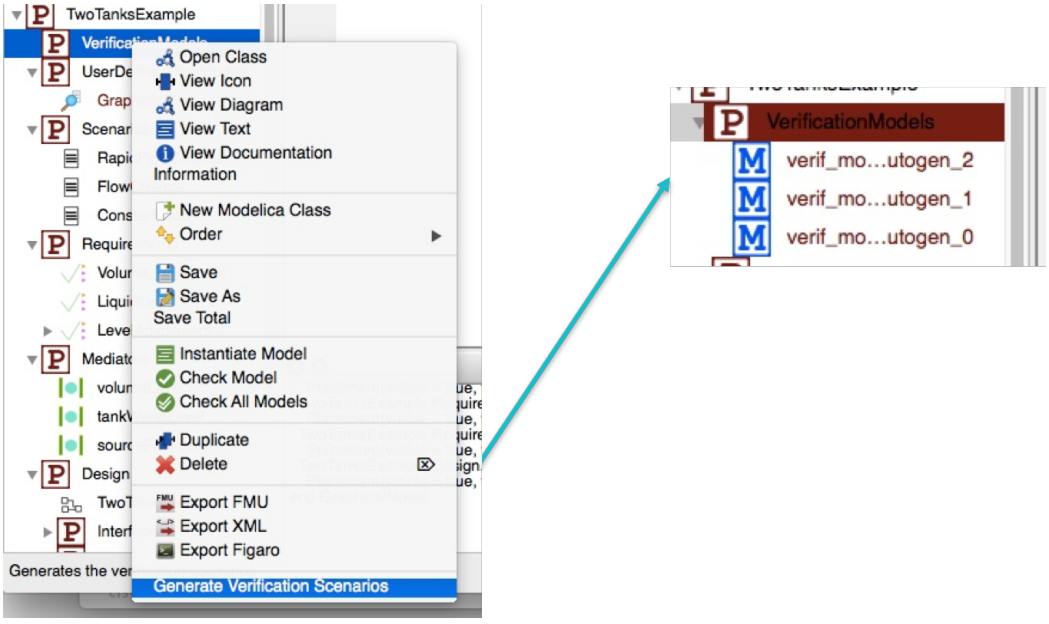


Figure 6: Generation of all verification model combinations

For the example in this tutorial a verification model is generated for each of the three scenarios, connecting all three requirements. An example of an automatically generated model:

```
within TwoTanksExample.VerificationModels;
model verif_model_autogen_2 "Autogenerated verification model"
  TwoTanksExample.Requirements.Volume_of_a_tank
      _agen_Volume_of_a_tank4_autogen_bind_0(tankVolume =
      _agen_TwoTanksDesign1.tank1.volume);
  TwoTanksExample.Requirements.Volume_of_a_tank
      _agen_Volume_of_a_tank4_autogen_bind_1(tankVolume =
      _agen_TwoTanksDesign1.tank2.volume);
  TwoTanksExample.Requirements.LiquidLevel _agen_LiquidLevel3_autogen_bind_0(
      waterLevel = _agen_TwoTanksDesign1.tank1.levelOfLiquid);
  TwoTanksExample.Requirements.LiquidLevel _agen_LiquidLevel3_autogen_bind_1(
      waterLevel = _agen_TwoTanksDesign1.tank2.levelOfLiquid);
  TwoTanksExample.Requirements.LevelAdjustment
      _agen_LevelAdjustment2_autogen_bind_0(inFlow =
      _agen_TwoTanksDesign1.tank1.levelOfLiquid);
  TwoTanksExample.Requirements.LevelAdjustment
      _agen_LevelAdjustment2_autogen_bind_1(inFlow =
      _agen_TwoTanksDesign1.tank2.levelOfLiquid);
  TwoTanksExample.Design.TwoTanksDesign _agen_TwoTanksDesign1(source.flowLevel =
      _agen_RapidFlowChange0.flowLevel);
  TwoTanksExample.Scenarios.RapidFlowChange _agen_RapidFlowChange0;
end verif_model_autogen_2;
```

Report Generation

Once a set of verification is generated, rather than to simulate each model separately and verify the status for each requirement, an extension has been implemented to simulate all of the verification models and to check which requirements have been violated in which models. The results are then summarized into a json file, with a list of models for which requirements have been violated and these can then be investigated further manually.

