Verification and Design Exploration through Meta Tool Integration with OpenModelica

Zsolt Lattmann[2], Adrian Pop[1], Johan de Kleer[3], Peter Fritzson[1], Bill Janssen[3],
Sandeep Neema[2], Ted Bapty[2], Xenofon Koutsoukos[2],
Matthew Klenk[3], Daniel Bobrow[3], Bhaskar Saha[3], Tolga Kurtoglu[3]
[1]Department of Computer and Information Science
Linköping University, SE-581 83 Linköping, Sweden
[2]Vanderbilt University, Nashville, TN, USA
[3]Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304 USA

lattmann@isis.vanderbilt.edu, {adrian.pop, peter.fritzson}@liu.se, dekleer@parc.com

Abstract

Modelica models are typically used for simulation to investigate properties of a possible system designs. This is often done manually or combined with optimization to select the best design parameters.

It is desirable to have systematic and partly automated support for exploration of the design space of possible designs and verifying their properties vs. requirements. The META design tool chain is being developed to support this goal. It provides an integration framework for components, designs, design spaces, requirements, and test benches, as well as verification of requirements for the generated design models during design exploration

This paper gives an overview of the META tools and their integration with OpenModelica. The integrated environment currently has four main uses of OpenModelica: importing Modelica models into the META tool model structure, performing simulations within test benches, analyzing Modelica models and automatically adding fault modes, and extracting equations (DAEs) for formal verification tools, e.g. the QRM using qualitative reasoning.

A prototype of the integrated tool framework is in operation, being able to generate and simulate thousands of designs in an automated manner.

Keywords: Modelica, simulation, design exploration, verification, etc.

1 Introduction

A design tool chain (META tools, Figure 1) is being developed for exploring design alternatives under certain condition and to verify their properties versus formalized requirements.

A design is built from component model building blocks defining component dynamic behavior and is defined as a composition of component models. A design space can represent different component alternatives as well as different design architectures.

After a design or design space has been created, test cases can be defined against the given requirement set. The test cases, which are called *test benches*, are executable versions of the system requirements.

From the test bench models, the META tools can compose analysis packages over a design space for different domains such as simulation of DAEs (differential algebraic equations), formal verification, static analysis, and structural analysis.

The integrated environment currently has four main uses of OpenModelica: importing Modelica models into the META tool model structure, performing simulations within test benches, analyzing Modelica models and automatically adding fault modes, and extracting equations (DAEs) needed for formal verification tools.

2 The OpenMETA Tool Chain

The OpenMETA[1] tool chain is being developed under DARPA's Adaptive Vehicle Make (AVM) [5] program that contains a set of projects one of them is the META project. The AVM program aims to reduce vehicle design and manufacturing time using the framework and toolset provided by the META program.

[1] Provided under MIT license

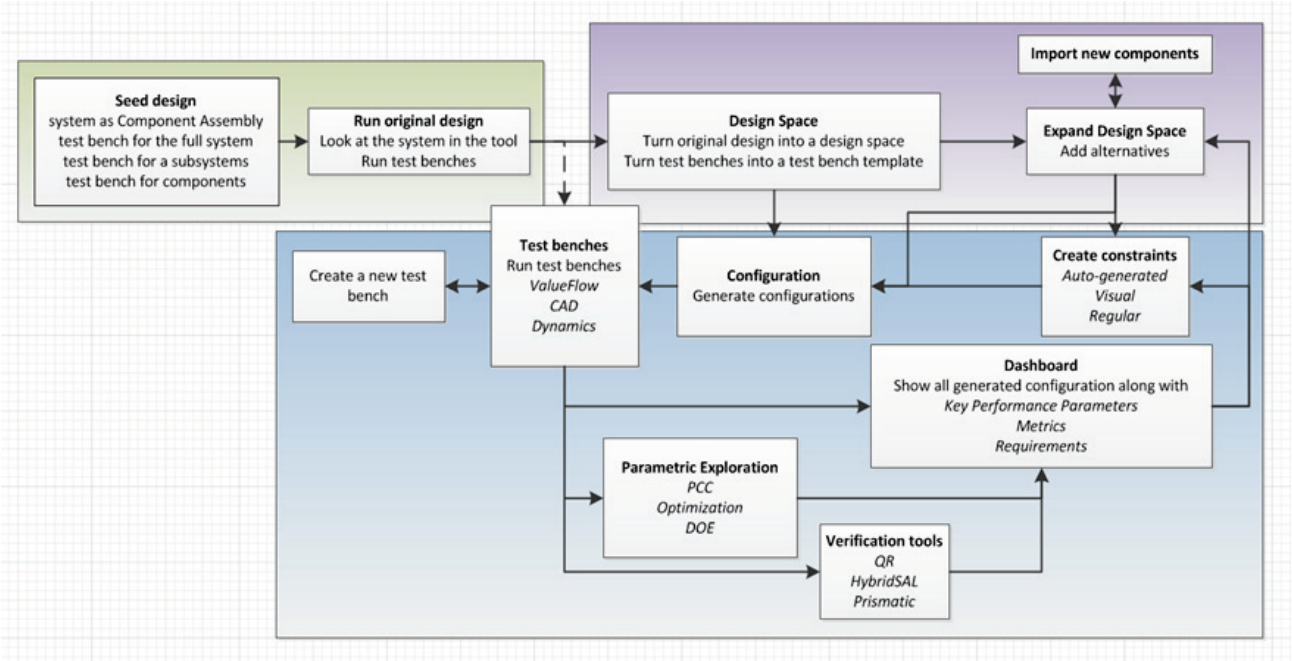

Figure 1. Design flow in the OpenMETA Tool Chain.

