MULTI-MODEL LANGUAGE SUITE FOR CYBER PHYSICAL SYSTEMS

FINAL REPORT

Sandeep Neema, Ted Bapty, Janos Sztipanovits

Institute for Software Integrated Systems, Vanderbilt University

March 31, 2013

Contract Number: FA8650-10-C-7075

Agency: AFRL

CONTENTS

1 INTRODUCTION

The AVM META projects have developed tools for designing Cyber Physical (or Mechatronic) Systems. Exemplified by modern Amphibious and Ground Military Vehicles, these systems are increasingly complex, take much longer to design and build, and are increasingly costlier. The vision of the AVM program is to revolutionize the design methodology of such systems and reduce the design time to 1/5th of the traditional systems engineering V methodology (MIL – STD 499).

The META tools realize this vision by advancing a novel design flow geared around the following core concepts:

- Component-Based Design enables design cycle compression by reuse of existing technology and knowledge, encapsulated in integratable and customizable components that can be rapidly used in a design. Components in CPS are heterogeneous, span multiple domains (physical – thermal, mechanical, electrical, fluid , .. and computational – software, computing platforms), and require multiple models to soundly represent the behavior, geometry, and interfaces, at multiple levels of abstractions. The META Language allows creating multi-model multi-domain representation of CPS components that are composable by design.
- Design Space Construction is supported by the META Language using concepts to represent design choices and parameterized components. These constructs in META enable a designer to systematically engineer a flexible and comprehensive design space for sub-systems and system that can be explored for satisfying product-specific requirements. The design spaces for subsystems and systems are assets that encapsulate design knowledge, which can be reused in a context different for which it was originally created.
- Multi-Scale Design Space Exploration incorporates multiple methods that trade accuracy with computation time for exploring the large design spaces. A combinatorial design space exploration tool (DESERT), rapidly prunes design space using highly scalable constraint satisfaction methods over static properties of components and designs (i.e. weight, power, cost, …). Higher fidelity, higher computation time methods such as qualitative reasoning, ODE simulations, are used to further explore and reduce the design space, iteratively converging over to solutions of interest, given a set of requirements.
- Testbenches for Design Evaluation capture requirements in a form that can be automatically evaluated for a system-under-test, using a large set of domain-specific analyses – ranging from hybrid dynamics simulation, software platform timing simulation, geometric parameter evaluation, finite element analysis, probabilistic certificate of correctness, qualitative simulation, among others. The model composition tools included in META operate over defined testbenches and synthesize artifacts necessary for executing analysis in domain tools such as Dymola, Pro-e/Creo, Truetime, Qualitative Envisionment, Abaqus, etc..

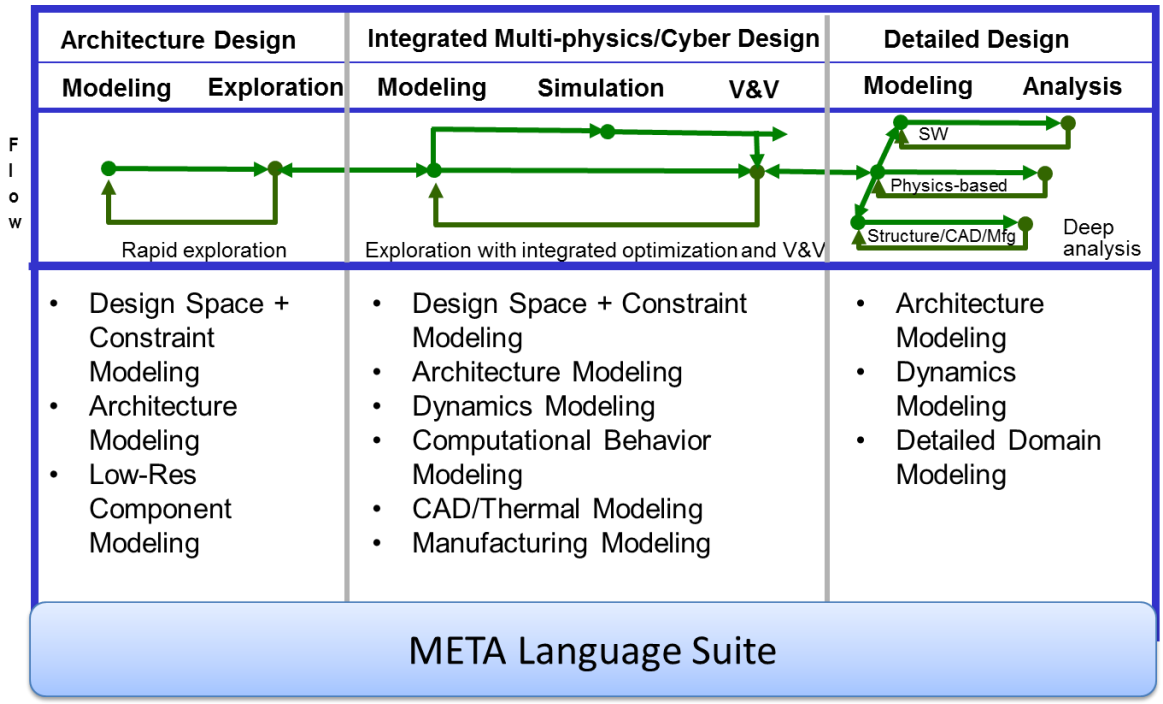
The META design flow and tools are built around the META Language, labeled the Cyber Physical Modeling Language (CyPhy). CyPhy is a model integration language which integrates models from different domains in a semantically sound manner that enables reasoning for correctness of models and modeling languages. This report describes the development of the CyPhy and related specifications and tools under the META Language contract (FA8650-10-C-7075).

2 PROJECT OVERVIEW

The charter of project, as envisioned in the proposal, was to develop a multi-modeling language suite for design and synthesis of Cyber Physical Systems.

2.1 META LANGUAGE ENABLES META DESIGN FLOW

The META design flow (separate contracted effort) articulates a design methodology and the associated tool flow for the CPS system design. The figure below summarizes the key elements of the META design flow, and motivates the key functionality that was needed in META language to enable the design flow.



The META design flow involves three core group of activities:

1. Initial Architecture design involves modeling and rapid Exploration of early design space sketched out with the system requirements. These activities in design flow require that the META language includes concepts for modeling system design space and constraints, enable representing the key architectural variants that can broadly support the customer requirements. The early architecture exploration also requires low compute intensity methods that can allow examination of lots of design options. The META language needs to support modeling low-resolution components to enable coarse grain exploration.

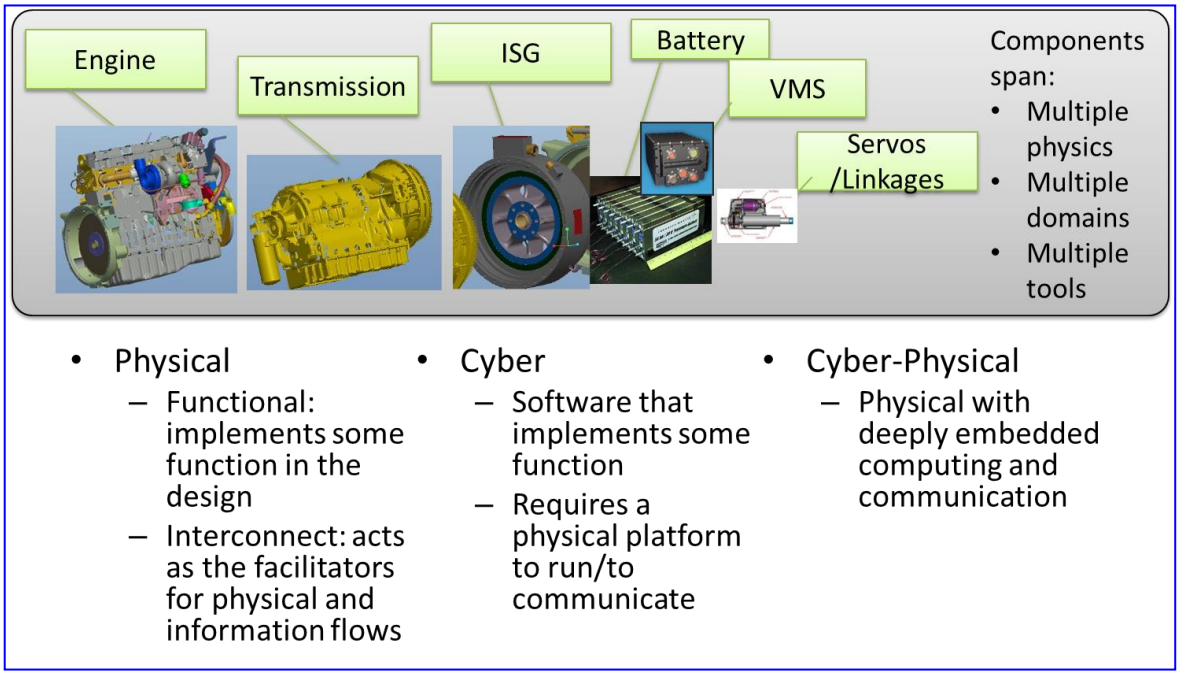
2. The Integrated Multi-Physics and Cyber Design stage expands upon the broadly identified architecture, and refines them with integrated design of physical and cyber components and conducts relevant tradeoffs. These activities in addition to modeling of the design space and

constraints require dynamics modeling for progressively refined performance simulation of the system. This phase also requires support for computational modeling to analyze the behavior and interaction of software with physical components. Geometry and geometry-driven analyses are central to the Physical nature of CPS, and consequently the modeling language suite needs to support CAD and derivative analysis such as thermal, FEA, and allow evaluation of designs for manufacturability.

3. The Detailed Design stage involves further refinement, and analysis, of designs leading towards production, which shifts the emphasis of modeling capabilities from design space to domain model elaboration.

2.2 META LANGUAGE REPRESENTS MULTI-MODEL MULTI-DOMAIN COMPONENTS

In addition to the diversity of the design and modeling activities, the META Language Suite needs to facilitate representation of a diverse range of cyber physical components.



The components that constitute a typical cyber physical system such as a military ground vehicle span a broad range from commodity physical components such as nuts and bolts, to large complex dynamical components like Engines, Transmissions, Sensors, Actuators, and Controllers. These components can generally be categorized as:

1. Physical – components consist purely of electro-hydro-mechanical elements with little or no programmability. Examples of such components include transmissions, differentials, gears, clutches, starter-generators, servos, among others. These can be further categorized as: functional – implementing a function in the design, or interconnect – that act as facilitator for physical energy flow or provide linkages such as nuts, bolts, pipes, and tubes.

2. Cyber – components are software components that require a computer processor to run, and implement some function such as the Vehicle Management Software, or controller algorithms implemented in software

3. Cyber-Physical – components cross-cut cyber and physical domains, such that they are physical and implement some function, however, contain deeply embedded computing and communication functions that enable configuration and control of the designed function. Modern combustion engines are a good example of cyber-physical components, in that they include programmable controllers that will interface with rest of the vehicle management system over communication buses (such as CAN and TT/FlexRay), and allow optimizing torque delivery by controlling air/fuel mixture and valve timing for optimal combustion.

The examples depicted above also illustrate the fact that META components span across different energy and physics domains. A combustion engine, for example, turns chemical energy into rotational mechanical energy, a battery delivers electrical energy from stored chemical energy, while an integrated starter generator (ISG) delivers mechanical rotational energy from electrical energy.

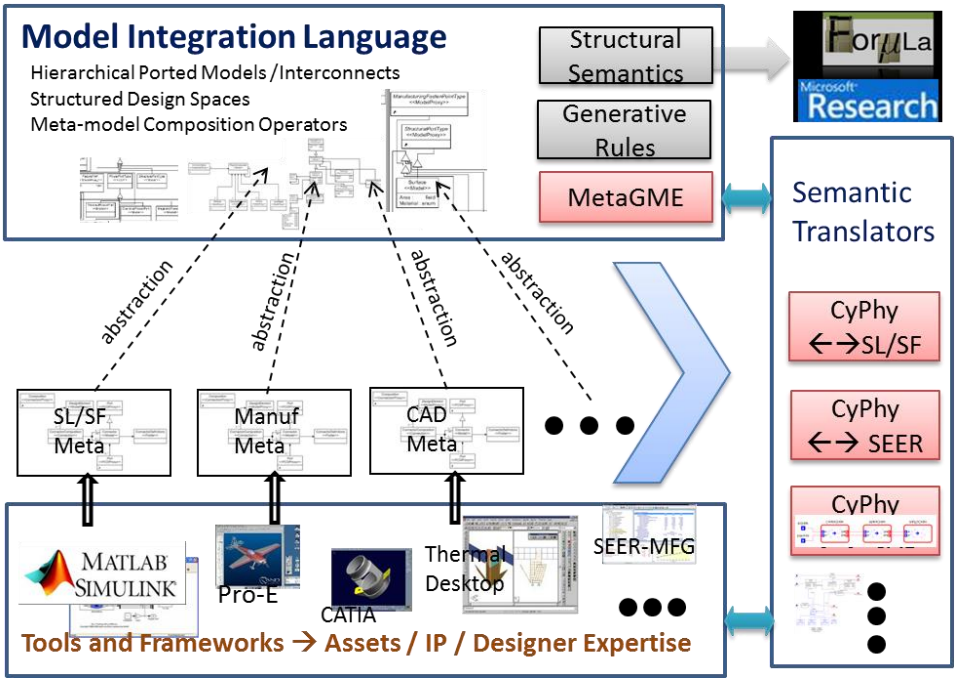
Moreover, the components depicted in figure require multiple models to describe and analyze their behavior. A combustion engine has a CAD model which represents the physical geometry including mass distribution, center-of-gravity, a dynamics model will describe the performance of the engine as a function of the driver and torque demand, a thermal model will describes heat generation, distribution, and dissipation as a function of the driver and torque demand.

Furthermore, these different models are often developed in different domain tools i.e. ProE/CREO or SolidWorks tools are used for CAD modeling, while Dymola and Simulink might be used for modeling dynamics. Often these models constitute an asset base of different engineering organizations, and have been developed with significant time and resource investment.

These motivating and constraining factors had a strong impact on the design of the META Language. The META Language had to be designed to represent components that are "Heterogeneous, Multi-Physics, and Multi-Model", in such a way that it could leverage and integrate existing model assets in domain tools.

2.3 META LANGUAGE IS A MODEL INTEGRATION LANGUAGE

The consequence of these factors was that we developed the META Language that we call CyPhy as a *Model Integration Language*. A Model Integration Language is a thin layer wrapping language that wraps the domain models and exports only the key interface and parameters that are relevant for integration. The wrapping maintains the link to the domain model – to allow integration in the domain tool. The integration language has a very small set of native modeling constructs by design. The native construction includes concepts such as hierarchical ported modules and interconnects, structured design spaces, and includes a variety of meta-model composition operators which enables systematic integration across different domain modeling languages.



The integration is done in a manner that abstracts the key properties and interfaces from the domain models that are relevant for integration across domains. These constitute the key variabilities, or design parameters that must be reasoned about in a multi-domain context. For example, when modeling system architecture the detailed and exact geometry may not be important, however, the key concepts of relevance are the join interfaces, surfaces and constraint with which components must be physically attached to each other. A systematic linkage of the abstractions and modeling concepts, automatically enables the projection from architecture models back into the domain models.