This file can also be compared to its previous version when something is changed in the model to make tracking the impact of changes on the model easier.

The report is generated via the call to `generateVerificationResults` of the OpenModelica API which takes the repository where the verification models have been automatically generated as a parameter.

```
v{
        "validation_results": {
                "package": "TwoTanksExample.VerificationModels",
                "verification_models_tested": "3",
                "requirements_found": "3",
                "requirements_tested": {
                        "number": "3",
                        "names": ["TwoTanksExample.Requirements.Volume_of_a_tank",
                             "TwoTanksExample.Requirements.LiquidLevel", "
                            Twotanksexample.Requirements.LevelAdjustment"]
                },
                "requirements_violated": [{
                        "name": "TwoTanksExample.Requirements.LiquidLevel",
                        "in_models": ["_agen_Volume_of_a_tank4_autogen_bind_0", "
                            _agen_Volume_of_a_tank4_autogen_bind_1"]
                }, {
                        "name": "TwoTanksExample.Requirements.LevelAdjustment",
                        "in_models": ["_agen_Volume_of_a_tank4_autogen_bind_0", "
                            _agen_Volume_of_a_tank4_autogen_bind_1"]
                }]
        }
}
```

References

[OTB+15] Martin Otter, Nguyen Thuy, Daniel Bouskela, Lena Buffoni, Hilding Elmqvist, Peter Fritzson, Alfredo Garro, Audrey Jardin, Hans Olsson, Maxime Payelleville, Wladimir Schamai, Eric Thomas, and Andrea Tundis. Formal requirements modeling for simulation-based verification, 2015. doi:10.3384/ecp15118625.

[Sch13] Wladimir Schamai. *Model-Based Verification of Dynamic System Behavior against Requirements*. 2013.

Appendix A

This paper explains the implementation of requirement traceability in OpenModelica.

The paper was presented at the EOOLT2017: 8th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools in Munich, Germany on December 1, 2017.

Traceability and impact analysis in requirement verification

[Work in Progress]

Lena Buffoni
Linköping University
Linköping, Sweden
lena.buffoni@liu.se

Adrian Pop
Linköping University
Linköping, Sweden
adrian.pop@liu.se

Alachew Mengist
Linköping University
Linköping, Sweden
alachew.mengist@liu.se

ABSTRACT

Seamless tracing of the requirements and associating them with the models and the simulation results is becoming increasingly important. This can be used to support several activities such as variant handling, impact analysis, component reuse, verification, and validation. This work in progress paper presents an approach for combining traceability with requirement verification in Modelica. Traceability is supported via the OSLC specification standard combined with Git version control system. All operations on artifacts of interest are traced. Currently, the traceability data is stored in a graph database which can be queried for generating various reports such as impact analysis, variant handling, etc.

CCS CONCEPTS

• **Computing methodologies** → **Modeling methodologies**; **Model verification and validation**;

KEYWORDS

Modelica, Verification, Traceability

ACM Reference Format:
Lena Buffoni, Adrian Pop, and Alachew Mengist. 2017. Traceability and impact analysis in requirement verification: [Work in Progress]. In *EOOLT'17: 8th International Workshop on Equation-Based Object-Oriented Languages and Tools, December 1, 2017, Wessling, Germany*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3158191.3158207

1 INTRODUCTION

In recent years the need for a more formal requirement verification process and for language and tool support has been increasingingly recognized by the cyber-physical modeling community. Several works on language and tool support have been proposed in this area [2, 4].

Having both the requirement and the model in the same language reduces the semantic gap in the terminology used between the requirement verification engineers and the system modelers, simplifies the modeling effort and allows for automated combination between the requirement models. However in industrial scale projects with complex systems, a large number of requirements, and several people or teams working on different parts of the system, it is still complicated to analyze the results of the requirement verification process just by looking at the simulation results.