The tool chain consists of a language/meta-model called Cyber Physical Modeling Language (*CyPhyML*), a set of *model transformation software* components that translate from CyPhyML models to various domain tools, an analysis package executor (referred to as *Job Manager*), and a visualizer (referred to as *Project Analyzer*) for inspecting and understanding the results of analysis packages.

We present (a) the concepts defined in CyPhyML in Section 2.1, (b) the integration points with and utilization of OpenModelica in Section 2.2 and Section 2.3, (c) collected analysis results in Section 2.4 and Section 2.5, and (d) the usage of formal verification methods in Section 2.6.

2.1 Concepts

CyPhyML is a Domain Specific Modeling Language (DSML) built for modeling cyber, physical, and manufacturing component models, composing the component models, making architecture trade-offs using design spaces, and encoding test cases for various analysis domain tools. CyPhyML is defined using the MetaGME language in the Generic Modeling Environment (GME [6]).

A CyPhyML Component model contains interfaces (physical, structural, and data) of a physical entity or a controller, key parameters of the component, and the relationship between component level parameters and domain model parameters. For instance, a mass component can have a *manufacturing domain model*, a *geometric domain model* (CAD), and a *behavior domain model* (Modelica model).

The component model level parameters can affect all domain model parameters at the same time, i.e., if the mass has dimension and density parameters, then the CAD model and the behavior model are parameterized with exactly the same values respecting unit conversions.

When the CAD and behavior models are composed, all parameters will be consistent across all domain models. CyPhy Component models do not contain any internal details of the domain models; they capture information only about interfaces and links to the domain models.

A CyPhy *Component Assembly model* can contain any number of CyPhyML Components and other CyPhyML Component Assemblies, which together provide system and subsystem concepts. This language feature makes hierarchical composition possible through interfaces (ports and parameters). A full system model is often called a point design or a single design configuration.

A CyPhyML *Design Space model* can encode multiple design configurations (i.e., component assemblies) by using alternative and optional containers inside the design space. Design space models generate a discrete design space in the form of design configurations using the Design Space Exploration Tool (DESERT [5]).

For instance, if the design space contains a mass component, alternative mass components can be added (e.g. using different geometric sizes, material, etc.); if 3 options are added for the mass component, the design space will grow to 3 design configurations. If we have a mass, spring, and damper system (similar to a very simplified suspension assembly) and 3 options are

DOI 10.3384/ECP14096353

available for each, then the overall design space would be 27 configurations.

To solve the design space exploration problem, CyPhyML supports design space constraints that can be expressed as auto-generated range constraints, property constraints (e.g. component level parameter limits), visual constraints (e.g. compatibility between components/material or symmetry), or as Object Constraint Language (OCL) constraints. Constraints are used to prune the exponentially large combinatorial design space to a feasible and manageable set of configurations.

Once a CyPhyML Design or Design Space is built, we can define the evaluation of designs using CyPhyML Test Bench models. CyPhyML Test Benches are used to set up boundary and environmental conditions for designs in which they should be evaluated. Test benches also provide sufficient information and any additional models (e.g. stimulus, load, external 'test' components) to the system to make simulation and analysis possible with a domain-specific tool.

CyPhyML supports various types of test benches, including Dynamics (i.e., Modelica simulations), formal verification, CAD (e.g. composing the 3D model and computing center of gravity or mass), finite element analysis, computational fluid dynamics, blast, ballistic, conceptual manufacturing, detailed manufacturing, and reliability analysis.

In this paper we focus on *formal verification* and Dynamics (Modelica) *simulation test benches* only. CyPhyML Test benches contain a top level system under test (design or design space), input parameters that can change environment, load, stimulus conditions (test component parameters), and outputs called metrics.

2.2 Importing Modelica Models

CyPhyML Components have associated behavior models in the form of linked Modelica models. Only Modelica parameters and Modelica connectors need to be represented in the CyPhyML Component model. The behavioral model aspect of a CyPhyML Component can be viewed as a lightweight wrapper around a Modelica model, which can be built using the OpenMETA tool set and its editor GME.

Building the Modelica model interface representation in GME can be cumbersome and a time consuming activity. All information about the interface exists in the Modelica model, already including the following: model name, model type, connector names, connector types, parameter values (e.g. default value, minimum value, and maximum value), and class restrictions.

The user has to provide a set of Modelica models in textual form (.mo files or one .mo package). A wide set of Modelica models can be imported in an automated way as CyPhyML Components or CyPhyML Test Components using the OpenModelica Compiler (OMC) API. There is a seamless integration between the OpenMETA tools and the OMC API. The OMC API provides functionality to load model files and libraries (i.e., packages), query containment and inheritance relationship between types, and navigate through model elements using the abstract syntax tree.

The Modelica model importer has certain limitations and it does not support the entire Modelica language. Conditional ports and parameters, enumerated types, and parameterized ports (which can change their internal structure) are not supported. 'Replaceable' elements have a limited support, for instance models with fluid port connectors can be imported and the 'Medium' type is correctly set in the CyPhyML Component.

If the model or library does not conform to the Modelica Specification and/or the OMC API cannot load the package, then the automated import functionality is not available in the OpenMETA tools, requiring users to build the CyPhyML Components manually.