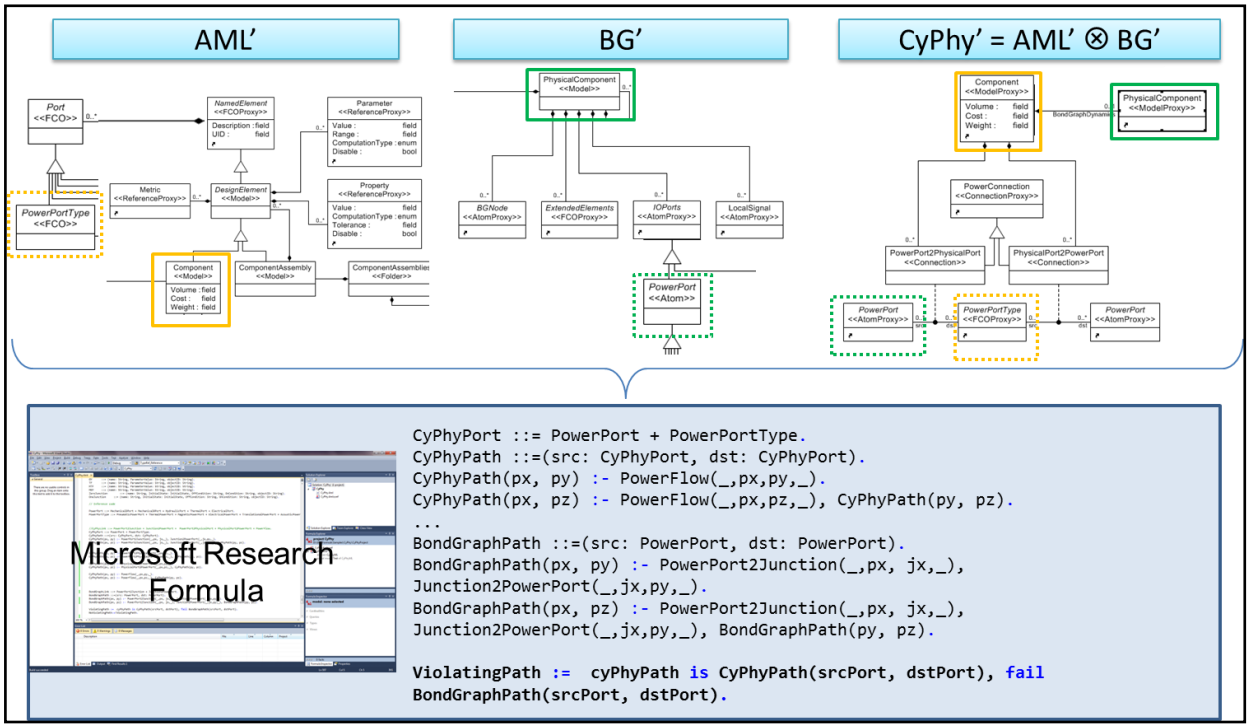
The Domain Tools and Frameworks depicted in the figure above are rich engineering infrastructures that were developed with significant investment, and have accumulated a large volume of Design Assets, Intellectual Property, Designer Expertise. The Model Integration Language approach enables reuse of these assets in the form of a META Component Library, and when systems are built using the components, the Model Integration Language approach allows to project the integrated models back into to the Domain Tools and Frameworks to analyze, visualize and refine the design.

A model integration language approach also allows to opportunistically link and add new design languages on demand, enabling an open language framework, that allows to adapt languges to accomodate evolving needs of design flows.

2.4 META LANGUAGE COMPOSITIONAL SEMANTICS

A major challenge in realizing a model integration approach, relates to the heterogeneous semantics of the modeling languages integrated together. The project address this challenge by formally specifying the semantics of the integrated domain languages, as well as formally specifying

the composition semantics. The figure below illustrates the compositional semantics of integrated the Dynamics Modeling Language with the Architecture Modeling Language



The figure depicts a subset of the metamodel (metamodels are definition of modeling languages using a UML class diagram notation) of the Architecture Modeling Language (AML). The figure also shows a subset of the Bond Graph language, which is a multi-physics modeling language. The complete Bond Graph language is pretty large and complex, however, for integration with architecture language the core concepts that are relevant for integration are abstracted out. The *PhysicalComponent* is a term in Bond Graph notation referring to a component, and *PowerPort* are the interface of BondGraph components. In the integration language the composition is accomplished as follows:

a) a *PowerPortType* is defined in the AML, and is inherited from the *Port* concept in the AML. This allows for creation of power ports in architecture component model, and

b) a containment relation is established between the AML Components and Bond Graph components, which allows embedding Bond Graph components in architecture components, and

c) the *PowerPort* of Bond Graph is linked with *PowerPortType* in AML

However, while this enables drawing Bond Graphs within Architecture model, several crucial question remains regarding the semantics and well-formedness of compositions. In the META language project these questions are addressed by use of FORMULA, a constraint logic programming environment, developed at Microsoft Research. The FORMULA tool allows specification and reasoning over well formedness of domain composition.

3 RESULTS

3.1 META SEMANTIC BACKPLANE

The Semantic Backplane includes modeling languages, models and tools for the semantic integration of Domain Specific Tool Chain (DSTC) configurations. The semantic integration is performed by

1. *Metamodeling* - defining structural and behavioral semantics of domain specific modeling languages
2. *Metamodel Analysis and Verification* - composing and relating DSTC-level domain specific modeling languages) and
3. *Metagenerators* - automatically generating model translators from formal specification of relationship between modeling languages.

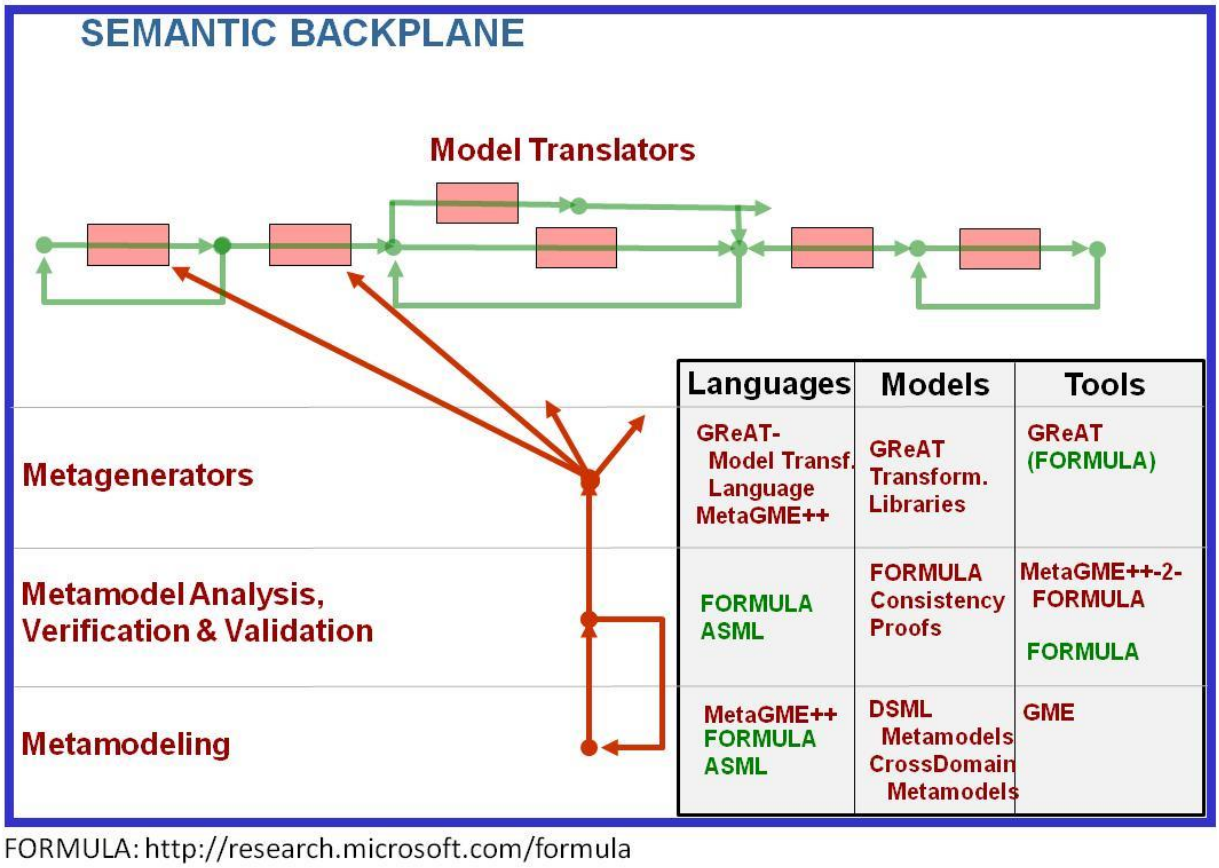


Figure 1: Semantic Backplane Overview

Tools and methods developed for the Semantic Backplane are not targeting the general engineering users: these are for a relatively small group of specialized experts responsible for the semantic integrity of the evolving domain specific tool chains.

An essential element of the Vanderbilt MIC tool suite is that most of the Semantic Backplane tools are "metaprogrammable" and used both in the Semantic Backplain and DSTC levels. In the followings we summarize the delivered components.

Metamodeling provides the formal specification of the semantics of the META modeling language suite.

3.1.1 METAMODELING LANGUAGES

1. *MetaGME++:* the mature MIC metamodeling language MetaGME (a variant of UML class diagrams and OCL) extended with generative constructs. MetaGME++ is used as metamodeling language for all MIC metaprogrammable tools. It has well established relationship with various standards, such as MOF, EMF, OWL and others.

2. *FORMULA:* constraint logic programming language developed by Microsoft Research. FORMULA is used as formal language for defining the structural semantics of *MetaGME++* and domain specific modeling languages defined using *MetaGME++*. (MSR and Vanderbilt ISIS collaborates in evolving FORMULA; e.g. current work expands the logic used in FORMULA with metric first order linear temporal logic).

3. *ASML:* a language variant for the Abstract State Machine (ASM) formal framework. We use ASMs as common semantic domain for specifying discrete behavioral semantics of modeling languages. ASML was selected because of its availability in the Visual Studio tool suite. (We expect that in the future we migrate to FORMULA as the supporting theory evolves). ASML-based behavioral semantics are operational specifications (as opposed to denotational), therefore they are executable and suitable for generating reference traces.

4. *DE:* lumped parameter differential equations as a common denotational semantic domain for a wide range of continuous time dynamics. We use a syntactic form that can be easily transformed. DE's provide a bridge towards symbolic mathematics tools developed for order reduction. The provided semantics for continuous dynamics is independent from simulation algorithms.

The metamodeling languages listed above are part of the deliverables. We expect that the metamodeling languages will continue to evolve beyond this project as an overall consolidation in the practical use cases for semantics. We are also investigating interesting other alternatives such as *BIP* (developed by Joseph Sifakis – 2008 Turing Award Laurate) for capturing interaction semantics among cyber components.

3.1.2 METAMODELS

Metamodels are models of domain specific modeling languages described using metamodeling languages. Their goal is to capture the formal structural and behavioral semantics of modeling

languages. The Semantic Backplane includes the *CyPhy Metamodel Library* that integrates semantic aspects of a given configuration of the META DSTC.

Being a model integration language, CyPhy includes a core set of language constructs for model and design space integration as well as an evolving suite of abstracted (sub)languages imported from various META tools. The abstracted sublanguages are the simplest possible well-formed subsets of the domain specific modeling languages of constituent META tools – still sufficient for capturing cross-domain interactions (structural and behavioral). Abstracting sublanguages for multi-model integration from bloated and complex domain languages is an important step toward making META DSTC-s practical.

At this point, the *CyPhy Metamodel Library* includes metamodels for the following sublanguages:

1. *ADML (Architecture Design Modeling Language)*: represents hierarchical component architectures and typed interfaces. Precise relationship is being defined between ADML and component modeling sublanguages of various standards or frequently used modeling languages, such as SysML (in progress), AADL (planned) and SL/SF. This relationship is defined as model transformation in GReAT (the MIC tool suite graph model transformation specification language) – and in some cases in FORMULA.

2. *ADSML (Architecture Design Space Modeling Language):* extends the design modeling languages with constructs for design space modeling, allowing traditional design languages to capture design spaces instead of just point designs. The extensions come in the form of introducing design containers with model structure variability such as Alternatives, Optional, and variable cardinality containment, as well as Parameterization of design elements. Introduction of these design space extensions at all levels within the design hierarchy provides a powerful and compact mechanism of representing very large design spaces.

3. *(Extended) BGML (Bond Graph Modeling Language):* is a multi-(energy/physics) domain formalism for representing lumped parameter dynamics of physical systems. A Bond Graph represents energy flow across systems in an energy domain neutral manner. Hybrid Bond Graphs are also able to represent hybrid dynamics with the aid of switched junctions, and support derivation of causality relation across systems.

Beyond the core model and design space integration language elements, CyPhy has been complemented with the following abstracted sublanguages imported from integrated tools:

1. *Simulink/Stateflow Interface Language:* The cyber aspects, specifically the controller design, are captured using Simulink/Stateflow models. The CyPhy metamodel integrates an abstracted Simulink/Stateflow metamodel, capturing the input, output, and parametric interface of Simulink models and defines associations with CyPhy components and component interfaces.

2. *Embedded Systems Modeling Interface Language (ESMoL):* The ESMoL language defines software components, computation and communication platform, and allocation of software components on platform. CyPhy metamodel defines the relation of software components with CyPhy components and allows defining the sensing, actuation, and control path by

specifying associations between energy interface of physical components, sensors and actuators with data interface of software components.

3. *CAD Constraint ML:* represents geometrical constraints (axial alignment, surface placement, between CAD components (linked into CyPhy components) and allow derivation of CAD assemblies with a network of geometric constraints

4. *Manufacturing (Cost) ML:* represents manufacturing cost drivers for buy and make parts. These drivers include factors such as parts types, complexity, and counts, join types, complexity, and counts for part assemblies. The Manufacturing ML is integrated within CyPhyML allowing associating manufacturing cost parameters with CyPhy components.

5. *Hydraulics ML (in progress):* is an abstraction of hydraulics systems modeling primitives as used for modeling Hydraulic systems in COTS tools (Boeing ICCA). These abstractions are being linked into the Fluid aspect of the CyPhy component model.

The metamodels above are represented in MetaGME++ and translated for verification and validation to FORMULA. (We do not expect the verification step fully completed by the end of September. Rather, we expect that the CyPhy Metamodel Library will continue evolving during the tool maturation period and beyond following the evolution of the META tool chain.)

3.1.3 METAMODELING TOOLS

1. *Generic Modeling Environment (GME)*: Vanderbilt's metaprogrammable modeling tool is the modeling environment for MetaGME++. Except the newly implemented support for the generative extension of MetaGME, the tool is mature and has been tested in major academic and industrial projects. GME is open source and distributed for research as well as commercial use.

2. *Unified Data Model (UDM)*: is a metaprogrammable API tool that provides API-s to programmatically manipulate domain-specific models built using GME (persisted in GME's native format or conformant XML). UDM is open source, has multiple programming language support (Java, C++, .net, Python), is mature and tested in various academic and industrial projects.

3. *GReAT:* is a Graphical modeling environment (and associated toolset) for formally defining (modeling) Model Transformations as Graph Rewriting specification over Domain Meta Models. The model transformations defined with GReAT can be interpretively executed for rapid prototyping, or compiled into executable specifications for performance. The formal definition provides opportunities for verifying the transformation, and allows for systematic evolution of the model transformation as the domain metamodels evolve.

3.2 CYBER PHYSICAL MODELING LANGUAGE (CYPHY)

The CyPhy Modeling Language is defined using MetaModels (described earlier). The CyPhy Metamodel is delivered as a deliverable of this project. This section documents the core concepts of CyPhy using sections of CyPhy metamodels.

3.2.1 COMPONENTS

Components in Cyphy are the basic units for composing system design. Components are self-contained models representing a physical or software part of the system. As an atomic component, they are not intended to be further subdivided at the level of representation in CyPhy, but can be used as a standalone part.

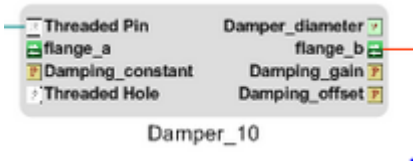


FIGURE 1: EXAMPLE COMPONENT MODEL

The component model represents several things about the actual component, including its physical representations and connections, its dynamic behavior, and numerical properties. The component in figure 1 shows several connections for structurally connecting (Threaded Pin & Hole), dynamically connecting (flange_a/b), and parameters (Damping Constant,...). These aspects are:

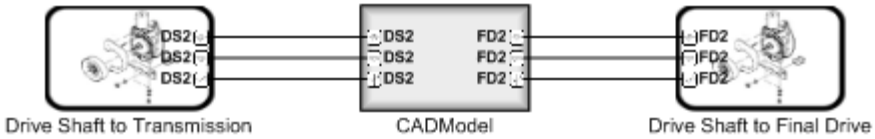


FIGURE 2: PHYSICAL – INSIDE THE STRUCTURAL ASPECT

- Physical implementation: The component will have a 3D shape, and various physical properties, such as mass, center of gravity, 3D geometry(CAD). As the components will be interconnected into *assemblies*, subsystems and systems, the *interfaces* are carefully defined to permit *composition* of models. The physical properties of the model are shown in the *Structural Aspect* of the model.
- Dynamics: The component will have behaviors one or more domains (e.g. Electrical, Thermal, Mechanical-Rotational, Mechanical-Translational, Hydraulic, etc. Dynamics is expressed in the Modelica language, which uses a mix of Causal (directional input or output is assigned to each port) and Acausal (power flows either direction based on its context, as in most physical systems).
- Cyber: The software is a critical part of the cyber-physical system design, with many components having a physical, dynamic, and software implementation. The Cyber aspect captures the software representation. The Cyber aspect is intended primarily for specifying controller logic for the system. Controllers can be specified in a combination of state diagrams and signal flow. Software is automatically generated from these models.