This paper is based on work done in [5] to automate the generation of verification models and on work done in [1, 6], proposing a fully Modelica based requirement verification support in OpenModelica and combines this with the work on traceability [3] to propose tool support for tracing requirement verification results.

The paper is structured as follows Section 2 introduces the basic concepts of the requirement modeling methodology. Section 3 presents the use-case used to illustrate the approach and Section 4 describes the traceability process. Finally conclusions are drawn in Section 5.

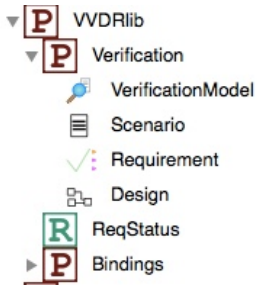


Figure 1: VVDRlib structure

2 BASIC CONCEPTS AND METHOD

The basic concepts of the VVDR (Virtual Verification of Designs against Requirements) methodology defined in [5] are represented in the VVDR library[1]. VVDR library is a small Modelica library containing all the interfaces used by the OpenModelica algorithm to automatically generate the bindings.

It has the following elements, represented in Figure 1:

Requirement the requirement itself can be represented in different ways: by standard Modelica equations, state machines or dedicated libraries [4].

Scenario a scenario describes how the system is stimulated during the simulation

Design a design alternative is a possible implementation of the system. Several design alternatives can be included in the verification process (eg: different configurations of the system).

[1]https://github.com/lenaRB/VVDRlib

EOOLT'17, December 2017, Munich, Germany

ACM ISBN 978-1-4503-6373-0/17/12…$15.00
https://doi.org/10.1145/3158191.3158207

Verification Model this is the combination of a set of requirements, a design to be tested and a scenario. A verification model can be defined by the user or generated automatically.

Bindings package this contains the data structure for specifying the mediators used to generate the connections between the requirements, designs and scenarios.

It is important to note that these interfaces are used by the tools to provide requirement modeling support only and do not restrict the way the elements are implemented themselves. The only restriction imposed by the library is a requirement interface that provides a status variable in each requirement which can take the following values, used to generate the verification reports:

Violated when the conditions of the requirement are not fulfilled by the design model, represented by -1 in the plots;

Not_violated when the conditions of the requirement are fulfilled by the design model, represented by 1 in the plots;

Not_applicable when the requirement does not apply, for instance a requirement that describes the behavior of a power system when it is switched on, cannot be verified when the system is off. This is important to identify requirements that were never tested during a simulation, represented by 0 in the plots.

For each requirement simulated, the following properties are recorded - wherther the requirement was verified, that is whether it was applicable at any point in time during the simulation, whether it was violated and if it was the time of the first violation.

For traceability, all modeling activities of interest and artifacts are recorded automatically and traced:

- Model creation, modification, deletion
- Simulation results
- Requirement verification reports

The work in [3] has been extended to provide classification of Modelica models into the VVDR classes: requirements, scenarios, designs and verification models.

The Open Services for Lifecycle Collaboration (OSLC) specification is used for integrating development lifecycle tools using Linked Data. For traceability purposes, in particular the OSLC Change Management specification is relevant. The following summarizes the main workflow that is used to create and record traceability information in OpenModelica during the development process:

(1) Commit artifact (model file, simulation result, report, etc) entity to Git repository and record the Git-hash
(2) Create unique URIs of the activity based on the Git-hash
(3) OSLC triples describing the activity are generated using the URIs
(4) OSLC triples are sent to the traceability daemon (and stored in a graph database)
(5) Retrieve the traceability information (traces to and traces from, used for impact analysis)

An example of a traceability graph is given in Figure 2. Entities (e.g. Modelica files, FMU, simulaiton results, reports) are shown in green, actions (e.g. model creation, model change, simulation result generation) are shown in yellow, agents (e.g. users with the names *"Alachew"*, *"Lena"*) are shown in blue, and their relationships "what come from what" and "what used what" (e.g. *"wasGeneratedBy"*, *"wasDerivedFrom"*, *"usedTool"*) are shown with red arrows.

3 USE CASE

The use case presented in this paper is a simple two tank model already described in detail in [5]. It consists of a single design alternative illustrated in Figure 3, three scenarios to illustrate different usage patterns and the four requirements specified as follows:

Req. 001 The volume of each tank shall be at least 2 m3.