We are currently working on supporting a more complete set of the Modelica language and multi-fidelity models where one CyPhyML Component can be linked to more than one Modelica model and where the different Modelica models represent different level of modeling abstraction of the behavior of the physical component.

The OpenMETA tools already have a limited support for multi-fidelity component models, but they have to be built manually. For any CyPhyML Test Bench the component fidelity selection can be specified for a class of components, e.g., spring or damper component models.

2.3 Generating Modelica Models

Once a set of Modelica components are imported into the CyPhyML we can build design models, design space models and test-bench models. These models are composed through interfaces (i.e., connectors and parameters), which is sufficient information to generate composed Modelica models of test bench models for a design or for the entire design space.

The generated Modelica models preserve the hierarchical decomposition of the system and organize all generated models into packages and sub-packages based on the CyPhyML project structure to ease navigation in the generated model. Each generated component model, used in the design, 'extend' the referenced Modelica model and overwrites the parameters with the CyPhyML Component instance values.

From each dynamics CyPhyML Test Bench there are two generated models in the Modelica package that are used to run analysis. One of them is a simulation model and the other one is an augmented version of the simulation model for formal verification purposes.

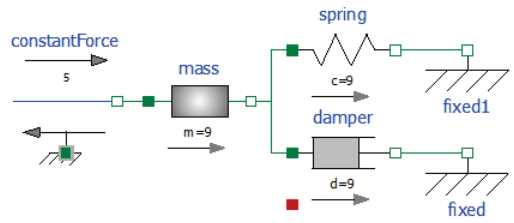

Figure 2. Mass-Spring-Damper in Modelica

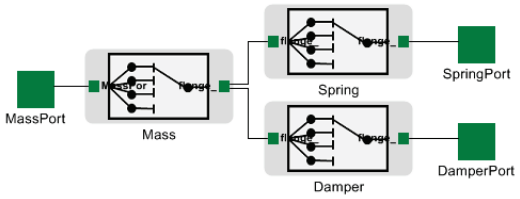

Figure 3. Mass-Spring-Damper design space in OpenMETA tools

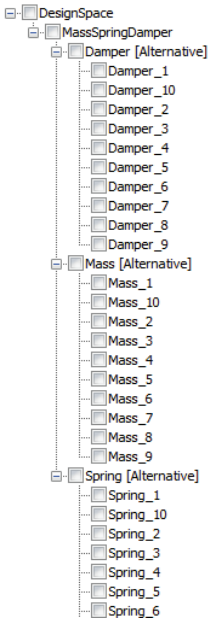

Figure 4. Mass-Spring-Damper design space tree and alternative options

We use a Mass-Spring-Damper (MSD) system, which contains Modelica Standard Library components, as a simple use case to show the workflow and the results of the formal verification tool for two configurations from a CyPhyML Design Space. Modelica model of the MSD system structure is shown in Figure 2. Figure 3 depicts the design space in the OpenMETA tools, where each component *Mass*, *Spring*, and *Damper* contains alternative components. The hierarchical structure and alternative options are shown in Figure 4. There are 10 alternatives in each design container, thus the design space generates 10x10x10 (1000) configurations. We have selected two configurations (#1 and #8) for further analysis by the verification tool.

Configuration #1 and configuration #8 have the same architecture, i.e., the same number and kind of components and the same connections among the components, but configuration #1 uses Mass 9 (m=9 kg), Spring 9 (c=9 N/m), Damper 9 (d=9 N.s/m) and configuration #8 uses Mass 3 (m=3 kg), Spring 8 (c=8 N/m), Damper 8 (d=8 N.s/m).

Figure 5 shows configuration #1 of a generated Modelica model for formal verification. The verification model inherits the simulation model and includes: the definition of the requirement status (success, unknown, violated), all physical limit definitions (e.g. `Limit1`: maximum absolute force cannot exceed a certain value) and requirements for the system, and defines all conditions under which the limits (e.g. `abs(Spring.f)>17`) and requirements are violated.

```
model verif_MassSpringDamperAuto2cfgs_cfg1
  extends CyPhy.TestBenches.MassSpringDamperAuto2cfgs_cfg1;

    // Requirement Definition
    type Requirement = enumeration(
      success,
      unknown,
      violated);

    // Limit-Checks definitions
    Requirement Limit1(start = Requirement.unknown, fixed=true)
    " Max absolute value limit-check on
      MassSpringDamper.Spring_1__Spring_9.flange_a.f";

    // Requirements-Checks


equation
    // Limit-Checks equations
    if Limit1 == Requirement.violated or
       MassSpringDamper.Spring_1__Spring_9.flange_a.f > 17 or
       MassSpringDamper.Spring_1__Spring_9.flange_a.f < -17 then

      Limit1 = Requirement.violated;
    else
      Limit1 = Requirement.unknown;
    end if;

    // Requirement-Metrics Checks equations


end verif_MassSpringDamperAuto2cfgs_cfg1;
```

Figure 5. Modelica model of MSD configuration #1.

2.4 Model translators and Job Manager

CyPhyML analysis model translators (i.e., analysis interpreters) are built to generate analysis packages from CyPhyML test benches, which contain domain tool specific input files, data structures, and scripts to perform the execution and collect results.