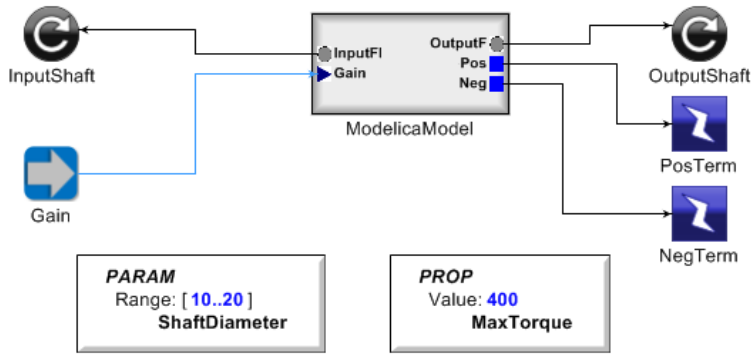


FIGURE 3: EXAMPLE DYNAMICS MODEL

In summary, components are multi-domain and multi-model, include interfaces for composition, have properties for informational and analytical evaluation, and can be parameterized.

3.2.2 DESIGN SPACES

Using components and assemblies allows the designer to capture a single design architecture, with a single choice of components. This has several drawbacks:

- Requirements often change during the design process, sometimes necessitating a redesign.
- Component and subsystem behavior is discovered during the design process, and the best choice of architecture and components may not be apparent until late in the design process.
- The design is applicable to a single target use, and can require substantial rework for other applications.

Instead, CyPhy/OpenMETA offers the concept of a ***Design Space***. The design space allows the models to contain multiple alternatives for components and assemblies. Any component or assembly can be substituted for another component or assembly with the same interface.

The editor offers a simple syntax for capturing design options. A design alternative container is created with an interface matching a component and the component is placed inside and wired to the external interfaces (there is a tool to automatically do this). Additional alternative components (or assemblies) are added to the alternative design container.

The semantics of this construct are to choose one of the internal components in place of the alternative container.

The design space is the combination of all options of all alternatives. Consequently, the design space can get very large (i.e. Design space size is # Alt1 * # Alt2 * # Alt3 *...). While this is a powerful mechanism to expand the range of designs under consideration, a mechanism is needed to limit the design space to a manageable size. For this purpose, design space constraints can be specified, and used by the DEsign Space ExploRation Tool (DESERT).

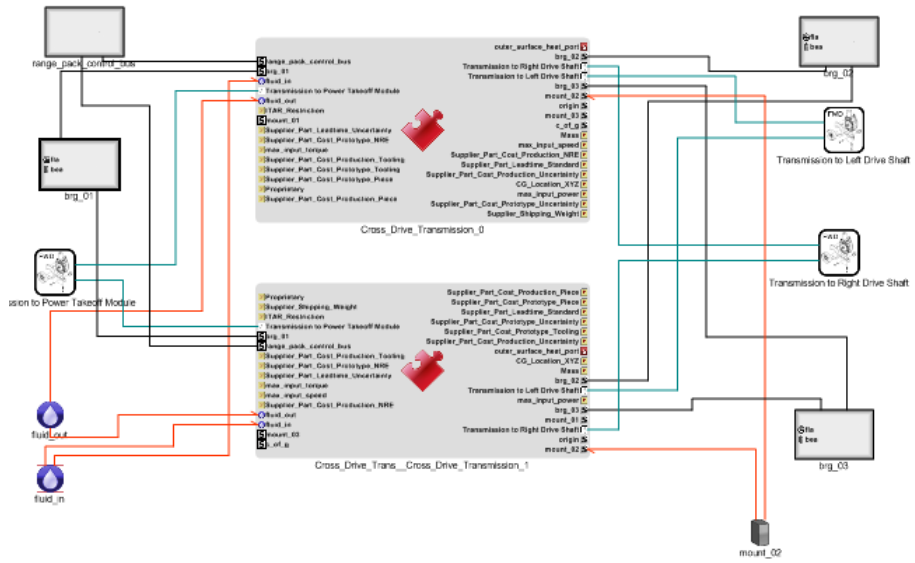


FIGURE 4: EXAMPLE DESIGN SPACE ALTERNATIVE

Design space constraints are simple, static operations/equations that can be specified on the properties or identities of components or assemblies in the design alternative space. Operations on the properties such as total weight and cost, thresholds on a component property (e.g. TRL > 3), or identity (e.g. All wheel types must match – do not mix and match)

FIGURE 5: EXAMPLE CONSTRAINTS

The DESERT Tool uses scalable techniques to apply these constraints to very large design spaces to rapidly prune the design space to a manageable size. The figure below shows the design space for the simple drivetrain. Prior to applying constraints, there are 288 configurations. After, there are 48. Typical design spaces can easily reach 10B configurations. After proper constraint application, these can be reduced to 1000's.

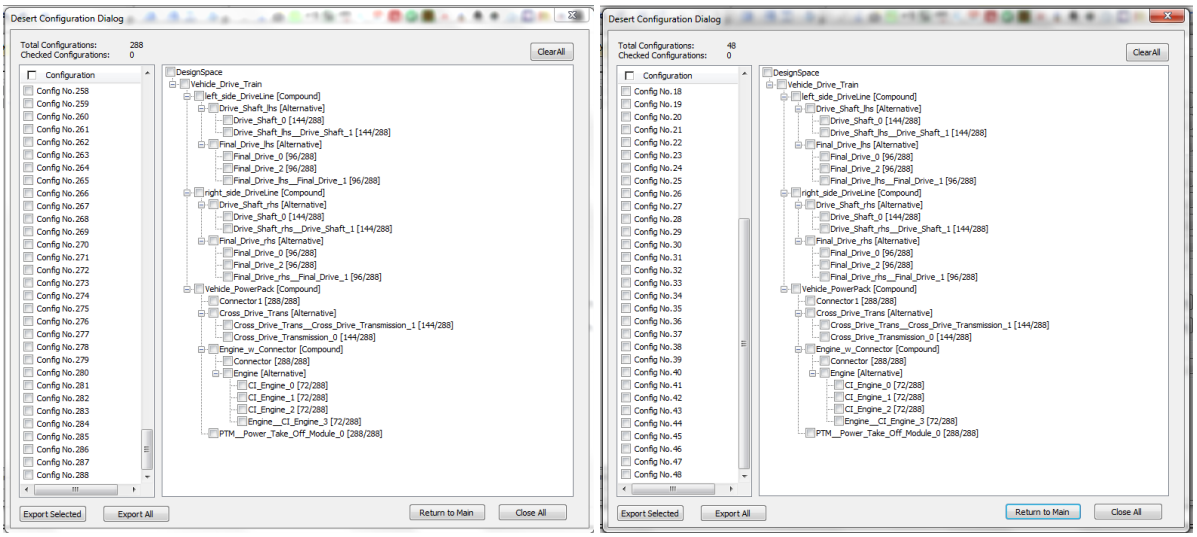


FIGURE 6: DESIGN SPACE EXPLORATION. BEFORE AND AFTER CONSTRAINT APPLICATION

Design space creation and exploration is a process of expansion and contraction of the design space. It can be a powerful tool to build adaptable, flexible designs.

3.2.3 DESIGN EVALUATION (TESTBENCH)

While application of constraints can eliminate design alternatives based on simple, static properties, much of the system behavior (desirable and undesirable) emerges from the dynamic interaction between components. These interactions occur across and between any and all of the physical domains within the spectrum of the model coverage.

Evaluation of a model configuration can be done vs. requirements imposed on a system design. Requirements are expressed in terms of Metrics that can be computed on the system models Examples of metrics include ***Speed, Maximum Towing Force, Acceleration time from 0 to 60 MPH***, etc. Requirements are tests on tests on these metrics, e.g.***" The vehicle must accelerate from 0 to 60 MPH in less than 8 Seconds"*** . Typically, the conditions and scenario will be specified for a requirement CyPhy support, e.g. Level ground, Pavement, and the scenario of Driver Throttle at 100%.

System performance evaluation is specified via a ***Test Bench***. A test bench is an executable specification of a requirement analysis. The parts of a Test Bench are:

- Test Drivers, reproducing the stimulus to the system
- Wraparound environment, providing the interfaces at the periphery of the system (e.g. the ground interface with the tires, the external air, ...)
- Metrics evaluation, taking measurements of the system properties and converting into a value of interest. The metrics are also tied to requirements, which can convert the metric to a design "score". And,
- The system under test – either a point design or a design space. In the case of a design space, the test bench can be applied over the entire set of feasible designs.

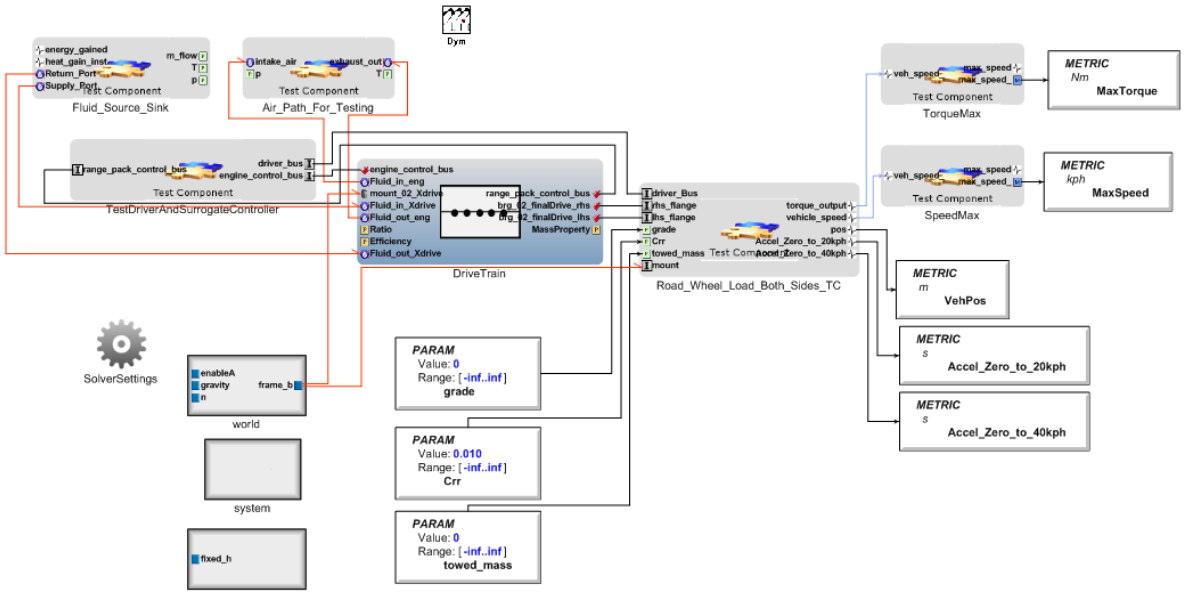


FIGURE 7: EXAMPLE TEST BENCH

The test benches are tied to specific workflows. Currently, CyPhy/OpenMETA implements test benches for:

- Dynamics, using a lumped parameter model executed in the Modelica language. Dynamics cover mechanical, electrical, hydraulic, and thermal domains.
- Structural, using 3D CAD assemblies to evaluate the physical compatibility of parts, locate potential interference, and compute physical properties such as Center of Gravity, Bounding Box, and assembled location of specific points on the system.
- Finite Element, using Finite element techniques to compute stress/strain, thermal propagation, computational fluid dynamics, etc.
- Mobility, using the NATO Reference Mobility Model to predict vehicle mobility based on aggregate system properties,
- Cyber, co-simulating dynamics with a time-based software/processor/network simulation.
- Manufacturability, creating the 3D CAD file, a set of properties of each manufactured join between parts, and an electronic Bill of Materials. From this design package, iFAB can predict a cost and schedule to manufacture the system.
- Complexity, evaluating the graph-energy complexity of the system based on its component complexity and structure of its connections. The complexity metric will correlate with system cost and schedule.

Test benches also have a set of limits associated with part minimum/maximum parameters, (such as maximum torques on a drive shaft), design limits associated with an assembly or the use of a part in a system (such as minimum allowed battery charge). The limits are automatically evaluated with each evaluation of a test bench. If limits are exceeded, a test bench result can be ignored or otherwise modified or treated with less trust.

3.2.4 CYPHY LANGUAGE FORMAL DOCUMENTATION

Included with the package as zipped html files

3.2.5 CYPHY LANGUAGE REFERENCE DOCUMENTATION

Included with the package as zipped html files

3.3 AVM COMPONENT AND DESIGN INTERCHANGE FORMAT (SPEC DOCUMENTATION)

See Appendix

3.4 QUALITATIVE REASONING AND ENVISIONMENT (SUBCONTRACTOR FINAL REPORT)

See Appendix.

3.5 RELATIONAL ABSTRACTION AND SAFETY PROPERTY VERIFICATION WITH HYBRIDSAL (SUBCONTRACTOR FINAL REPORT)

See Appendix.

4 PUBLICATIONS

"A Multi-Modeling Language Suite for Cyber Physical Systems," Neema S., Bapty T., Karsai G., Sztipanovits J., Corman D., Herm T., Stuart D., Mavris D., Proceedings of the 4th International Workshop on Multi-Paradigm Modeling

"Towards Automated Evaluation of Vehicle Dynamics in System-Level Design", Lattman Z., et al, Proceedings of the ASME 2012 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference, IDETC/CIE 2012, August 2012, Chicago, IL, USA

"Towards Automated Exploration and Assembly of Vehicle Design Models", Wrenn R., et al, Proceedings of the ASME 2012 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference, IDETC/CIE 2012, August 2012, Chicago, IL, USA

"Foundation for Model Integration: Semantic Backplane," Simko G., et al, Proceedings of the ASME 2012 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference, IDETC/CIE 2012, August 2012, Chicago, IL, USA

5 APPENDIX 1: A MULTI-MODELING LANGUAGE SUITE FOR CYBER PHYSICAL SYSTEMS

See MPM10_v3.pdf as part of a provided zip file.

6 APPENDIX 2: TOWARDS AUTOMATED EVALUATION OF VEHICLE DYNAMICS IN SYSTEM-LEVEL DESIGN

See dynamics.pdf as part of a provided zip file.

7 APPENDIX 3: TOWARDS AUTOMATED EXPLORATION AND ASSEMBLY OF VEHICLE DESIGN MODELS

See cad_dse.pdf as part of a provided zip file.

8 APPENDIX 4: FOUNDATION FOR MODEL INTEGRATION: SEMANTIC BACKPLANE

See DETC2012-70534.pdf as part of a provided zip file.

AVM Component Model Specification Documentation Version 1.1

Adam Nagel
Institute for Software Integrated Systems (ISIS)
Vanderbilt University
adam@isis.vanderbilt.edu

Developed for the DARPA Adaptive Vehicle Make (AVM) Program

July 20, 2012

Contents

1 Overview

In the context of the AVM program, an **AVM Component** is a well-defined, self-contained, real-world entity that can be used in a design. The AVM Component Model captures metadata and data about the entity.

From a system designer's perspective, it is a "black box," below which the designer does not need composition details. It contains a number of *Features*, which can include numerical properties, links to detailed CAD and behavior models, and fabrication instructions. It also contains a number of *Associations* to relate these instantiated features or sub-parts of these features.

AVM Components are distributed and managed via the VehicleForge Component Exchange.

2 Implementation

An AVM Component Model serves several purposes:

- Serve as a manifest for the Component package
- Define the interfaces and properties of the component
- Serve as a wrapper for integrating various domain models of the component and using them in composing system models

2.1 File Format

AVM Component models are stored in JavaScript Object Notation (JSON). A JSONSchema enforces the syntax of the spec.

2.1.1 Schema

AVM_Component_Schema.json enforces the syntax of a JSON Component Model by checking the root node, Features, Associations, and File objects for the fields required by each.