Req. 002 The level of liquid in a tank shall never exceed 80% of the tank height.

Req. 003 With all tanks full, the maximum time to drain all tanks shall be 50 seconds.

Req. 004 After each change of the tank input flow, the controller shall, within 20 seconds, ensure that the level of liquid in each tank is equal to the reference level with a tolerance of ± 0.05 m.

Req. 005 In a pressurized tank the pressure will not vary by more than 5% from the reference level.

Figure 4 shows the overall structure of the package.

Once the different components of the system are defined, the binding algorithm implemented in OpenModelica can be used to generate all the possible verification models (Figure 5). This is done by collecting all the possible design alternatives, scenarios and requirements from the selected packages. For each design alternative and for each scenario a set of all the requirements that can be verified is collected and a verification model is generated. Since in this use case there is a single design alternative and 3 possible scenarios, this results in three possible combinations, one for each scenario. Every verification model includes the set of requirements for which bindings could be computed.

Once one of these models is simulated, requirement status values can be plotted for each requirement to verify whether they were violated during the simulation. Figure 6 shows that Req 002 is violated twice for tank 1 during the simulation with the "overflow" scenario, is first set to 0.06 and increased to 0.25 after 150 seconds. Each time as the water flow increases, we can see that it reaches over 80% of the tank volume before it is adjusted by the controller.

However verifying requirements this way, one by one, is cumbersome for a larger system, moreover when re-simulating a modified system we want to be able to detect quickly if the changes to the design or the requirement model have affected the systems ability to fulfill the requirements. To this end we introduce the first prototype for traceability support in the next section.

4 TRACING REQUIREMENTS

4.1 Tracing requirement violations

The goal of the verification status report is to simulate all the models in the selected package that inherits from `VerificationModel` and to produce a simulation report which will quickly allow to identify the areas of interest.

The algorithm for generation of verification models builds a graph of the relationships between the scenarios, requirements and design alternatives which is then passed to the function generating the report. The report is generated by collecting all the instances

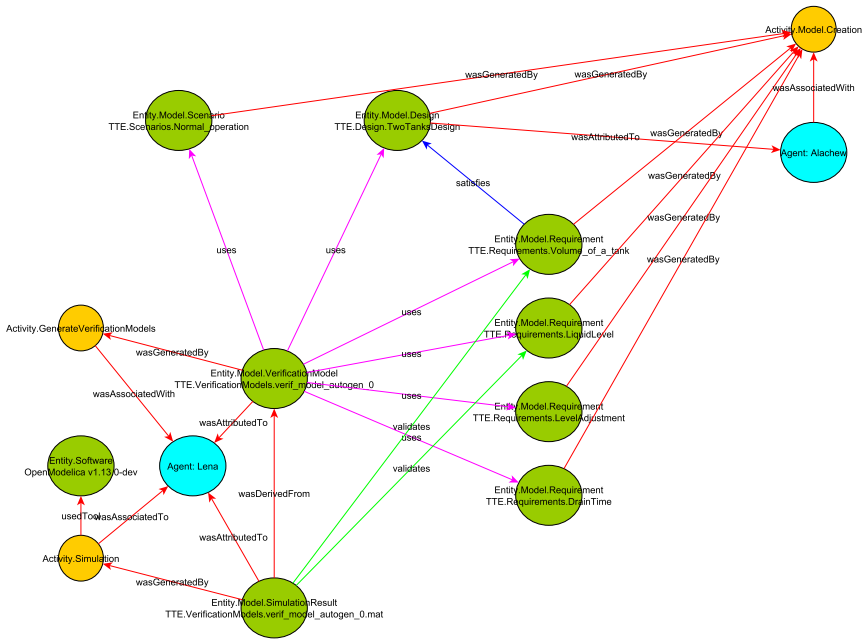


Figure 2: Traceability graph

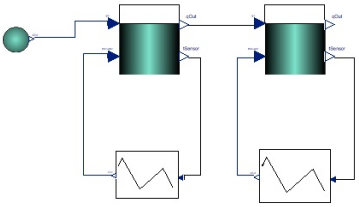


Figure 3: Tank design alternative

of requirements for each verification model and by checking the simulation results for each instance.