The OpenMETA tools can generate analysis packages for all test benches over the entire design space. This raises another scalability issue: executing all analysis packages may take significant time. In order to reduce the overall runtime, the META Job Manager [8] can run the individual/independent analysis packages in parallel either locally utilizing multiple CPU cores, or

DOI 10.3384/ECP14096353

on a remote compute cloud provided by the VehicleFORGE [8] platform. After analyses are executed, the results are stored locally.

2.5 Analysis Results

The raw analysis results are cumbersome and can be extremely difficult to compare. To address this issue, CyPhyML defines metrics for the key performance parameters of design configurations. These numbers, which are often driven by system requirements, provide the basis for design trade-offs and ranking, as well as for making decisions under specific circumstances about which design configuration is best.

Metrics are stored in a manifest file that contains the key information about the design configuration, test bench, and the projection of results. This single file is much smaller than the raw data and makes design space comparisons significantly easier. In general, we noticed roughly a three orders of magnitude size reduction when using only the compact manifest file (e.g. 15 GB of raw analysis data -> 10 MB).

The OpenMETA tools provide a data visualizer called the Project Analyzer. The Project Analyzer can be used locally in a web browser or deployed on VehicleFORGE (or another server). It loads all analysis data from the manifest files (no data from raw files is loaded) and provides different visualization techniques to display results, visualize requirements, rank designs based on the user's weighting preference on metrics, show physical limit violations on components, display constraint plots, show formal verification results, etc.

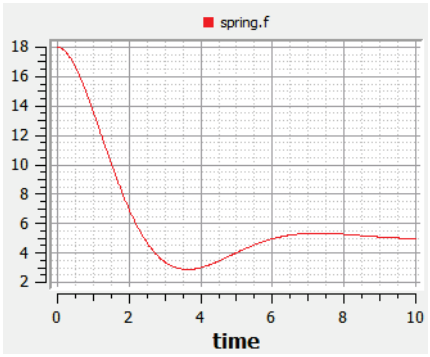

Figure 6. Mass-Spring-Damper simulation results configuration #1 force on the spring component

OpenModelica can be used to visualize the raw simulation results if needed. Figure 6 shows the force [N] on the spring component for design configuration #1.

The Project Analyzer provides various visualization techniques using different widgets to visualize the results over a design space. Figure 7 shows the parallel axes plot widget, where the vertical axes correspond to the metrics (velocities) and each colored line between the axes represent a design configuration. The requirements objective and threshold values are shown on the right hand side of the axes.

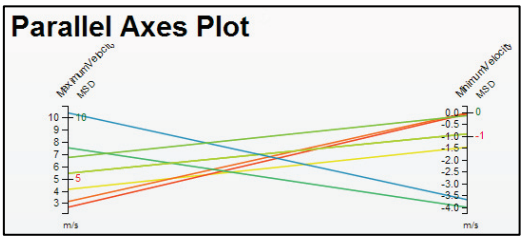

Figure 7. Project Analyzer parallel axes plot

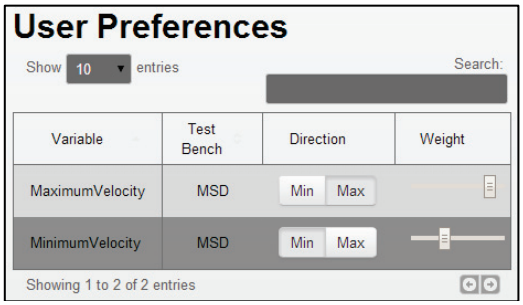

Figure 8. Project Analyzer user preferences settings

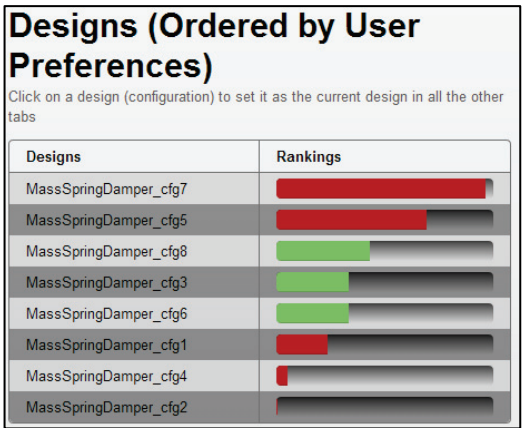

Figure 9. Project Analyzer designs by user preferences and color coded based on requirements

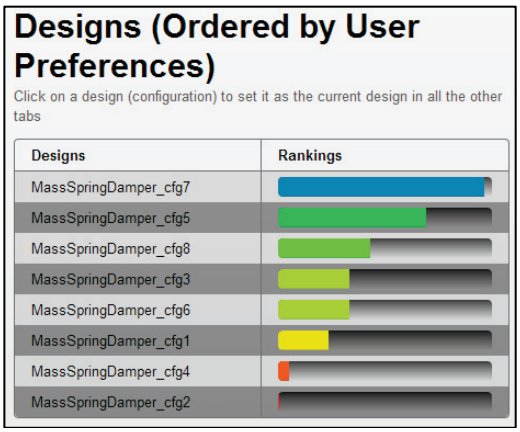

Figure 10. Project Analyzer designs by user preferences and color coded based on ranking

Users can set their weighting preference (Figure 8) for each metric value, which would determine the ranking

of the designs shown in Figure 9 and Figure 10. Designs on each widget can be color coded based on requirements (Figure 9), ranking (Figure 10), limit violations, or design scores.