The schema does not check the completeness or well-formedness of elements deried from the AVM Core Type Libraries, beyond verifying that they meet the requirements of Features.

2.2 Component Package

A full AVM Component "package" may include:

- AVM Component definition in JSON, conforming to this spec *(required)*
- Files required for using the component in analysis and simulation, such as:

  CAD files

  Modelica files

  Simulink files
- Additional materials such as:

  photos

  diagrams

  documentation

2.3 Software Library

The AVM Component model is also supported by a software library that can parse, validate, and export AVM Component models. This library also exposes function calls useful for reading and manipulating an AVM Component model. The library is implemented in C# and can be used within Microsoft's .NET framework.

A Python implementation is planned.

The provided software library also includes classes for handling types from the AVM Core Type Libraries, which are documented separately.

2.4 Features and Types

An AVM Component model consists of *Features*, which are instances of *Types*, which are taken from a set of core vocabularies defined by the program. One such vocabulary is the **META Core Type Library**.

The core types in each vocabulary can be extended to produce more strongly-typed **Types**, such as using the **AVM.META.NamedValue** type to define a specific concept of **Weight**. Using this strong type, the many components using **Weight** can be understood to be using the same concept, beyond matching name strings. These core and extended type libraries are hosted and edited on **VehicleForge**.

2.4.1 Types

All *Types* in core and extended AVM vocabularies must inherit from **AVM.Base**, depicted in the ClassDiagram.

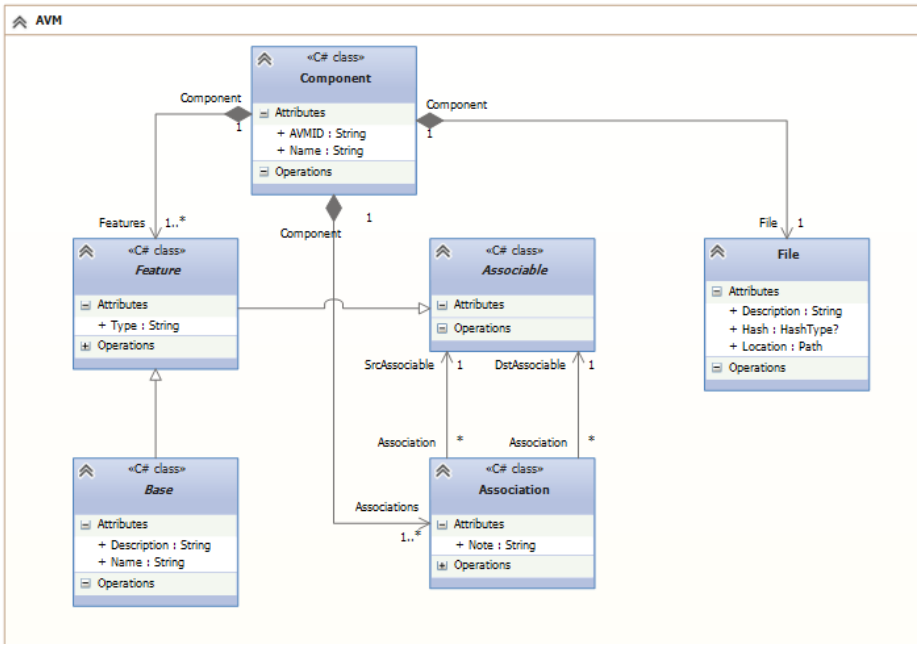
2.4.2 Non-AVM Types

An AVM Component model may also include, as features, types not specified in the core or extended type libraries. However, these features may not be read by the Software Library, and won't necessarily have a common meaning across the tools of the AVM program.

3 Classes

This section includes descriptions of the classes used in the AVM Component spec, along with their attributes. Custom data types and enumerations are documented in Section 4: Type Definitions.

3.1 ClassDiagram



3.2 Component

Component is the root element of an AVM Component definition. An Component contains any number of **Features** and **Associations**.

Attribute	Type	Required	Description
Description	String	no	A description of the component.
AVMID	AVMID	no	An ID unique across all components used in the AVM program.
Name	String	yes	The name of the component.

3.2.1 Feature

A *Feature* is a piece of data associated with a component. All core and extended types in the AVM program vocabularies derive from this class. Inherits from *Associable*.

Attribute	Type	Required	Description
Type	TypeRef	yes	The type of the feature.

3.2.2 Associable

An abstract basesclass for elements which can be the targets of *Associations*. All *Features* are Associable, and sub-parts of *Features* may also be associable. Associable sub-parts of core and extended types derive from this class.

3.2.3 Association

An **Association** defines a directional relation between two associable objects. The semantics of this association are determined by the types of the features involved. These semantics are provided in the core type libraries.

Attribute	Type	Required	Description
Note	String	no	A note describing the association.

3.2.4 Base

An abstract baseclass from which all core types derive, such as those in the AVM Core Type Libraries.

Attribute	Type	Description
Description	String	A description of the type.
Name	String	The name of the type.

3.2.5 File

Describes a file that is distributed as part of the Component package. This supports the Component Model's role as a Manifest for the full Component package.

Attribute	Type	Description
Description	String	A description of the file.
Hash	HashType	A hash of the file (optional).
Location	Path	The path of the file.

4 Type Definitions

Custom data types and enumerations used by the spec are defined in this section.

4.1 TypeRef

Types are identified using dot notation incorporating the namespace of the type. For example, **NamedValue** in the META Core Type Library is indicated with **AVM.META.NamedValue**. These locators will correspond with those used by the VehicleForge query interface to the extended type library, which has not yet been finalized.

4.2 AVMID

The format for AVMID has not yet been finalized. These IDs are defined, assigned, and tracked by the VehicleForge Component Exchange.

4.2.1 Path

The relative path to the resource, from the root node of the component package.

4.2.2 HashType

A hash of the resource. The required formatting is "[hash method]:[hash]".

5 Changelog

5.1 Version 1.1

The changes in this version supported the construction of a parsing library for AVM Component models.

Changes:

- First release of the component model parsing library, implemented in C# within the .NET framework
- Implementation details removed from Class Diagram
- Class Diagram implemented in Visual Studio Class Designer, to enable automatic code generation

- *Associable* baseclass added for Association targets that are not Features, such as sub-parts of Features
- Made explicit the *Base* baseclass for Types
- Added *File* concept
- Added *Description* to Component class

METAX R&D Final Report

PARC, Inc

September 25, 2012

1 Project Overview

The METAX project is developing a toolchain that will enable a 5x speed up in the development time for complex cyber-physical systems (CPS). PARC's role as a sub-contractor to Vanderbilt University (VU) involved integrating our Qualitative Reasoning Module (QRM) with VU's CyPhy-based toolchain. Given that the Ricardo's C2M2L models are written in Modelica [3], and our previous experience with the language, we focused our integration efforts around current open source Modelica tools (e.g., www.openmodelica.org) and models. The addition of Qualitative Reasoning (QR)[1] to the design process is a promising approach as it underlies many design tasks and enables reasoning at multiple levels of abstraction. Qualitative reasoning supports designers by determining if a desired function is realizable, identifying dangerous unexpected interactions, and guiding detailed design and redesign. While current behavioral analysis tools require fully specified designs, qualitative reasoning provides feedback to designers on underspecified models.

The remainder of this report describes our approach to applying QR to design. We begin by describing our approach for performing QR on Modelica models. Then, we describe applications of QR to the design process with examples. And we close with a report on the current status of the project.

2 Performing Qualitative Reasoning with Modelica Models

With Modelica being the common dynamic simulation modeling langauge of AVM, it was necessary to translate Modelica models into a form enabling qualitative reasoning. The standard approach for modeling in QRM is to define a system as a set of connected components where the behavior of each component is defined by a set of equations. Given that Modelica is an object-oriented equation-based modeling language, our initial approach was to perform a syntactic (text-to-text) translation of individual component models. This approach was reconsidered for three reasons: (1) Modelica does not have a standard representation for behavioral modes, (2) Modelica includes a model construction language performing computation during model instantiation, and (3) open-source Modelica compilers perform a number of optimizations when constructing the model.

Therefore, we focused our efforts on translating the instantiated Modelica model's Differential Algebraic Equations (DAEs) into qualitative constraints for use in QRM. Before describing this process, we provide an overview of qualitative simulation, the underlying inference algorithm of QRM.

2.1 Qualitative Simulation

Qualitative simulation [2][6] is the process of projecting forward, from an initial situation and a model, all possible qualitative states that may occur, an *envisionment*. Qualitative representations of continuous quantities (e.g., the angular velocity of a gear) are central to this process. In our familiar Newton-Leibnitz calculus, we use variables to represent quantities that can take any value from the real number line, and vary with time. Variables can have arbitrarily many higher-order derivatives. Likewise, in qualitative reasoning, these variables and their derivatives take on values – except that the values are qualitative. Each variable (or derivative) has a quantity space consisting of an ordered set of *landmark values* representing important points for understanding the behavior of the model (e.g., the turn-on voltage for a diode). A qualitative value is either a landmark or the open interval denoted by two adjacent landmarks. For a door, there are two landmark values: Closed and Open. The door's position can be at one of these two landmarks, or between the (Closed, Open). The qualitative value also has a direction (a qualitative derivative) of increasing, decreasing or steady. The most common quantity space uses just the sign of the real quantity. We represent the interval x<0 as Q-, x=0 as Q0, and x>0 as Q+).

A qualitative state is an assignment of qualitative values to variables in the model. We represent equations as qualitative constraints. Consider the equation governing a resistor, V=I*R, where voltage, V, and current, I, are quantities and R is a fixed parameter with a positive value. The resulting multiplication constraint ensures that the qualitative product of I and R is V. Because R is a positive constant value, if I is a negative value, then V must also be a negative value. Furthermore, their derivatives must also match. Figure 1 defines qualitative addition and multiplication for sign values.

Figure 1: Qualitative Arithmetic Tables

+	Q-	Q0	Q+
Q-	Q-	Q-	Q?
Q0	Q-	Q0	Q+
Q+	Q?	Q+	Q+

*	Q-	Q0	Q+
Q-	Q+	Q0	Q-
Q0	Q0	Q0	Q0
Q+	Q-	Q0	Q+

Non-linear relationships are captured in a number of ways. For polynomials, simple power constraints are used. For other non-linear functions, such as exponentials, we use the monotonic function M+ constraint [6].

One of the most significant consequents of the coarseness of qualitative values is that variables may be qualitatively constant for long periods of time (perhaps infinite). Hence, qualitative simulation need only consider the instants of time at which there is a possible change in qualitative value,. The passage of time is represented as an alternating

sequence of instants and intervals. A qualitative state can either describe an instant or an interval. Qualitative simulation determines all trajectories through the qualitative state space from an initial state. Given a state, qualitative simulation computes possible successors for each variable's qualitative value and uses constraints to determine how they may be combined to form a next, state or states, if any. The rules for generating successor values and directions are based on the mean value theorem from calculus [4]. Consider a position quantity that was between the open and closed landmarks and moving toward closed. There are four possible successors for this quantity. Its value may remain in the interval or reach the closed landmark and it may continue increasing or become steady (its derivative stays positive or becomes Q0).

From basic calculus if a variable is non-zero at an instant, it will remain at that qualitative value in the following interval. If the variable is 0, it will have the qualitative value of its derivative over the following interval. There is one ambiguous case: if a variable and its immediate derivative are both 0, the qualitative value on the following interval is ambiguous (but the variable and its derivative must be qualitative equal during the interval). Consider $x=t^2$ when t=0. The qualitative values x and dx/dt are both Q0, but x=Q+ on the following interval.

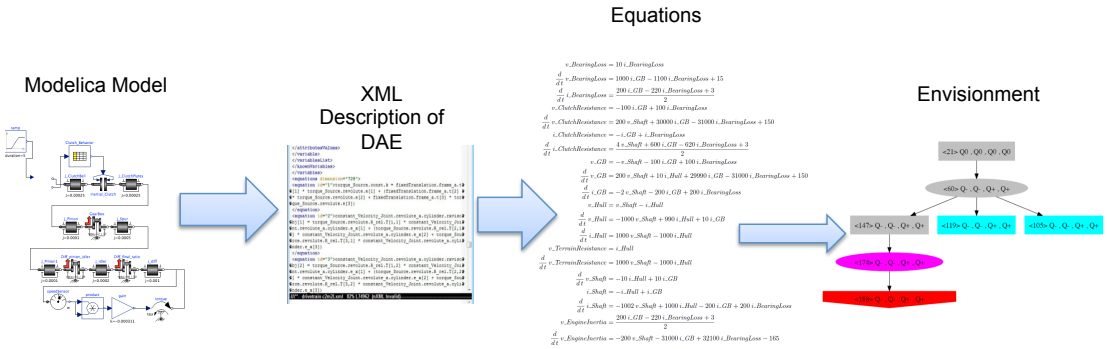
Cyber-physical systems include dynamics that are discrete as well as continuous (e.g., an input signal to open the door, the changing of gears in a drive train, a diode switching from off to on). Qualitative Reasoning typically models discrete changes by modes or operating regions, but as described in Section 2.3, we augmented QRM to align with Modelica's simulation with discrete changes occurring when the conditions of conditional constraints change.

2.2 Translating Modelica Models

Figure 2 provides an overview of our approach for performing qualitative simulation of Modelica models. In the first step, we use OpenModelica to produce a flattened DAE model encoded in an XML file. This DAE includes the results of model construction, algebraic simplification, and index reduction as part of the compilation process. Given the set of equations produced by the OM compiler, we are often able to create the set of constraints necessary to perform qualitative simulation. There are still certain Modelica constructs that we cannot handle, but we are gradually reducing the size of this untranslatable set. More discussion of this point in Section 4.1.

In addition to the equations, we also extract additional information about the variables of the system from the XML. The variables are divided into two classes: `ordered`, which vary over time, and `known`, which are constant through the simulation. Modelica also allows designers to set initial values of variables through the use of the `start` and `fixed` keywords. With this information, we instantiate one or more initial qualitative states and produce an envisionment by qualitative simulation constrained by the equations.

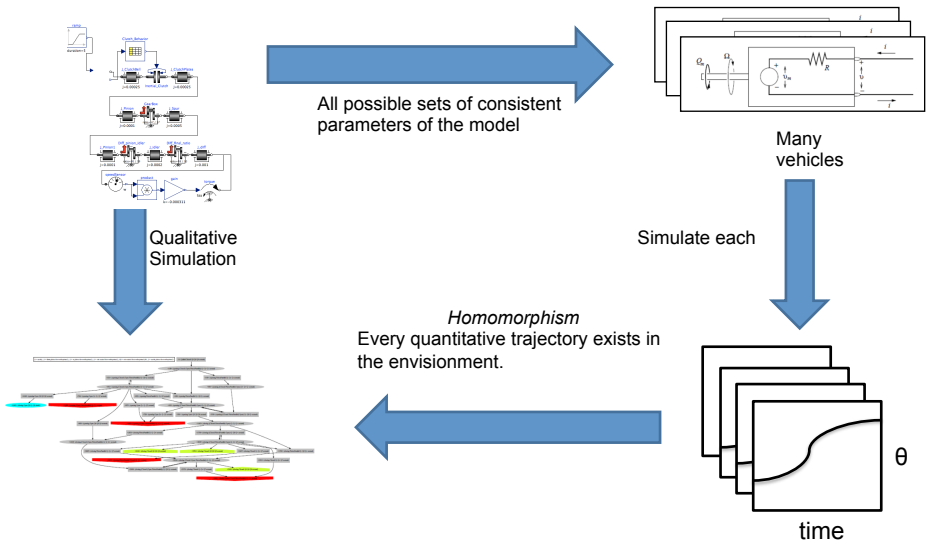
Figure 2: Overview of process to generate envisionments from Modelica models



2.3 Aligning Simulation Semantics with Modelica

In this section, we describe the semantics of qualitative simulation and its relationships to Modelica semantics. Figure 3 illustrates the semantics of our qualitative simulation algorithm. Given a model (upper left) with underspecified parameters (e.g., the peak power output between 500-800 horsepower), qualitative simulation produces an envisionment (lower left) using the equations that define the model. The meaning of this envisionment is that every consistent numeric assignment of the underspecified parameters (upper right), will result in a quantitative simulation (the lower right), and each quantitative simulation will correspond to a trajectory in the envisionment.