For instance, the json structure below is part of a generated data-structure for the use-case presented in this paper.

```
{
  "validation_results":{
      "package":"TwoTanksExample.VerificationModels",
      "verification_models_tested":"3",
      "design_alternatives_found":"1",
      "design_alternatives_bound":"1",
      "requirements_found":"5",
      "requirements_bound":"4",
      "requirements_tested":"3",
      ...
  }}
```

We can use this report to detect several types of problems:

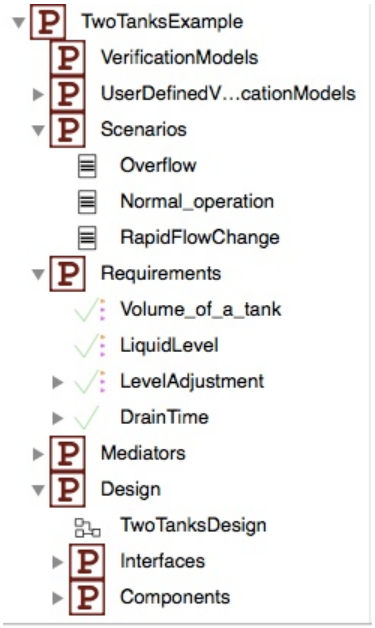


Figure 4: The contents of the TankExample package

Not all the requirements are bound. This means that one of the requirements is not applicable to the chosen design alternative. For example a requirement is written for a pressurized tank, but the design alternative verified has no pressure sensor to connect to the requirement. This is to be expected, because the requirements

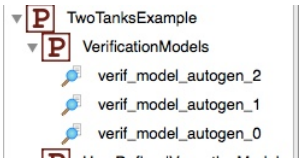


Figure 5: Verification models generated automatically

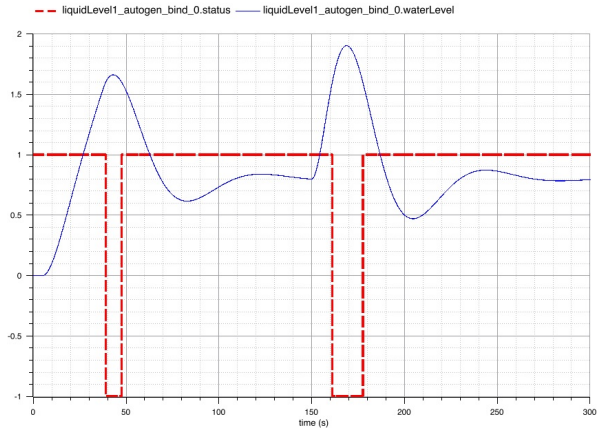


Figure 6: Requirement Req 002 status for tank 1 in the overflow scenario

are written from a specification rather than for a particular system implementation choice, but can be a sign of something missing in the design.

Not all the bound requirements are tested. If a requirement is bound but never tested it may be a sign that none of the scenarios cover the test conditions necessary for this requirement. This is the case for `Req 004`, because in none of the scenarios the tank is ever emptied. This can be a sign that parts of the behavior have not been correctly or fully tested.

A number of requirements have been violated. The report provides a list of requirement violations across scenarios. Looking into the violated requirement will bring up more details.This will provide a list of scenarios in which the requirement was violated and the time of the first violation of the requirement.


```
{
  "validation_results":{
     ...
    "validated_models":[ {
        "name":"verif_model_autogen_0",
        "requirements":[ {
            "name":"_agen_Volume_of_a_tank5_autogen_bind_0",
            "type":"TwoTanksExample.Requirements.Volume_of_a_tank",
            "wasVerified":"true",
            "wasViolated":"false"
          },{
            "name":"_agen_LiquidLevel4_autogen_bind_0",
            "type":" TwoTanksExample.Requirements.LiquidLevel",
            "wasVerified":"true",
            "wasViolated":"true",
            "firstViolationTime":"0"
    }]},{
        "name":"verif_model_autogen_1",
        "requirements":[ {
            "name":"_agen_LevelAdjustment3_autogen_bind_1",
            "type":"TwoTanksExample.Requirements.LevelAdjustment",
            "wasVerified":"true",
            "wasViolated":"false"
          },{
            "name":"_agen_LiquidLevel4_autogen_bind_0",
            "type":" TwoTanksExample.Requirements.LiquidLevel",
            "wasVerified":"true",
            "wasViolated":"false"
}]}]}
```