2.6 Simulation and Verification of Generated Modelica Models

We have chosen to separate the verification and the simulation models in the generated code. We aim to run the simulations as fast as it possible, since running the test benches over a design space can take significant time even if parallel execution is used.

Using the OpenMETA tools and the parallel execution capability provided by the Job Manager we run hundreds of design configurations over tens of test benches. The simulation model does not need to contain verification properties and unnecessary auxiliary variables. Therefore, the simulation can first be executed, and then a post processing script can validate the limit violations and requirements on the simulation results.

This approach will give us a faster execution time for simulation models. The OpenMETA tools use OpenModelica to execute the generated simulation models.

Modelica models for verification are translated to Differential Algebraic Equations using the OpenModelica Compiler. The Mass-Spring-Damper configurations are translated to DAEs and then a formal verification tool analyzes both configurations.


```
"AnalysisStatus": "OK",
"TierLevel": 0,
"DesignName": "MassSpringDamperAuto2cfgs_cfg1",
"FormalVerification": [
  {
    "Source": "PARC",
    "Result": "FAIL",
    "ReqName": "Limit1",
    "Details": [
      {
        "GroupBody": [
          "Decrease MassSpringDamper.Spring_1__Spring_9.c",
          "Decrease roof.s0",
          "Increase floor.s0",
          "Decrease MassSpringDamper.Spring_1__Spring_9.s_rel0",
          "Increase MassSpringDamper.Damper_1__Damper_9.s_rel"
        ],
        "GroupTitle": "Changes suggested by QRM's symbolic differe
      }
    ]
  }
],
```


Figure 11. Verification results for MSD configuration #1.


```
"AnalysisStatus": "OK",
"TierLevel": 0,
"DesignName": "MassSpringDamperAuto2cfgs_cfg8",
"FormalVerification": [
  {
    "Source": "PARC",
    "Result": "UNKNOWN",
    "ReqName": "Limit1",
    "Details": []
  }
],
```


Figure 12. Verification results for MSD configuration #8.

The limit restrictions and requirements are the same for configuration #1 and #8. Figure 11 and Figure 12 depict the results of the formal verification results for configuration #1 and configuration #8 respectively. Section 3 and Section 4 describes the integrated formal verification method and reliability analysis in more detail respectively.

3 Qualitative Reasoning Module

The Meta tool suite includes a Qualitative Reasoning Module (QRM) which performs qualitative analyses of system behavior. In contrast to Modelica solvers which produce exact numerical results given exact numerical inputs and parameter values, qualitative simulators predict the possible time evolution of a system in qualitative terms.

Qualitative values are characterized by ranges, for example Q+ represents any possible positive value. Qualitative values can be demarcated with landmarks, for example `[l,u]` where the value lies between `l` and `u`. A qualitative analysis may show that a particular design cannot ever meet its requirements—something that is impossible to show with numerical solvers.

One challenge to more widespread use of Qualitative Reasoning is the lack of extensive qualitative model libraries. One cannot expect a designer to write their own qualitative models. Therefore we have spent considerable time and effort into automatically translating Modelica models into terms suitable for qualitative analysis.

Our translator starts with the exported DAE from OpenModelica. This has required significant extensions of model importers to qualitative algorithms. In addition, the DAE exporters have had to be extended to provide additional information. Qualitative reasoning requires declarative models. Any Modelica model used by AVM which is not purely declarative is being converted to declarative form manually.

Qualitative reasoning is most useful in early stages of conceptual design where the parameters, topologies, and requirements have not been completely articulated. Topologies which cannot possibly achieve customer requirements can be eliminated without having to determine specific parameters.

Qualitative analysis can also analyze a design which fails to meet some requirement and suggest qualitative parameter changes which will bring the design closer to meeting a requirement. The screenshot (Figure 13) illustrates analyzing a mass-spring-damper system to identify qualitative changes to meet a requirement.