Figure 3: Qualitative simulation semantics



Aligning this definition, with Modelica's simulation semantics is straightforward for continuous integration, but requires some explanation for discrete events. In Modelica,

continuous integration proceeds until an event occurs. Events may trigger other events before continuous integration proceeds again. This differs from the standard qualitative simulation interval, instant, interval semantics [1]. This is evident in how static versus sliding friction are modeled. While typical QR modelers would define two modes and transitions between them, Modelica modelers employ a state machine programmed using a number of discrete variables. Figure 4 contains an illustrative subset of the partial model for `PartialFriction` from the MSL. In order for an object to go from rest to having a positive velocity, two events happen in succession at the same time point. First, the `startForward` boolean becomes true when the torque is enough to overcome the static friction, then the `mode` changes value to `Forward` if the resulting torque is enough to overcome sliding friction. For more details on how Modelica treats events see [11].

Given that Modelica simulation using the `PartialFriction` model above results in 5 different sets of values at the same time point, we created an analogous event system in which the qualitative states at the instant the event sequence begins and ends are represented in the quantitative simulation. This allows us to maintain the semantics in Figure 3. We illustrate this using a simplified version of the Brake model, which does not have an interpolation function, and system shown in Figure 5. The system includes a ramp torque and an brake holding an inertia at rest. In the envisionment, the long sequence of rectangles in the figure represents the sequence of events which begin the inertia spinning. There are five terminal intervals. In four of these terminals, the ramp plateaus before overcoming the static friction (there are four due to different amounts of torque being applied on the brake). In the fifth terminal state, the system begins spinning and then continuously accelerates. The full Modelica model for this system is included as in Appendix A.

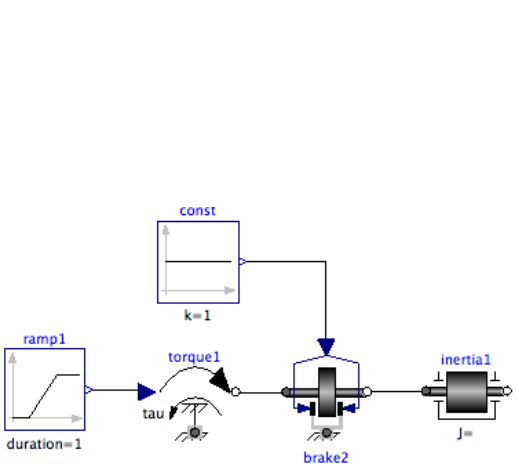
3 Qualitative Reasoning in the Design Process

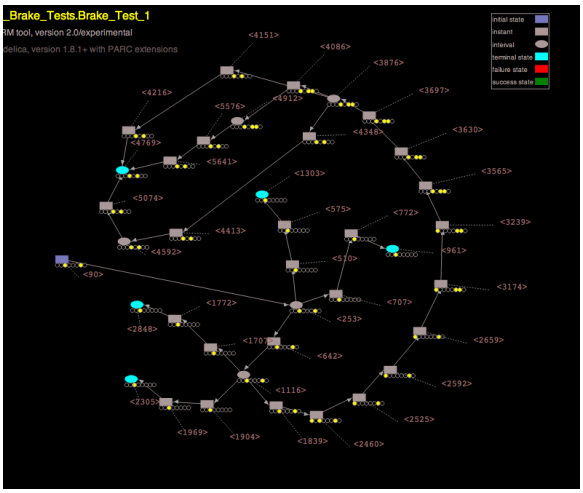
We believe Qualitative Reasoning is the fundamental basis upon which engineers reason about physical systems. Qualitative reasoning plays a key role in every facet of designing a system ranging from early stage design [8] through understanding of simulation results, to planning how designs need to be modified to meet requirements. Unfortunately, none of the commonly used design/analysis tools provide computational QR support for these tasks. Leaving qualitative reasoning entirely to the human engineer risks missing critical inferences. Our vision is to create a design tool chain in which qualitative reasoning supports every segment of the life cycle of a product. Over the course of this project, we have identified and explored four design tasks benefiting from QR: exploring the design space early, finding acceptable parameter relationships, focusing probabilistic verification and identifying changes to bring the model closer to meeting the requirements or unexpected nearby failures via trajectory analysis.

Figure 4: Subset of the PartialFriction model from the Modelica Standard Library illustrating the use of multiple events to trigger a discrete transition

```
partial model PartialFriction
...
  Boolean startForward(start = false, fixed = true);
  Integer mode(final min = Backward, final max = Unknown,
                           start = Unknown, fixed = true);
 equation
  startForward = pre(mode) == Stuck
                      and (sa > tau0_max / unitTorque
                                          or pre(startForward)
                                  and sa > tau0 / unitTorque)
                                         or pre(mode) == Backward
                                                 and w_relfric > w_small
                                  or initial() and w_relfric > 0;
  mode = if free
    then Free
    else if (pre(mode) == Forward
                or pre(mode) == Free
                or startForward)
              and w_relfric > 0 then
                Forward
              else if (pre(mode) == Backward
                           or pre(mode) == Free
                           or startBackward)
                          and w_relfric < 0 then
                           Backward
                else
                    Stuck;
...
end PartialFriction;
```

Figure 5: Brake model and resulting envisionment



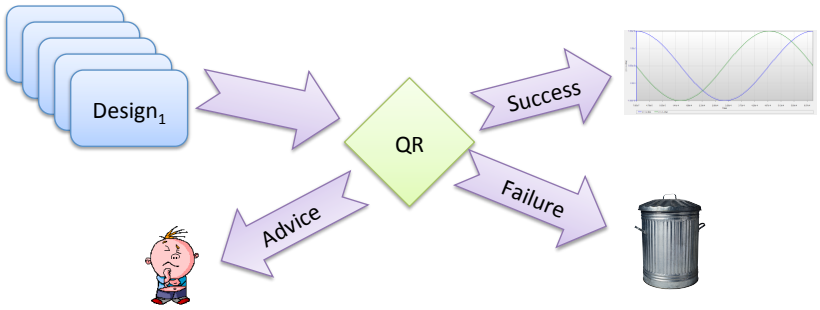


3.1 Early Design Exploration

One aspect of design space exploration is topological design (i.e., determining the configuration of abstract components). Figure 6 illustrates how Qualitative Reasoning assists in this process. Given a set of potential designs created by a designer or a search algorithm (e.g., DESSERT [10], GraphSynth [7]), QRM performs qualitative simulation on each design. After the envisionment graph has been created, QRM provides the following analysis. If none of the trajectories violate requirements, then for all consistent assignments of parameters, the system will satisfy all requirements. That is, the system will not reach a state that violates a safety requirement or transition along an undesirable trajectory. If some trajectories violate requirements and others do not, then the design may satisfy the requirements with appropriate constraints on component parameter values. In either case, detailed design is required to determine the assignment of parameter values that best fits the needs of the designer. If all of the trajectories violate requirements, detailed design is not necessary because no set of parameter values will satisfy the requirement. In one experiment reported at a PI meeting, we showed how a program automatically found the design for a simple circuit that satisfied a specified requirement.

In addition to filtering out bad designs, qualitative reasoning provides feedback to the designer or automated design tools. The envisionment itself is a form of feedback expressing the range of possible behaviors for a design. In the next three sections, we describe other types of feedback that can be automatically extracted from an envisionment.

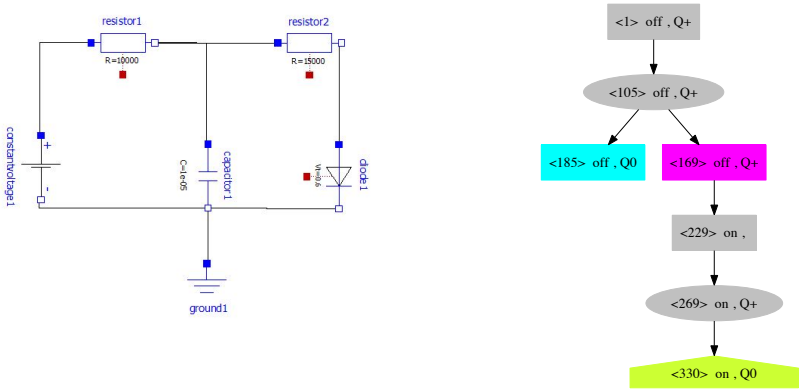
Figure 6: Qualitative reasoning during early design



3.2 Acceptable Parameter Relationships

For many designs, qualitative simulation will result in both trajectories that satisfy the requirements and trajectories that do not. As observed by Iwasaki [5] this ambiguity is useful in design because it can alert designers to potential problems, and it can also be used to construct additional constraints to guide detailed design. In this section we present guards, which are constraints on the model that focus the envisionment toward successful trajectories.

Figure 7: Design and corresponding envisionment which is ambiguous with respect to the requirement that the light bulb turns on a short time after the switch is flipped



One source of ambiguity arises from the fact that each set of landmark variables is defined with respect to a single component. Consider the circuit in Figure 7 with the requirement is that the light bulb (diode) illuminates after the switch is flipped. The envisionment includes one successful and one failed trajectory. In this model, there is a battery voltage landmark (`VBat`) in the battery model and turn on voltage landmark (`OnV`) in the diode model. This ambiguity results from ambiguity in the ordering of these landmarks. In this situation, we are able to automatically extract the parameter relationship necessary for the successful trajectory as follows. First, we propagate the landmarks across constraints. Next, we look for each variable with multiple landmarks

and create a set of models each representing a total ordering of the landmarks. In our diode example, each voltage variable will have two landmarks. Therefore, we generate three models representing the possible orderings the two landmarks:

1. `OnV` > VBat (-∞, `-OnV`, `-VBat`, `Q0`, `VBat`, `OnV`, ∞)
2. `OnV` = VBat (-∞, `-L1`, `Q0`, `L1`, ∞)
3. `OnV` < VBat (-∞, `-VBat`, `-OnV`, `Q0`, `OnV`, `VBat`, ∞)

`L1` is a new landmark that is created that is equal to `OnV` and `VBat`. We create three qualitative models with the same set of constraints, but with different quantity spaces for the voltage variables in the model. We perform qualitative simulation on each of the systems. The envisionment of the first system consists of three states terminating when the capacitor is charged and the diode is off. By simulating the second system, QRM determines that the initial conditions are inconsistent because the only trajectory leads to an endless loop of discrete transitions (i.e., the diode switches between off and on). The third system results in an envisonment of the successful trajectory. Therefore any parameter settings that satisfy the inequality defining the system, `OnV` < `VBat`, will result in a system that meets the requirements. In this manner, we can infer inequalities to guide parameter selection in detailed design.

3.3 Focusing Probabilistic Verification

Probabilistic Certificate of Correctness (PCC) is an analysis of a fully specified design whose parameters are specified by distributions. The standard approach involves sampling the parameter space and performing simulations. Extremely rare events are either ignored, or poorly estimated due to limit samples. Given an envisionment, we can identify which paths lead to these failures and also perform selective re-sampling to better estimate their probabilities. Thereby focusing the quantitative analysis on "black swans", low probability high impact events. This idea was originally used to explore the potential outcomes of military engagements in DARPA's Deep Green program [4].

Figure 8: Focused sampling for PCC calculations

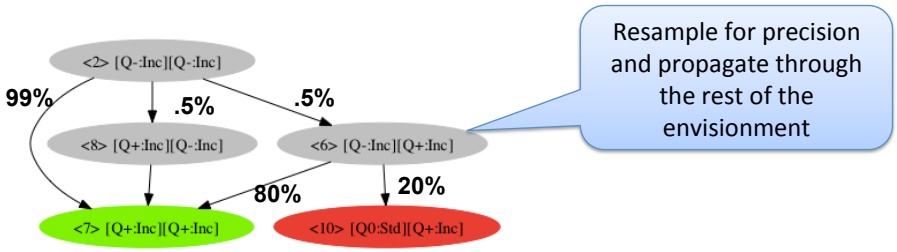


Figure 8 provides an illustrative example of an envisionment graph with transition probabilities between the states. From the initial situation (2), the vast majority of

parameter settings result in a direct transition to a successful state (7). A small percentage goes to an intermediate state (8), which from the envisionment, we know results in a successful outcome. But the small number of samples which transition to 6 provide an interesting opportunity to resample. Consider an monte carlo simulation of 1000 samples. 5 of which transition to state 6, and of these, all but 1 transitions results in a successful transition. The small number of samples from state 6 may not provide an accurate conditional probability, therefore we could resample from the parameters that reach this state to better assess the likelihood of the design resulting in the failure in state 10.

3.4 Trajectory Analysis

When all parameters are known, qualitative reasoning helps the designer to interpret the results of the simulation. *Trajectory analysis* is the process of exploring the design by looking at the trajectory of the quantitative simulation through the envisionment. Recall from Section 2.3 that every possible quantitative model has a corresponding trajectory in the envisionment of the qualitative simulation. Consider a failed trajectory, using comparative analysis [12], qualitative reasoning suggests possible parameters to change in order to change the trajectory to a successful one. Or in the case of a successful simulation, the envisionment could illustrate which requirement failures are "closest" to the trajectory. Providing this knowledge to the design could assist with redesign efforts to improve the robustness of the design.

3.5 Integrating QR with Design Tools

In order to realize the benefits of these techniques, they need to be accessible to designers. Consequently, we have focused substantial effort on integrating our techniques with existing tools. We have integrated with a version of the OMEdit tool (www.openmodelica) which provides a GUI for creating and simulating designs. QRM is invoked on a model by simply clicking the "ParcQR" button on the OMEdit interface. To achieve this, we fixed a number of outstanding bugs in the XML equation dumping capabilities of the OpenModelica compiler. This had the benefit of facilitating the integration with CyPhy through a standalone executable, enabling QRM to be used by the AVM community. Furthermore, the XML equations are also used as an input language to SRI's Hybrid-SAL verification system. By integrating with current design tools, we are able to make continuous improvements to our tools available to designers as quickly as possible.

4 Current Status of QRM

While work is ongoing with respect to the model translation efforts and the creation of interfaces enabling qualitative reasoning to support designers, we report in this section the current status of QRM. We begin by discussing the differences between "good" and "bad" Modelica providing a scope for understanding our progress. Then we discuss to what extent QRM works with existing models in the Modelica Standard Library, the

August release of C2M2L models from Ricardo, and the RC car models from Vanderbilt. Finally, we describe the classes of requirements applicable to our approach and how they are translated into QRM.

4.1 Good Modelica vs. Bad Modelica

Well-formed models support many different kinds of analysis including simulation, verification and fault analysis. Unfortunately, Modelica allows for arbitrary non-declarative algorithm blocks (e.g., interpolation tables, matrix transformations, external code) as part of any model. These constructs appear in many Modelica Standard Library (MSL) and C2M2L models. Therefore, we proposed the following principles of good modeling: (1) as declarative as possible, minimize callouts to external code, (2) compositionality, i.e., no function in structure, and (3) a standard mode representation. Models conforming to these principles will support not only dynamic simulation through a Modelica compiler, but will also support verification and fault analysis. When models require algorithm blocks, analogous models or abstractions will need to be created to use the verification techniques employed by PARC, SIFT, and SRI in the METAX projects. An advantage of declarative representation is that enables diagnosis and exploration of faulty behavior by interrogating the model. Currently, our model translation to QRM works on fully declarative models and QRM requires qualitative representations of math operations. At this point, we have not implemented trigonometry operators, for which there are understood approaches in the QR community [9]. Therefore, fully declarative models without trigonometry should work in the current version of QRM.

Another modeling issues is the need for a standard interface to the results of different Modelica compilers. While Dymola is an industry standard, it is also an expensive commercial product without a well-defined interface for programmatic interaction. OpenModelica and JModelica are two open-source Modelica compilers. While the open-source nature of these projects facilitates integration with other software tools, they are both currently immature with respect to covering the entire Modelica language and the models being produced by Ricardo in C2M2L. As indicated in Section 2.2, we are using the flattened DAE produced by OpenModelica.

4.2 Application of QRM to Existing Models

While, the full integration with VU tools has been tested, and demonstrated at the September AVM PI meeting, using the Battery model and test bench shown in Appendix B, in this section, we report on the current status of model translation and QRM on three sets of existing models: the Modelica Standard Library, August C2M2L Modelica models from Ricardo, and the RC Car models released by VU in August. As our approach relies on OpenModelica, we report the number of models for which OpenModelica produces a simulation for context. Also as indicated in Section 3, a total envisionment is not always necessary or desired, therefore we outline the number of models which produces depth-bounded envisionment as well as a total envisionment within a reasonable timeout (X seconds).