4.2 Traceability and impact analysis

The previous section illustrates how we can trace problems in a particular system design using requirements. However if the modeler makes a change in the design and goes through verification process again, some requirements might be affected (eg: a requirement that was satisfied in all scenarios before might be violated in certain scenarios). In order to support the verification process throughout the development life-cycle we can use the variant traceability in OpenModelica coupled with the requirement violations to track the places which might have caused the system to fail requirements satisfied previously.

Because all artifacts and actions are traced one can use the traceability information and the Git repository to provide impact analysis from different perspectives, i.e. what is affected (with regards to verification) by a change in: a requirement, a scenario or a design. One could even start from two different verification reports and highlight their differences with respect to all involved artifacts.

5 CONCLUSION AND FUTURE WORK

In this paper we have presented ongoing work on providing improved support for requirement verification throughout the development process by leveraging the work done on traceability in OpenModelica and integrating it with the automatic model composition methodology and by extending it with batch simulation and agglomerating the information in an easy to interpret report.

In this first prototype we simply display a list of requirement violations and corresponding times. In the future it would be useful to define tolerance levels to requirement violations. For example a single spike over a value limit might mean a requirement violation in one case but be tolerated by the system in another.

As this is work in progress we have not yet finished integrating all the pieces together. We currently have Git versioning support, the traceability database with all the artifacts linked together and means to run verification models automatically to generate requirement satisfaction and coverage reports. For impact analysis we need to develop queries over the traceability data starting from requirement verification reports. Integration of this framework in the OpenModelica connection editor is under way.

ACKNOWLEDGMENTS

This work has been supported by Vinnova in the ITEA3 OPENCPS project and in the Vinnova RTISIM project. Support from the Swedish Government has been received from the ELLIIT project, as well as from the European Union in the H2020 INTO-CPS project. The OpenModelica development is supported by the Open Source Modelica Consortium.

REFERENCES

[1] Lena Buffoni and Peter Fritzson. 2014. Expressing Requirements in Modelica. (2014).

[2] Alfredo Garro, Andrea Tundis, Daniel Bouskela, Audrey Jardin, Nguyen Thuy, Martin Otter, Lena Buffoni, Peter Fritzson, Martin SjÃűlund, Wladimir Schamai, and Hans Olsson. 2016. On formal cyber physical system properties modeling: A new temporal logic language and a Modelica-based solution. In *2016 IEEE International Symposium on Systems Engineering (ISSE)*. 1–8. https://doi.org/10.1109/SysEng.2016.7753137

[3] Alachew Mengist, Adrian Pop, Adeel Asghar, and Peter Fritzson. 2017. Traceability Support in OpenModelica Using Open Services for Lifecycle Collaboration (OSLC). In *Proceedings of the 12th International Modelica Conference*. Modelica Association and Linköping University Electronic Press.

[4] Martin Otter, Nguyen Thuy, Daniel Bouskela, Lena Buffoni, Hilding Elmqvist, Peter Fritzson, Alfredo Garro, Audrey Jardin, Hans Olsson, Maxime Payelleville, Wladimir Schamai, Eric Thomas, and Andrea Tundis. 2015. Formal Requirements Modeling for Simulation-Based Verification. (2015). https://doi.org/10.3384/ecp15118625

[5] Wladimir Schamai. 2013. *Model-Based Verification of Dynamic System Behavior against Requirements*.

[6] Wladimir Schamai, Lena Buffoni, Nicolas Albarello, Pablo Fontes De Miranda, and Peter Fritzson. 2015. An Aeronautic Case Study for Requirement Formalization and Automated Model Composition in Modelica. (2015). https://doi.org/10.3384/ecp15118911