DOI 10.3384/ECP14096353

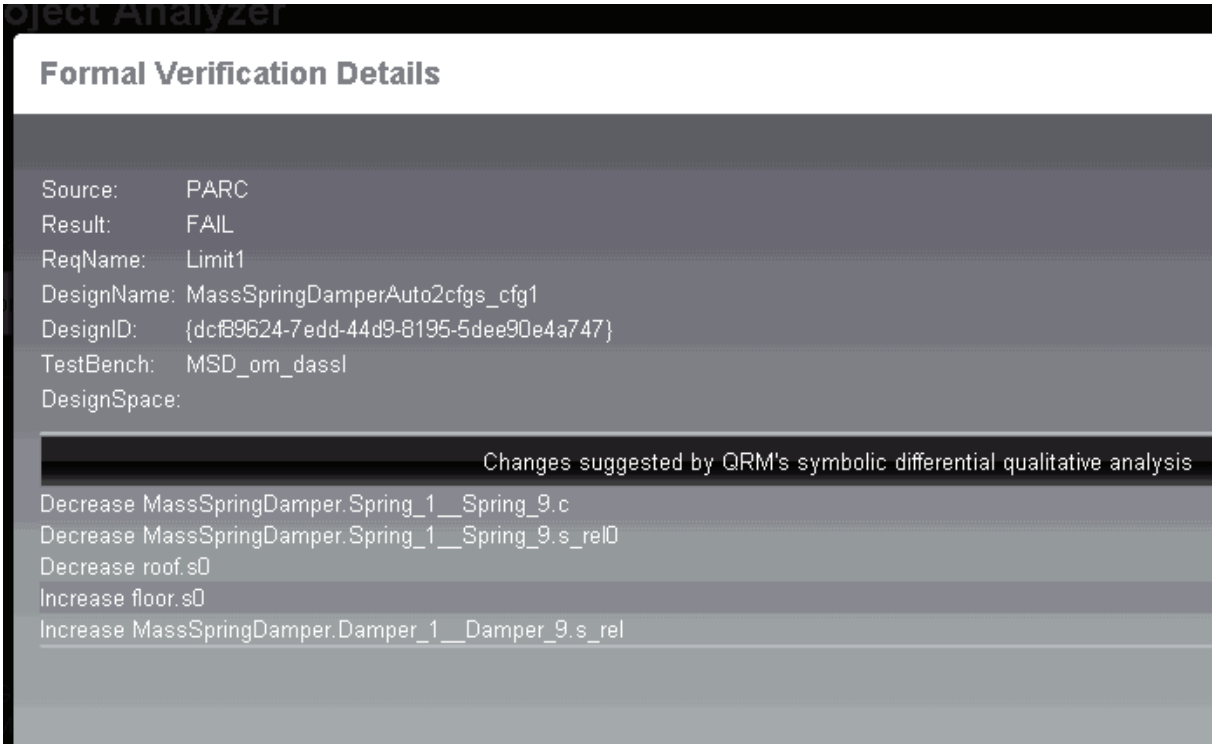

Figure 13. Using QRM for analysis of a mass-spring-damper system.

4 Reliability Analysis

The Meta tool suite includes a reliability analysis tool. This tool automatically allows a designer to evaluate the reliability of various components as well as various design configurations. The reliability tool has three major modules: (1) automatic construction of Modelica fault models, (2) determination of the fault probability distributions, (3) computing system reliability given (1) and (2).

Our fault augmenter takes a MSL model as input and automatically constructs its fault modes, which includes power port failures such as open and shorts as well as important parameter shifts. For each fault model, we construct damage maps which provide a probability density function for important parameters and is indexed by the type of material the particular component is constructed out of (e.g., steel), CAD properties, and Modelica variable values. The damage maps are constructed through a separate probabilistic process. More details can be found in [13]. In this paper we will focus on how the reliability tool is used by a designer (i.e., the third module).

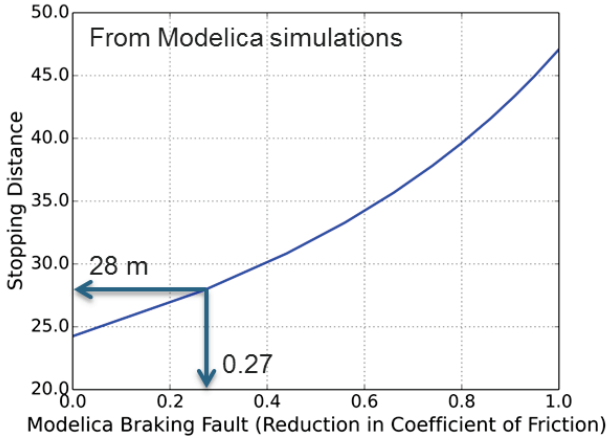

Figure 14. Braking distance / the coefficient of friction

Suppose a designer needs to choose brake in their design (vehicle drive train) that will meet its stopping requirement of 28 m from 60 kph. Given a fault augmented model, we can determine stopping distance by running multiple Modelica simulations.

From the Modelica simulations we can see that the stopping criterion fails after a fault amount of 0.27. Using the reliability formula and given a reduction in friction the actual physical damage is 0.85.

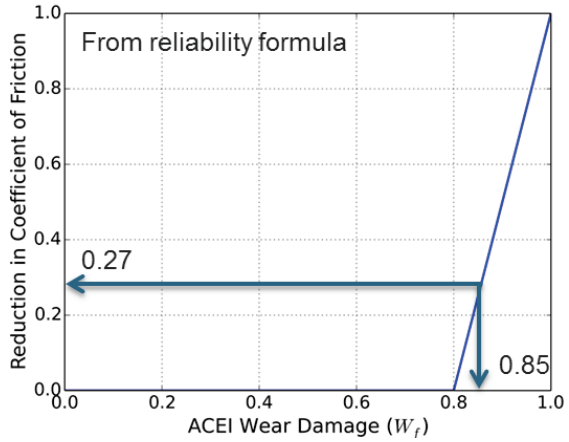

Figure 15. ACEI Wear damage / coeff. of friction.

Finally we refer to the damage map to determine the probability that the vehicle will meet its braking requirement after 75 missions.

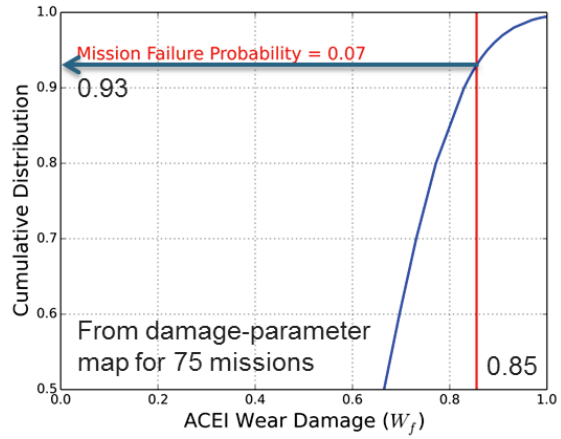

Figure 16. ACEI Wear damage / cumulative distribution.