4.3 What Requirements can be Addressed?

Our analysis of the FANG requirements document indicates that QRM applicable to Vehicle Functionality Requirements, or behavioral requirements. These requirements describe a state of the cyber-physical system and a context of use. These requirements include safety requirements specifying a state that should be avoided, e.g., requirement 4.7.5 "On a level, hard surface without use of service brakes, the vehicle shall operate at a speed no greater than: 4.0 kph," and performance requirements illustrating a state that must be achieved, e.g., requirement 4.8.1 "The vehicle shall steer at least 30 degrees while coasting with the engine inoperative at speeds of $\geq$ 40 khp." Our work with VU has discussed an appropriate interface for entering these requirements into CyPhy and analogous Modelica templates.

Figure 9: Example FANG requirement in Modelica

```
...
parameter Real threshold = 30;
type Requirement = enumeration(violated, success, unknown);
Requirement req1 (start = Requirement.unknown);
if req1 == Requirement.success or threshold < vehicle1.orientation then
  req1 = Requirement.success;
elseif final()
  req1 = Requirement.violated;
else
  req1 = Requirement.unknown;
end if;
...
```

Consider the performance requirement requirement concerning the vehicle steering. This is a simple threshold requirement that vehicle must enter a state in which the orientation of vehicle has rotated 30 degrees. If in the use case where the operator steers the vehicle right and the orientation of the vehicle begins at 0 degrees, then the designer would enter the requirement that the vehicle reaches a state in which its heading is greater than 30 degrees. Figure 9 illustrates the filled in Modelica template for this requirement.

The requirement to the CyPhy test bench and passed to the model translator with the associated Modelica model. Requirement violations terminate trajectories in qualitative simulation and color coded in our visualization for the envisionment. This allows the engineer to quickly access which requirements may be violated by a designer to understand the behavior which leads to it.

5 Discussion

PARC's performance on the METAX project has focused on the integration of the Qualitative Reasoning Module (QRM) with the Vanderbilt tool-chain. As described in this report, qualitative reasoning can be used to support engineers in a variety of design tasks including exploring the design space, determining acceptable parameter relationships, focusing numeric analysis, and providing feedback based on the results of quantitative simulations. To place these capabilities in the hands of designers, we integrated our tool with OpenModelica's compiler, which enabled integration with an open-source design tool (OMEdit) as well as Vanderbilt's CyPhy tool-chain. We have demonstrated this through a number of examples at PI meetings over the course of this program. As the program progressed, we have been given models from system engineers at Vanderbilt and Ricardo. Our analysis of these models led us to define "Good Modelica" which we have outlined here as a set of modeling principles to enable model use in a range of analysis tasks including simulation, verification, and fault analysis. Our analysis of models that do not currently work in QRM results in two groups of models: (1) imperative models and (2) models including unhandled operators (e.g., trigonometry). While imperative models pose a problem for all verification approaches, Ricardo agrees that imperative models are frequently the result of bad modeling and is working to make their models as declarative as possible. The second group of models is addressed by implementing model translation and qualitative versions of the offending operators. Consequently, we expect the number of models in each group to reduce overtime. By integrating QRM with the results of compiled Modelica models, we are able to apply our tool to a large number of existing models and provide continuous improvements ensuring that the remaining semantically grounded.

References

[1] de Kleer, J., and Williams, B. C., Eds. *Qualitative Reasoning about Physical Systems II.* Elsevier, Amsterdam, Oct. 1991. *Artificial Intelligence 51.*

[2] Forbus, K. D. Qualitative process theory. *Artificial Intelligence 24*, 1 (1984), 85–168. Also in: Bobrow, D. (ed.) *Qualitative Reasoning about Physical Systems* (North-Holland, Amsterdam, 1984 / MIT Press, Cambridge, Mass., 1985).

[3] Fritzson, P. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1.* Wiley-IEEE Press, Piscataway, NJ, 2004.

[4] Hinrichs, T. R., Forbus, K. D., de Kleer, J., Yoon, S., Jones, E., Hyland, R., and Wilson, J. Hybrid qualitative simulation of military operations. In *IAAI* (2011), D. G. Shapiro and M. P. J. Fromherz, Eds., AAAI.

[5] Iwasaki, Y. Qualitative reasoning and the science of design. *IEEE Expert 12*, 3 (may/jun 1997), 2 –4.

[6] Kuipers, B. *Qualitative reasoning: modeling and simulation with incomplete knowledge*. MIT Press, Cambridge, MA, USA, 1994.

[7] Kurtoglu, T., and Campbell, M. Automated synthesis of electromechanical design configurations from empirical analysis of function to form mapping. *Journal of Engineering Design 20*, 1 (2009), 83–104.

[8] Kurtoglu, T., and Tumer, I. Y. Ffip: A framework for early assessment of functional failures in complex systems. In *International Conference on Engineering Design—ICED'07* (2007).

[9] Liu, J. A method of spatial reasoning based on qualitative trigonometry. *Artificial Intelligence 98*, 1-2 (1998), 137 – 168.

[10] Neema, S., Sztipanovits, J., Karsai, G., and Butts, K. Constraint-based design-space exploration and model synthesis. In *EMSOFT* (2003), R. Alur and I. Lee, Eds., vol. 2855 of *Lecture Notes in Computer Science*, Springer, pp. 290–305.

[11] Otter, M., Elmqvist, H., and Mattsson, S. E. Hybrid modeling in modelica based on the synchronous data flow principle. In *ACCEPTED FOR THE 1999 IEEE SYMPOSIUM ON COMPUTER AIDED CONTROL SYSTEM DESIGN, CACSD 1999* (1999), Aug, pp. 22–27.

[12] Weld, D. S. Theories of comparative analysis. Technical Report 1035, MIT Artificial Intelligence Lab, May 1988.

6 Appendixes

6.1 Appendix A

```
package Simple_Brake_Tests
  model Brake_Test_1
    Modelica.Mechanics.Rotational.Sources.Torque torque1;
    Modelica.Blocks.Sources.Ramp ramp1(duration = 1);
    Modelica.Blocks.Sources.Constant const(k = 1);
    Modelica.Mechanics.Rotational.Components.Inertia inertia1;
    Modelica.Mechanics.Rotational.Components.Brake brake2;
  equation
    connect(const.y,brake2.f_normalized);
    connect(brake2.flange_b,inertia1.flange_a);
    connect(torque1.flange,brake2.flange_a);
    connect(ramp1.y,torque1.tau);
  end Brake_Test_1;

  model BrakePARC
```

```
  extends Modelica.Mechanics.Rotational.Interfaces.
     PartialElementaryTwoFlangesAndSupport2;
  parameter Real mue0(final min = 0);
  parameter Real peak(final min = 1) = 1 "peak*mue_pos[1,2] =
      maximum value of mue for w_rel==0";
  parameter Real cgeo(final min = 0) = 1 "Geometry constant
     containing friction distribution assumption";
  parameter Real fn_max(final min = 0, start = 1) "Maximum
     normal force";
  //    extends Modelica.Mechanics.Rotational.Interfaces.
     PartialFriction;
  extends PartialFrictionPARC;
  Real phi "Angle between shaft flanges (flange_a, flange_b)
     and support";
  Real tau "Brake friction torqu";
  Real w "Absolute angular velocity of flange_a and flange_b
     ";
  Real a "Absolute angular acceleration of flange_a and
     flange_b";
  Real fn "Normal force (=fn_max*f_normalized)";
  // Constant auxiliary variable
  Modelica.Blocks.Interfaces.RealInput f_normalized "
     Normalized force signal 0..1 (normal force = fn_max*
     f_normalized; brake is active if > 0)";
equation
  phi = flange_a.phi - phi_support;
  flange_b.phi = flange_a.phi;
  w = der(phi);
  a = der(w);
  w_relfric = w;
  a_relfric = a;
  flange_a.tau + flange_b.tau - tau = 0;
  fn = fn_max * f_normalized;
  tau0 = mue0 * cgeo * fn;
  tau0_max = peak * tau0;
  free = fn <= 0;
  tau = if locked then sa * unitTorque else if free then 0
     else cgeo * fn * (if startForward then mue0 else if
     startBackward then -mue0 else if pre(mode) == ModeType.
     Forward then mue0 else -mue0);
end BrakePARC;
```

```
partial model PartialFrictionPARC "Partial model of Coulomb
   friction elements - PARC using enumerated types"
  parameter Real w_small = 10000000000 "Relative angular
     velocity near to zero if jumps due to a reinit(..) of
     the velocity can occur (set to low value only if such
     impulses can occur)";
  // Equations to define the following variables have to be
     defined in subclasses
  Real w_relfric "Relative angular velocity between
     frictional surfaces";
  Real a_relfric "Relative angular acceleration between
     frictional surfaces";
  //SI.Torque tau "Friction torque (positive, if directed in
     opposite direction of w_rel)";
  Real tau0 "Friction torque for w=0 and forward sliding";
  Real tau0_max "Maximum friction torque for w=0 and locked";
  Boolean free "true, if frictional element is not active";
  // Equations to define the following variables are given in
      this class
  Real sa(final unit = "1") "Path parameter of friction
     characteristic tau = f(a_relfric)";
  Boolean startForward(start = false, fixed = true) "true, if
      w_rel=0 and start of forward sliding";
  Boolean startBackward(start = false, fixed = true) "true,
     if w_rel=0 and start of backward sliding";
  Boolean locked(start = false) "true, if w_rel=0 and not
     sliding";
  type ModeType = enumeration(Unknown, Free, Forward, Stuck ,
      Backward );
  ModeType mode(start = ModeType.Unknown, fixed = true);
protected
  constant Real unitAngularAcceleration = 1 annotation(
     HideResult = true);
  constant Real unitTorque = 1 annotation(HideResult = true);
equation
  startForward = pre(mode) == ModeType.Stuck and (sa >
     tau0_max / unitTorque or pre(startForward) and sa > tau0
      / unitTorque) or pre(mode) == ModeType.Backward and
     w_relfric > w_small or initial() and w_relfric > 0;
  startBackward = pre(mode) == ModeType.Stuck and (sa < -
     tau0_max / unitTorque or pre(startBackward) and sa < -
     tau0 / unitTorque) or pre(mode) == ModeType.Forward and
     w_relfric < -w_small or initial() and w_relfric < 0;
```

```
    locked = not free and not (pre(mode) == ModeType.Forward or
        startForward or pre(mode) == ModeType.Backward or
       startBackward);
    a_relfric / unitAngularAcceleration = if locked then 0 else
        if free then sa else if startForward then sa - tau0_max
        / unitTorque else if startBackward then sa + tau0_max /
        unitTorque else if pre(mode) == ModeType.Forward then
       sa - tau0_max / unitTorque else sa + tau0_max /
       unitTorque;
    mode = if free then ModeType.Free else if (pre(mode) ==
       ModeType.Forward or pre(mode) == ModeType.Free or
       startForward) and w_relfric > 0 then ModeType.Forward
       else if (pre(mode) == ModeType.Backward or pre(mode) ==
       ModeType.Free or startBackward) and w_relfric < 0 then
       ModeType.Backward else ModeType.Stuck;
    annotation(Documentation(info = "<html>
                            <p>
                            Basic model for Coulomb friction
                               that models the stuck phase in
                               a reliable way.
                            </p>
                            </html>"));
  end PartialFrictionPARC;
end Simple_Brake_Tests;
```

6.2 Appendix B

```
within C2M2L_RC_Car.Powertrain.Battery;
model Battery
  extends Modelica.Blocks.Interfaces.BlockIcon;
  parameter Real InnerResistor = 0.001;
  parameter Real InitialVoltage( start = 6.5, min = 5.5);
  parameter Real LifeTime = 5400;
  parameter Real minV = 5.5;
  parameter Real maxV = 6.5;
  parameter Real Cap = -(LifeTime/8.55564)*log(1-(minV/maxV));
  Real battery_life
    "Percentage of charge which the battery has left. Battery
       is considered dead when this value drops to zero.
       Determined by looking at the battery voltage; battery is
        dead once the voltage hits 5.5";
  Modelica.Electrical.Analog.Basic.Capacitor capacitor(v(start
    = InitialVoltage), C=Cap);
```

```
  Modelica.Electrical.Analog.Basic.Resistor resistor(R=1000000)
    ;
  Modelica.Electrical.Analog.Basic.Resistor resistor1(R=
    InnerResistor);
  Modelica.Electrical.Analog.Interfaces.Pin positive;
  Modelica.Electrical.Analog.Interfaces.NegativePin negative;
  Modelica.Blocks.Interfaces.RealOutput batterylife;
equation
  battery_life = (positive.v - 5.5)*100;
  battery_life = batterylife;
  connect(capacitor.p, resistor1.p);
  connect(resistor.p, resistor1.p);
  connect(resistor1.n, positive);
  connect(capacitor.n, negative);
  connect(resistor.n, negative);
end Battery;

within C2M2L_RC_Car.Powertrain.Battery;
model RCBatteryTest
  import RCBattery;
  import C2M2L_RC_Car;
  Modelica.Electrical.Analog.Basic.Ground ground;
  Modelica.Electrical.Analog.Basic.Resistor resistor(R=1000);
  C2M2L_RC_Car.Powertrain.Battery.Battery    rCBattery1(
    InnerResistor=2, InitialVoltage=6.5);
equation
  connect(resistor.n, ground.p);
  connect(rCBattery1.positive, resistor.p);
  connect(rCBattery1.negative, ground.p);
end RCBatteryTest;
```

HybridSAL Relational Abstraction

Ashish Tiwari

SRI International, Menlo Park, CA. ashish.tiwari@sri.com

Abstract. We describe the relational abstraction component of the Meta Verification tool chain. Relational abstraction is a technique for verifying safety requirements. It does so by abstracting the given system, which could have a combination of continuous and discrete dynamics, into a purely discrete system. The abstract discrete system is then analyzed using automated verification techniques, such as model checking. The relational abstraction component consists of:

- a Modelica to HybridSal translator,
- HybridSal relational abstracter, and
- Sal infinite bounded model checker and k-induction prover.

This report describes these components, its strength and limitations, and its use for safety verification.

1 Introduction

The HybridSal relational abstraction tool is part of the Meta verification tool chain. The relational abstraction component of the tool chain can verify safety properties of a system model. Safety properties state that the system always remains inside some safety bounds; or, in other words, the system never reaches an "unsafe" state.

The systems being analyzed here are complex in at least two different ways. First, they are high dimensional; that is, their state is described by a large number of variables. Second, the dynamics of the system contain a combination of both discrete and continuous behaviors. As a result, automated verification of such systems is challenging. Relational abstraction is an approach for verification that addresses this challenge by first constructing an abstraction of the system, and then verifying safety properties on the abstract system. Relational abstractions are often more precise than a purely qualitative abstraction of the system, but they are still abstract enough to enable tractable analysis using state exploration verification tools.

The HybridSal relational abstraction tool has its own modeling language for hybrid dynamical systems, called the HybridSal language. It extends the `SAL` language, which was designed for modeling discrete state transition systems, with features that enable modeling of continuous dynamical systems. The HybridSal language has support for modeling a system as a composition of subsystems, and supports nondeterminism and symbolic parameter values.

For purposes of integration with the larger Meta verification tool box, HybridSal also accepts Modelica models. Specifically, the tool currently accepts as

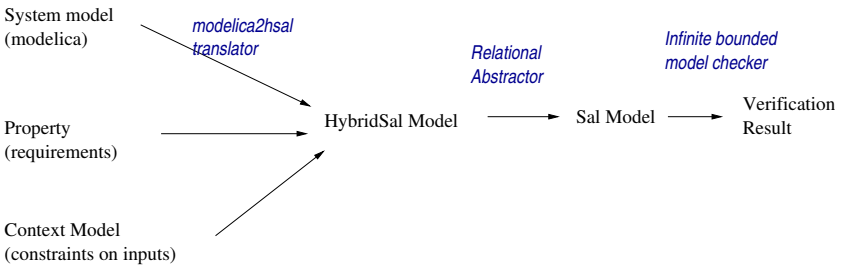


Fig. 1. Relational Abstraction Tool Flow.

input the XML dump of the flattened Modelica model generated by the OpenModelica compiler. This frontend is implemented as a separable component of HybridSal, called `modelica2hsal`. Due to time constraints and the richness of the Modelica language, the `modelica2hsal` tool can convert on a limited subset of Modelica models into HybridSal. The `modelica2hsal` tool, its features, and its limitations are described in Section 2.