With the reliability tool the designer can choose the component, requirement, number missions, desired probability of success, etc.

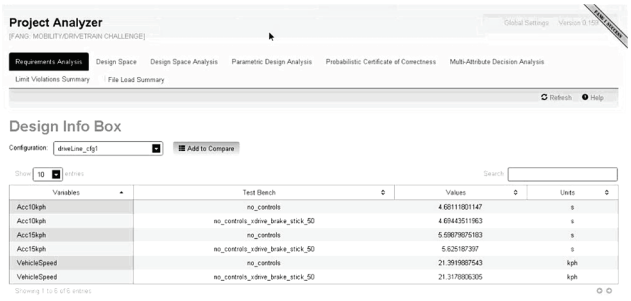

Figure 17. Reliability tool.

The needed Modelica simulations required to render these reliability calculations are expensive. Our approach is to pre-compute as much as possible. For example, the damage maps are all pre-computed. For the simple vehicle model analysis presented here we have pre-computed all Modelica simulations (and use interpolation) to enable the reliability tool to respond instantly. However, for complex novel designs the reliability calculations will take hours and possibly days on a single machine. Fortunately, reliability calculations scale linearly with the number of processors.

5 OpenModelica Tool Support

OpenModelica is used in four different places in the OpenMETA tool chain:

- importing Modelica models and associating them with CyPhyML component models,
- performing simulations of composed Modelica models, i.e., CyPhyML test benches,
- analyzing Modelica models and automatically adding fault modes, and
- extracting Differential Algebraic Equations (DAEs) for formal verification tools.

The OpenModelica compiler (Figure 18) has been slightly extended to facilitate integration with the META Tool chain.

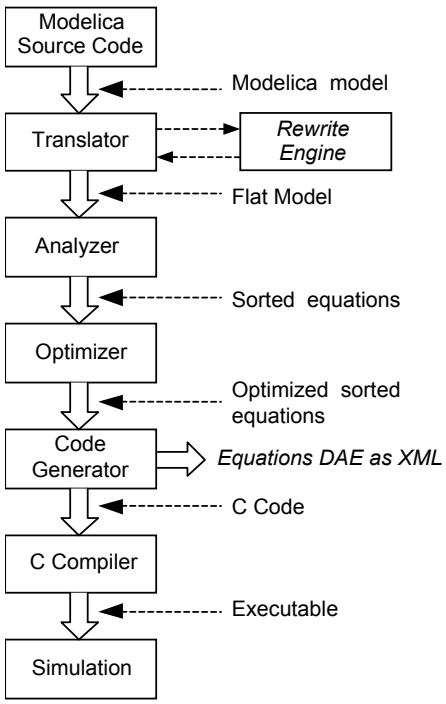

Figure 18. The OpenModelica compiler (OMC) structure and simulation execution. A rewrite engine and enhanced DAE XML output have been added for Meta Tool usage.

5.1 User-defined Rewrite Rules for Model Simplification

In order to make it feasible to apply formal verification to models they need to be simplified as much as possible so that their complexity is drastically reduced.

DOI 10.3384/ECP14096353

To support model simplification a *rewrite engine* for user-defined rewrite rules has been added to the OpenModelica compiler (Figure 18).

Note that this simplification could be applied by the formal verifier tool on the final DAE, but if the rewrite rules are applied as early as possible inside the Modelica compiler further simplifications can be discovered and applied.

The final model representation form as reduced and optimized symbolic equations is output in an XML representation for further processing by the Meta Tools QRM module.

The user defined rewrite rules have the form:

```
rewrite(old_expression, new_expression);
```

Note that `old_expression` and `new_expression` can contain special component references in the form of quoted identifiers starting with $, for example: '$1', '$2', '$x', '$y', etc.

The part of the expression tree where the special component reference appear is bound to that component reference.

As an example, consider the rule:

```
rewrite(
  abs('$1'),
  if ('$1' >= 0) then '$1' else -'$1');
```

which could be applied to an expression:

```
  abs(y + z)
```

In this case `$1` will be bound to `y+z` and the transformed expression becomes:

```
if ((x+y) > 0) then (x+y) else -(x+y)
```

The bounding operation is similar to pattern matching or unification in languages that support such features. Some examples of user-defined rewrite rules:

```
rewrite(
  abs('$1'),
  if ('$1' >= 0) then '$1' else -'$1');

rewrite('$1' ^ 2, '$1' * '$1');

rewrite(semiLinear(0.0, '$1', '$2'), 0.0);

rewrite(noEvent('$1'), '$1');

rewrite(
   Modelica.Fluid.Utilities.regStep(
    '$1', '$2', '$3', '$4'),
   if ('$1' > '$4') then '$2' else '$3');
```

The rules are loaded from a file given by the user and the rules are matched/applied to the expressions appearing in the abstract syntax tree.

Note that the application of the rules happen during semantic checking of expressions so that the resulting type before and after the application of the rule can be checked. In the cases where the bound expressions are arrays the operation is applied for each element, for example:

```
rewrite(
   Modelica.Math.Matrices.isEqual('$1',
     '$2', '$3'),
   '$1'=='$2');
```

applied to:

```
Modelica.Math.Matrices.isEqual({{x,y},
 {z,w}}, {{a,b},{c,d}}, eps)
```

will result in:

```
x == a and y == b and z == c and w == d
```

One can see that in some cases not all variables are used as for example `eps` above. For the purpose of formal verification the given expression is enough as the `eps` is used only for robustness of simulation.