The HybridSal relational abstraction algorithm and implemention is described in Section 3. We also describe the limitations of the abstracter in that section.

The abstraction constructed by the HybridSal relational abstraction tool is output as a `SAL` model. We briefly describe the SAL model checking tools that can be used to analyze this abstract system in Section 4.

Figure 1 shows the overall architecture of our verification tool and its three main components. We describe, in detail, these three components in subsequent sections.

2 `modelica2hsal`: Converting Modelica to HybridSal

The `modelica2hsal` tool converts a Modelica model into a HybridSal model. The input to the tool is a file containing the XML dump of the flattened Modelica model generated by the OpenModelica compiler. The output of the tool is a file containing the HybridSal model. The tool also takes a second optional argument, which is a file that contains the context model and the requirement specification. When the optional argument is provided, the `modelica2hsal` tool automatically includes the model of the context and the property in the generated HybridSal file.

The key technical difficulty in converting Modelica models into HybridSal models is that Modelica represents models as a set of *differential algebraic equations* (DAEs), whereas HybridSal represents systems as a collection of *ordinary differential equations* with switching between them (switched ODEs). In the process of converting DAEs into switched ODEs, one has to eliminate the algebraic constraints in the DAEs. There is no algorithmic procedure to achieve this goal.

There are heuristics based on simplifying the DAEs using different simplification rules.

The `modelica2hsal` tool is implemented as follows:

1. Read the XML dump generated by Modelica
2. Output the DAEs in a .dae file
3. Parse the .dae file and create a .daexml file
4. Simplify the .daexml
5. Map the simplified .daexml into HybridSal model

The main step in the translation is the simplification step above. The `modelica2hsal` tool currently implements several different simplifications, including

Constant propagation: If a variable is known to have a fixed constant value, then it is uniformly replaced by that value in the entire model and eliminated. The constant propagation process can be written as the following rewrite rule:

$$x = c, e[x] \rightarrow e[x \mapsto c] \tag{1}$$

where e is any expression and $e[x \mapsto c]$ denotes the result of replacing x by c everywhere in e.

partial evaluation: If subexpressions can be evaluated as a result of constant propagation, then they are evaluated and replaced by the computed value for them. Application of partial evaluation can result in more possibilities for constant propagation, and vice versa. Some sample rewrite rules that describe partial evaluation are mentioned below:

$$c1 + c2 \rightarrow c \qquad \text{where } c \text{ is the sum of } c1 \text{ and } c2 \tag{2}$$

$$0 * e \rightarrow 0 \tag{3}$$

$$0 + e \rightarrow e \tag{4}$$

$$e - 0 \rightarrow e \tag{5}$$

$$\texttt{ite}(\texttt{True}, e1, e2) \rightarrow e1 \tag{6}$$

$$\texttt{False} \wedge e \rightarrow \texttt{False} \tag{7}$$

Term normalization: Expressions in the model are normalized using rewriting rules. For example, $x + 2x + y$ is normalized to $3x + y$, and $x + (-x)$ is normalized to 0. The Modelica expression language has "if-then-else" as a language construct. Sometimes the different operating modes of the system are hidden inside such "if-then-else" expressions. For example, the differential equation

$$\frac{dx}{dt} = \text{if c then e1 else e2}$$

is a way to represent two modes: in one mode (when c is true), $dx/dt = e1$, and in the second mode (when c is false), $dx/dt = e2$. However, nested

"if-then-else" expressions and combination of "if-then-else" expressions with algebraic operators, such as $+, -, *$ make the problem of identifying modes more difficult. We use the following normalization rules for handling these cases:

$$\texttt{ite}(c1, \texttt{ite}(c2, e1, e2), e3) \rightarrow \texttt{ite}(c1 \wedge c2, e1, \texttt{ite}(c1 \wedge \neg c2, e2, e3)) \quad (8)$$

$$\texttt{ite}(c1, e1, e2) + e \rightarrow \texttt{ite}(c1, e1 + e, e2 + e) \quad (9)$$

$$\texttt{ite}(c1, e1, e2) - e \rightarrow \texttt{ite}(c1, e1 - e, e2 - e) \quad (10)$$

$$\texttt{ite}(c1, e1, e2) * e \rightarrow \texttt{ite}(c1, e1 * e, e2 * e) \quad (11)$$

$$\texttt{ite}(c1, e1, e2)/e \rightarrow \texttt{ite}(c1, e1/e, e2/e) \quad (12)$$

Note that application of one rule can trigger application of others. We apply all simplification and rewrite rules until they can be applied no more.

The rewrite rules, Rule (1)–Rule (12), is not intended to be exhaustive list of all rules used by the `modelica2hsal` translator. Its purpose is to just give an idea of the simplification steps performed during the translation.

There are plenty of special rewrite rules to simplify and handle expressions that are intrinsic to Modelica, such as,

1. set and set access:
2. computing cross product and dot product of two vectors
3. computing transpose of a matrix
4. user-defined interpolation function
5. arithmetic expressions over set terms
6. predefined functions, such as, `sqrt`, `abs`, `cos`, `sin`, `Real`, `noEvent`, `der`

Note that the simplification rules may not always be successful in eliminating certain functions; for example, sin can be eliminated (using the simplification rewrite rules) if it is always applied to a statically known constant (for e.g., 0 or $\pi/2$), but it can not be eliminated in general.

An important step during the simplification phase is the step that replaces (non fixed) variables by equivalent expressions. This is a way to eliminate algebraic constraints from the DAEs. We take care not to eliminate *state variables*. If we can generate an expression $x = e$ by simplification, and x is not defined to be a state variable, we replace x by e everywhere in the Modelica model and eliminate x.

There are also additional simplification rules for handling tables, and the interpolation function. Essentially, a table is converted to a large "if-then-else" expression.

2.1 `modelica2hsal`: Translating Simplified DAEs to HybridSal

After the DAEs are simplified using the simplification rules described above, the tool attempts to convert them into HybridSal. The final HybridSal model is a *synchronous composition* of three different base modules:

– a controller
– a plant
– a monitor

The property (requirement) are captured separately in the HybridSal file. The context model is currently integrated into the HybridSal model. Ideally, it should be implemented as a separate synchronously composed module (like the monitor module).

The final conversion step is performed as follows:

1. identify the variables that define the state space of the system
2. identify the modes of the system
3. create the controller module
4. create the plant module
5. create the monitor module

The modes are identified by collecting all predicates that occur in the "if-then-else" expressions in the DAEs.

The controller module. Next, we generate the controller module. The controller module updates all discrete variables. It is obtained from those equations in the set of all DAEs that
(a) do not contain the derivative, `der`, operator
(b) can be written as $x = e$, and x is a discrete state variable
To guarantee the latter condition, `modelica2hsal` also implements a symbolic equation solver.

The controller also makes sure that, in each step, some predicate (from the set of all predicates computed above) always changes.

The feature of HybridSal that makes the translation possible is that HybridSal allows two variants of each state variable in expressions – x **and** x' – x is the previous value and x' is the value in the next time step. Modelica also uses two variants of each state variable: `pre`(x) and x, with the same meaning.

The plant module. The plant module is created from those equations in the system of DAEs that
(a) contain the derivative, `der`, operator
(b) can be written as $der(x) = e$, where x is a continuous state variable
As mentioned before, all conditions that occur in if-then-else expressions are collected and they define the modes of the system. The plant HybridSal module enumerates all modes of the system; and for each mode, it contains a system of differential equations extracted from the DAEs.

Each HybridSal module also has an initialization block, where variables can be initialized. Initial values for variables are extracted from the 'initialValue' attribute of the variables in Modelica XML dump. Some initialization equations are also extracted from the DAEs that contain the 'initial' function in the expressions.

```
{"context" : "TRUE",
 "property" :
 {"f"    : "G",
  "nargs": 1,
  "args" :
   [{"f"    : "=>",
     "nargs": 2,
     "args" :
   [{"f"    : ">=",
     "nargs": 2,
     "args" : ["MassSpringDamper_Mass_Steel_ModelicaModel_s", 1] },
    {"f"    : "<=",
     "nargs": 2,
     "args" : ["MassSpringDamper_Damper_Damper_mo_v_rel", -4] } ] } ] } }
```

Fig. 2. Example of a context and property description in json format that can be used as an optional input to the `modelica2hsal` translator. The context here is `True` and the property is $G(s \geq 1 \Rightarrow v_{rel} \leq -4)$, where s and v_{rel} denote the variables whose full names can be found in the json description above. G is the "always" temporal operator.

The monitor module. A separate HybridSal module is used to enforce that the system dynamics satisfy some constraints. Specifically, algebraic equations that can not be eliminated by simplification and equation solving can be just monitored. The monitor module is *synchronously composed* with the rest of the system, so if it can not make a step (because some constraint failed), then the whole system deadlocks. Monitors are also used to implement the context model.

Context and Property Modeling in HybridSal The `modelica2hsal` tool takes an optional second argument that is a file containing the context assumptions and the property to be proved of the model. This information can be provided in two different formats: either in XML format, or in JSON format.

Rather than describing the syntax of the json file, we just give an example of a sample json file in Figure 2. Similarly, we give an example of an XML representation of the context and property in Figure 3.

2.2 `modelica2hsal`: Limitations

The `modelica2hsal` converter can successfully convert several OpenModelica generated XML dumps into HybridSal. However, it is also unable to convert many more other XML dumps into HybridSal. In this section, we discuss the causes for this incompleteness.

The big sources of incompleteness are the following:

Trignometric functions. If the Modelica DAEs contain trignometric functions, then the `modelica2hsal` translator can fail to generate a HybridSal model. This is because HybridSal does not support trignometric functions

```
<CONTEXTPROPERTY>
<CONTEXT>
  <APPLICATION INFIX="YES"> <NAMEEXPR>AND</NAMEEXPR>
  <TUPLELITERAL>
    <APPLICATION INFIX="YES" PARENS="1"> <NAMEEXPR>&gt;</NAMEEXPR>
      <TUPLELITERAL>
        <NAMEEXPR>u</NAMEEXPR> <NUMERAL>500</NUMERAL>
      </TUPLELITERAL>
    </APPLICATION>
    <APPLICATION INFIX="YES" PARENS="1"> <NAMEEXPR>&lt;</NAMEEXPR>
      <TUPLELITERAL>
        <NAMEEXPR>u</NAMEEXPR> <NUMERAL>1000</NUMERAL>
      </TUPLELITERAL>
    </APPLICATION>
  </TUPLELITERAL>
  </APPLICATION>
</CONTEXT>
<PROPERTY>
  <APPLICATION> <LTLOP>G</LTLOP>
    <TUPLELITERAL>
      <APPLICATION INFIX="YES" PARENS="1"> <NAMEEXPR>&lt;</NAMEEXPR>
        <TUPLELITERAL>
          <NAMEEXPR>J_crank.w</NAMEEXPR> <NUMERAL>2000</NUMERAL>
        </TUPLELITERAL>
      </APPLICATION>
    </TUPLELITERAL>
  </APPLICATION>
</PROPERTY>
</CONTEXTPROPERTY>
```

Fig. 3. Property and context model presented in XML. The `modelica2hsal` converter can take this as an optional argument and generate HybridSal with necessary information. Here the context is $u \geq 500 \wedge u \leq 1000$, and the property is $G(\texttt{J\_crank.w} \leq 2000)$. G is the "always" temporal operator.

in its expression language. The reason why HybridSal does not include trignometric functions is that there is no easily decidable theory for them.

Nonlinear models. If the Modelica DAEs contain nonlinear expressions, such as $x*y$ or x^2 or $\sqrt{(x)}$, then the `modelica2hsal` translator can fail to generate a HybridSal model. This is because HybridSal does not support nonlinear models.

New user defined functions. If the Modelica DAEs contain calls to user-defined functions that are not already handled by the `modelica2hsal` translator, then the `modelica2hsal` translator can fail to generate a HybridSal model. This is because the `modelica2hsal` translator does not parse the C-like code of user-defined functions. Rather, handling of certain user-defined

functions that occur commonly in the models is hard coded into the translator. One such example is the linear interpolation function.

Special variables. If the Modelica DAEs contain special (predefined in Modelica) variables, then the `modelica2hsal` translator can fail to generate a HybridSal model. One such special variable is the `time` variable. Since this variable occurs frequently, we have added a flag `--addTime` to the translator that allows handling of Modelica models that contain the `time` variable. Alternatively, there is also an option of ignoring the `time` variable and *all* equations that contain the `time` variable. This is useful when the `time` variable is used to create a particular context for performing simulation. The flag `--removeTime` should be used in this case.

Processing of DAEs. If the Modelica DAEs are not simplified into a form that can be written as a set of switched ODEs with some discrete update, then the translator can fail to generate a HybridSal model.

Caveat: In some cases, it may happen that the translator terminates successfully and creates a HybridSal file, but the generated HybridSal file is not well formed. The translator, currently, does not do a well-formedness check on the generated file. For example, if the simplification procedure on the DAEs results in two equations that appear to be both setting the value of dx/dt, the generated HybridSal file can have two ODEs for dx/dt. The HybridSal relational abstraction tool can fail on such models. In these cases, a manual inspection of the generated HybridSal file may be required.

Caveat: The `modelica2hsal` translator has not been extensively tested. It remains unclear how to validate the translator, because there is nothing known about the input Modelica models that can be cross checked with the generated HybridSal model. While the translator attempts to perform a *semantic preserving* translation from Modelica to HybridSal, it may not always guarantee the equivalence of Modelica model and the HybridSal model. In most cases, we expect the output to be equivalent, and in cases when it is not, we expect the HybridSal model to be *more abstract* than the input Modelica model. This is because, on a few occassions, the translator may ignore some equations in the given DAEs, and hence, generate a model with more behaviors.

Caveat: One issue that arises when translating requirements from Modelica to HybridSal is the naming issue. In the process of model transformation and model simplification, variables get renamed, or eliminated, or mapped into other variables. Hence, a single state variable can have multiple names in different models. When translating requirements, it is important to ensure that the naming is consistent across the model and the properties. The current implementation of the translator does not perform this consistency check.

2.3 `modelica2hsal`: Performance

The translator simplifies each expression separately, and hence it is computationally very fast. So far, scalability of the translator has not been an issue. On the available benchmarks (RC Car component models, Mass Spring, etc.), the

translator is able to successfully translate about 50% of the Modelica models into HybridSal. The main cause for failure on the other models are trignometric functions.

As a final note, we remark that Modelica may not be a good choice for a front-end language that is driving backend verification tools. We give two main reasons below.

- Modelica is a language for building simulation models. As a result, it does not have some key features – such as nondeterminism – that are essential for a language describing (partial) designs, or highly abstract systems.
- Modelica does not provide a clean separation of three parts of a model – the context model, the actual system model, and the properties. Often, context model is tightly integrated into the system model.

3 HybridSal Relational Abstracter

After the Modelica model is translated into HybridSal, it is abstracted into a discrete `SAL` model using the relational abstraction tool. In this section, we will describe the theory and implementation of the relational abstractor.

A dynamical system $(\mathbb{X}, \xrightarrow{a})$ with state space $\mathbb{X}$ and transition relation $\xrightarrow{a} \subseteq \mathbb{X} \times \mathbb{X}$ is a *relational abstraction* of another dynamical system $(\mathbb{X}, \xrightarrow{c})$ if the two systems have the same state space and $\xrightarrow{c} \subseteq \xrightarrow{a}$. Since a relational abstraction contains all the behaviors of the concrete system, it can be used to perform safety verification.