6 Integrated OpenModelica Meta Tools Environment

The following summarizes the main capabilities of the integrated OpenModelica – META Tools environment:

- Parse Modelica models (OpenModelica compiler API called through Python) and import model interfaces (parameters and connectors) into the META tool chain.
- Run simulations of composed Modelica models using the OpenModelica (OMC compiler).
- Be able to formally *evaluate verification properties* of system designs using OpenModelica and verification tools QR/HybridSal (XML DAE).
- OpenMETA tools can *compose simulation models over a design space* including different architecture variations in an automated way.
- Verification problems and simulation models can be encoded as *test bench*, which can be evaluated over a *design space*.
- Using the OpenMETA tools the *JobManager* provides sufficient capabilities to utilize all CPU cores in the user's computer.
- The analysis simulation/verification can run locally or on a remote execution cluster.
- Simulation and verification results are collected and visualized through a common interface called *Project Analyzer*.

7 Related Work

Automated verification of dynamic behavior of design models against formalized requirements is described in

[10] and [11]. A prototype of an integrated tool chain for model based functional safety analysis is presented in [12].

8 Conclusions

This paper has presented an overview of the META tools for design space exploration and design verification, and their integration with OpenModelica.

The integrated environment currently has four main uses of OpenModelica: importing Modelica models into the META tool model structure, performing simulations within test benches, analyzing Modelica models and automatically adding fault modes and extracting equations (DAEs) for formal verification tools, e.g. the QRM using qualitative reasoning.

A prototype of the integrated tool framework is in operation, being able to generate and simulate thousands of designs in an automated manner.

9 Acknowledgements

This work was partially sponsored by The Defense Advanced Research Agency (DARPA) Tactical Technology Office (TTO) under the META program and is Approved for Public Release, Distribution Unlimited. The views and conclusions in this document are those of the authors and should not be interpreted as representing the official policies, either expressly or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

This work has also been partially supported by the Swedish Governmental Agency for Innovation Systems (Vinnova) within the ITEA2 MODRIO project, and by the Swedish Research Council (VR).

References

[1] Modelica Association. *Modelica—A Unified Object-Oriented Language for Physical Systems Modeling: Language Specification Version 3.2 rev 2*. Available at http://www.modelica.org, August, 2013.

[2] Modelica Association. *Modelica Standard Library 3.2 rev 1*. http://www.modelica.org. Aug. 2013.

[3] Peter Fritzson. *Principles of Object Oriented Modeling and Simulation with Modelica 2.1*, ISBN 0-471-471631, Wiley-IEEE Press. 2004.

[4] Peter Fritzson. *Principles of Object Oriented Modeling and Simulation with Modelica 3.3*, Accepted for Publication, Wiley-IEEE Press. 2004.

[5] Adaptive Vehicle Make. http://www.darpa.mil/Our_Work/TTO/Programs/Adaptive_Vehicle_Make__(AVM).aspx

[6] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomasson, G. Nordstrom, J. Sprinkle, and P. Volgyesi, "The Generic Modeling Environment", Workshop on Intelligent Signal Processing, Budapest, Hungary, May, 2001.

[7] Sandeep Neema, J. Sztipanovits, G Karsai, and K. Butts. Constraint-based design-space exploration and model synthesis". In *Embedded Software*, R. Alur and I. Lee, eds., pp. 290–305, Vol. 2855 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2003.

[8] Laszlo Juracz, Zsolt Lattmann, Tihamer Levendovszky, Graham Hemingway, Will Gaggioli, Tanner Netterville, Gabor Pap, Kevin Smyth, Larry Howard. VehicleFORGE: A Cloud-Based Infrastructure for Collaborative Model-Based Design., In *Proc. of 2nd International Workshop on Model-Driven Engineering for High Performance and CLoud computing* (MDHPCL), MODELS 2013, Miami, Fl., USA, 2013.

[9] Raj Minhas, Johan de Kleer, Ion Matei, Bhaskar Saha, Daniel G. Bobrow and Tolga Kurtoglu. Using Fault Augmented Modelica Models for Fault Diagnostics. Submitted to Modelica'2014. Dec 2013.

[10] Wladimir Schamai. *Model-Based Verification of Dynamic System Behavior against Requirements - Method, Language, and Tool*. Linköping Studies in Science and Technology, Dissertation No. 1547, www.ep.liu.se, Nov 12, 2013.

[11] Wladimir Schamai, Philipp Helle, Peter Fritzson, and Christiaan 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)*. In conjunction with MODELS'2010. Oslo, Norway, Oct 4, 2010.

[12] Lena Rogovchenko-Buffoni, Andrea Tundis, Muhammed Zoheb Hossain, Mattias Nyberg, Peter Fritzson. An Integrated Tool chain For Model Based Functional Safety Analysis. Accepted to *Journal of Computational Science*, June, 2013.

[13] Johan de Kleer, Bill Janssen, Daniel G. Bobrow, Tolga Kurtoglu, Bhaskar Saha, Nicholas R. Moore and Saravan Sutharshana, Fault Augmented Modelica Models, *24th International Workshop on Principles of Diagnosis*, Jerusalem, pp. 71-78, 2013.