HybridSAL relational abstracter is a tool that computes a relational abstraction of a hybrid system as described by Sankaranarayanan and Tiwari [8]. A hybrid system $(\mathbb{X}, \rightarrow)$ is a dynamical system with
(a) state space $\mathbb{X} := \mathbb{Q} \times \mathbb{Y}$, where $\mathbb{Q}$ is a finite set and $\mathbb{Y} := \mathbb{R}^n$ is the n-dimensional real space, and
(b) transition relation $\rightarrow := \rightarrow_{cont} \cup \rightarrow_{disc}$, where $\rightarrow_{disc}$ is defined in the usual way using guards and assignments, but $\rightarrow_{cont}$ is defined by a system of *ordinary differential equation* and a *mode invariant*. One of the key steps in defining the (concrete) semantics of hybrid systems is relating a system of differential equation $\frac{d\boldsymbol{y}}{dt} = f(\boldsymbol{y})$ with mode invariant $\phi(\boldsymbol{y})$ to a binary relation over $\mathbb{R}^n$, where $\boldsymbol{y}$ is a n-dimensional vector of real-valued variables. Specifically, the semantics of such a system of differential equations is defined as:

$$\begin{aligned} \boldsymbol{y}_0 \rightarrow_{cont} \boldsymbol{y}_1 \;\; & \text{if there is a } t_1 \in \mathbb{R}^{\geq 0} \text{ and a function } F \text{ from } [0, t_1] \text{ to } \mathbb{R}^n \text{ s.t.} \\ & \boldsymbol{y}_0 = F(0), \boldsymbol{y}_1 = F(t_1), \text{ and} \\ & \forall t \in [0, t_1] : \left(\frac{dF(t)}{dt} = f(F(t)) \wedge \phi(F(t)) \right) \end{aligned} \tag{13}$$

The concrete semantics is defined using the "solution" F of the system of differential equations. As a result, it is difficult to work directly with it.

The relational abstraction of a hybrid system $(\mathbb{X}, \xrightarrow{c}_{cont} \cup \xrightarrow{c}_{disc})$ is a discrete state transition system $(\mathbb{X}, \xrightarrow{a})$ such that $\xrightarrow{a} = \xrightarrow{a}_{cont} \cup \xrightarrow{c}_{disc}$, where $\xrightarrow{c}_{cont} \subseteq$

$\xrightarrow{a}_{cont}$. In other words, the discrete transitions of the hybrid system are left untouched by the relational abstraction, and only the transitions defined by differential equations are abstracted.

The HybridSal relational abstracter tool computes such a relational abstraction for an input hybrid system. In this section, we describe the tool, the core algorithm implemented in the tool, and we also provide some examples.

3.1 `hsal2sal`: Abstraction Algorithm

Given a system of linear ordinary differential equation, $\frac{d\boldsymbol{x}}{dt} = A\boldsymbol{x}+\boldsymbol{b}$, we describe the algorithm used to compute the abstract transition relation $\xrightarrow{a}$ of the concrete transition relation $\xrightarrow{c}$ defined by the differential equations.

The algorithm is described in Figure 4. The input is a pair (A, b), where A is a $(n\times n)$ matrix of rational numbers and $\boldsymbol{b}$ is a $(n\times 1)$ vector of rational numbers. The pair represents a system of differential equations $\frac{d\boldsymbol{x}}{dt} = A\boldsymbol{x} + \boldsymbol{b}$. The output is a formula ϕ over the variables $\boldsymbol{x}, \boldsymbol{x'}$ that represents the relational abstraction of $\frac{d\boldsymbol{x}}{dt} = A\boldsymbol{x} + \boldsymbol{b}$. The key idea in the algorithm is to use the eigenstructure of the matrix A to generate the relational abstraction.

The following proposition states the correctness of the algorithm.

Proposition 1. *Given (A, b), let ϕ be the output of procedure* `linODEabs` *in Figure 4. If $\rightarrow_{cont}$ is the binary relation defining the semantics of $\frac{d\boldsymbol{x}}{dt} = A\boldsymbol{x} + b$ with mode invariant True (as defined in Equation 13), then $\rightarrow_{cont} \subseteq \phi$.*

By applying the above abstraction procedure on to the dynamics of each mode of a given hybrid system, the HybridSal relational abstracter constructs a relational abstraction of a hybrid system. This abstract system is a purely discrete infinite state space system that can be analyzed using infinite bounded model checking (inf-BMC), k-induction, or abstract interpretation.

We make two important remarks here. First, the relational abstraction constructed by procedure `linODEabs` is a Boolean combination of linear *and nonlinear* expressions. The nonlinear expressions can be replaced by their conservative linear approximations. The HybridSal relational abstracter performs this approximation by default. It generates the (more precise) nonlinear abstraction (as described in Figure 4) when invoked using an appropriate command line flag. Both inf-BMC and k-induction provers rely on satisfiability modulo theory (SMT) solvers. Most SMT solvers can only reason about *linear* constraints, and hence, the ability to generate linear relational abstractions is important. However, there is significant research effort going on into extending SMT solvers to handle nonlinear expressions. HybridSal relational abstracter and SAL inf-BMC have been used to create benchmarks for linear *and nonlinear* SMT solvers.

Second, Procedure `linODEabs` can be extended to generate even more precise *nonlinear* relational abstractions of linear systems. Let $p_1, p_2, \ldots, p_k$ be k (linear and nonlinear) expressions found by Procedure `linODEabs` that satisfy the equation $\frac{dp_i}{dt} = \lambda_i p_i$. Suppose further that there is some λ_0 s.t. for each i

`linODEabs`(A, b): *Input*: a pair (A, b), where $A \in \mathbb{R}^{n\times n}, b \in \mathbb{R}^{n\times 1}$.
Output: a formula ϕ over the variables $\boldsymbol{x}, \boldsymbol{x'}$

1. identify all variables $x_1, \ldots, x_k$ s.t. $\frac{dx_i}{dt} = b_i$ where $b_i \in \mathbb{R}\ \ \forall i$
   let E be $\{\frac{x_i' - x_i}{b_i} \mid i = 1, \ldots, k\}$
2. partition the variables $\boldsymbol{x}$ into $\boldsymbol{y}$ and $\boldsymbol{z}$ s.t. $\frac{d\boldsymbol{x}}{dt} = A\boldsymbol{x} + \boldsymbol{b}$ can be rewritten as

$$\begin{bmatrix} \frac{d\boldsymbol{y}}{dt} \\ \frac{d\boldsymbol{z}}{dt} \end{bmatrix} = \begin{bmatrix} A_1 & A_2 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \boldsymbol{y} \\ \boldsymbol{z} \end{bmatrix} + \begin{bmatrix} \boldsymbol{b_1} \\ \boldsymbol{b_2} \end{bmatrix}$$

   where $A_1 \in \mathbb{R}^{n_1\times n_1}$, $A_2 \in \mathbb{R}^{n_1\times n_2}$, $\boldsymbol{b_1} \in \mathbb{R}^{n_1\times 1}$, $\boldsymbol{b_2} \in \mathbb{R}^{n_2\times 1}$, and $n = n_1 + n_2$
3. set ϕ to be *True*
4. let $\boldsymbol{c}$ be a real left eigenvector of matrix A_1 and let λ be the corresponding real eigenvalue, that is, $\boldsymbol{c}^T A_1 \ = \ \lambda \boldsymbol{c}^T$
5. if $\lambda == 0 \wedge c^T A_2 == 0$: set $E := E \cup \{\frac{\boldsymbol{c}^T(\boldsymbol{y'} - \boldsymbol{y})}{\boldsymbol{c}^T \boldsymbol{b_1}}\}$; else: $E := E$
6. if $\lambda \neq 0$: define vector $\boldsymbol{d}$ and real number e as: $\boldsymbol{d}^T = \boldsymbol{c}^T A_2/\lambda$ and $e = (\boldsymbol{c}^T \boldsymbol{b_1} + \boldsymbol{d}^T \boldsymbol{b_2})/\lambda$
   let $p(\boldsymbol{x})$ denote the expression $\boldsymbol{c}^T \boldsymbol{y} + \boldsymbol{d}^T \boldsymbol{z} + e$ and let $p(\boldsymbol{x'})$ denote $\boldsymbol{c}^T \boldsymbol{y'} + \boldsymbol{d}^T \boldsymbol{z'} + e$
   if $\lambda > 0$: set $\phi := \phi \wedge [(p(\boldsymbol{x'}) \leq p(\boldsymbol{x}) < 0) \vee (p(\boldsymbol{x'}) \geq p(\boldsymbol{x}) > 0) \vee (p(\boldsymbol{x'}) = p(\boldsymbol{x}) = 0)]$
   if $\lambda < 0$: set $\phi := \phi \wedge [(p(\boldsymbol{x}) \leq p(\boldsymbol{x'}) < 0) \vee (p(\boldsymbol{x}) \geq p(\boldsymbol{x'}) > 0) \vee (p(\boldsymbol{x'}) = p(\boldsymbol{x}) = 0)]$
7. if there are more than one eigenvectors corresponding to the eigenvalue λ, then update ϕ or E by generalizing the above
8. repeat Steps (4)–(7) for each pair $(\boldsymbol{c}, \lambda)$ of left eigenvalue and eigenvector of A_1
9. let $\boldsymbol{c} + \imath \boldsymbol{d}$ be a complex left eigenvector of A_1 corresponding to eigenvalue $\alpha + \imath\beta$
10. using simple linear equation solving as above, find $\boldsymbol{c_1}$, $\boldsymbol{d_1}$, e_1 and e_2 s.t. if p_1 denotes $\boldsymbol{c}^T \boldsymbol{y} + \boldsymbol{c_1}^T \boldsymbol{z} + e_1$ and if p_2 denotes $\boldsymbol{d}^T \boldsymbol{y} + \boldsymbol{c_2}^T \boldsymbol{z} + e_2$ then

$$\frac{d}{dt}(p_1) \ = \ \alpha p_1 - \beta p_2 \qquad \frac{d}{dt}(p_2) \ = \ \beta p_1 + \alpha p_2$$

    let p_1' and p_2' denote the primed versions of p_1, p_2
11. if $\alpha \leq 0$: set $\phi := \phi \wedge (p_1^2 + p_2^2 \geq {p_1'}^2 + {p_2'}^2)$
    if $\alpha \geq 0$: set $\phi := \phi \wedge (p_1^2 + p_2^2 \leq {p_1'}^2 + {p_2'}^2)$
12. repeat Steps (9)–(11) for every complex eigenvalue eigenvector pair
13. set $\phi := \phi \wedge \bigwedge_{e_1, e_2 \in E} e_1 = e_2$; return ϕ

Fig. 4. Algorithm implemented in HybridSal relational abstracter for computing relational abstractions of linear ordinary differential equations.

$\lambda_i = n_i \lambda_0$ for some *integer* n_i. Then, we can extend ϕ by adding the following relation to it:

$$p_i(\boldsymbol{x'})^{n_j} p_j(\boldsymbol{x})^{n_i} = p_j(\boldsymbol{x'})^{n_i} p_i(\boldsymbol{x})^{n_j} \tag{14}$$

However, since p_i's are linear or quadratic expressions, the above relations will be highly nonlinear unless n_i's are small. So, they are not currently generated by the relational abstracter. It is left for future work to see if good and useful linear approximations of these highly nonlinear relations can be obtained.

3.2 `hsal2sal`: The Relational Abstracter Tool

The HybridSal relational abstracter tool, including the sources, documentation and examples, is freely available for download [10].

The input to the tool is a file containing a specification of a hybrid system and safety properties. The HybridSal language naturally extends the SAL language by providing syntax for specifying ordinary differential equations. SAL is a guarded command language for specifying discrete state transition systems and supports modular specifications using synchronous and asynchronous composition operators. The reader is referred to [7] for details. HybridSal inherits all the language features of SAL. Additionally, HybridSal allows differential equations to appear in the model as follows: for each real-valued variable x, the user defines a dummy variable *xdot* which represents $\frac{dx}{dt}$. A differential equation can now be written by assigning to the *xdot* variable. Assuming two variables x, y, the syntax is as follows:

```
guard(x,y) AND guard2(x,x',y,y') --> xdot' = e1; ydot' = e2
```

This represents the system of differential equations $\frac{dx}{dt} = e1, \frac{dy}{dt} = e2$ with mode invariant $guard(x, y)$. The semantics of this guarded transition is the binary relation defined in Equation 13 conjuncted with the binary relation $guard2(x, x', y, y')$. The semantics of all other constructs in HybridSal match exactly the semantics of their counterparts in SAL.

Figure 5 contains sketches of two examples of hybrid systems modeled in HybridSal. The example in Figure 5(left) defines a module `SimpleHS` with two real-valued variables x, y. Its dynamics are defined by $\frac{dx}{dt} = -y + x$, $\frac{dy}{dt} = -y - x$ with mode invariant $y \geq 0$, and by a discrete transition with guard $y \leq 0$. The HybridSal file `SimpleHS.hsal` also defines two safety properties. The latter one says that x is always non-negative. This model is analyzed by abstracting it

```
bin/hsal2hasal examples/SimpleEx.hsal
```

to create a relational abstraction in a SAL file named `examples/SimpleEx.sal`, and then (bounded) model checking the SAL file

```
sal-inf-bmc -i -d 1 SimpleEx helper
sal-inf-bmc -i -d 1 -l helper SimpleEx correct
```

The above commands prove the safety property using k-induction: first we prove a lemma, named `helper`, using 1-induction and then use the lemma to prove the main theorem named `correct`.

The example in Figure 5(right) shows the sketch of a model of the train-gate-controller example in HybridSal. All continuous dynamics are moved into one module (named `timeElapse`). The `train`, `gate` and `controller` modules define the state machines and are pure SAL modules. The `observer` module is also a pure SAL module and its job is to enforce synchronization between modules on events. The final system is a complex composition of the base modules.

The above two examples, as well as, several other simple examples are provided in the HybridSal distribution to help users understand the syntax and working of the relational abstracter. A notable (nontrivial) example in the distribution is a hybrid model of an automobile's automatic transmission from [2].

```
SimpleEx: CONTEXT = BEGIN
 SimpleHS: MODULE = BEGIN
  LOCAL x,y,xdot,ydot:REAL
  INITIALIZATION
   x = 1; y IN {z:REAL| z <= 2}
  TRANSITION
   [ y >= 0 AND y' >= 0 -->
       xdot' = -y + x ;
       ydot' = -y - x
   [] y <= 0 --> x' = 1; y' = 2]
 END;
 helper: LEMMA SimpleHS |-
   G(0.9239*x >= 0.3827*y);
 correct : THEOREM
   SimpleHS |- G(x >= 0);
END
```

```
TGC: CONTEXT = BEGIN
 Mode: TYPE = {s1, s2, s3, s4};
 timeElapse: MODULE = BEGIN
  variable declarations
  INITIALIZATION x = 0; y = 0; z = 0
  TRANSITION
   [mode invariants -->
     --> xdot' = 1; ydot' = 1; zdot' = 1]
 END;
 train: MODULE = ...
 gate: MODULE = ...
 controller: MODULE = ...
 observer: MODULE = ...
 system: MODULE = (observer || (train []
   gate [] controller [] timeElapse));
 correct: THEOREM system |- G ( ... ) ;
END
```

Fig. 5. Modeling hybrid systems in HybridSal: A few examples.

Users have to separately download and install SAL model checkers if they wish to analyze the output SAL files using k-induction or infinite BMC.

The HybridSal relational abstracter constructs abstractions compositionally; i.e., it works on each mode (each system of differential equations) separately. It just performs some simple linear algebraic manipulations and is therefore very fast. The bottleneck step in our tool chain is the inf-BMC and k-induction step, which is orders of magnitude slower than the abstraction step (we have not tried abstract interpretation yet). The performance of HybridSal matches the performance reported in our earlier paper [8] on the navigation benchmarks (which are included with the HybridSal distribution). In [8] we had used many different techniques (not all completely automated at that time) to construct the relational abstraction.

3.3 `hsal2sal`: Strengths and Limitations

The HybridSal relational abstracter is a tool for verifying hybrid systems. The other common tools for hybrid system verification consist of
(a) tools that iteratively compute an overapproximation of the reachable states [4],
(b) tools that directly search for correctness certificates (such as inductive invariants or Lyapunov function) [6, 9], or
(c) tools that compute an abstraction and then analyze the abstraction [5, 1, 3].
Our relational abstraction tool falls in category (c), but unlike all other abstraction tools, it does not abstract the state space, but abstracts only the transition relation.

The key benefit of relational abstraction is that it cleanly separates reasoning on continuous dynamics (where we use control theory or systems theory)

and reasoning on discrete state transition systems (where we use formal methods.) Concepts such as Lyapunov functions or inductive invariants (aka barrier certificates) for continuous systems are used to construct very precise relational abstractions, and formal methods is used to verify the abstracted system. In fact, for several classes of simple continuous dynamical systems, lossless relational abstractions can be constructed, and hence all incompleteness in verification then comes from incompleteness of verification of infinite state transition systems.

We note that our tool is the first in its space and is still under active development. We list below some of its shortcomings and ways in which we plan to address them in the future.

Precision. While relational abstractions are more precise than purely qualitative abstractions, there is occassionally a need for more precise abstraction. There are several ways forward. One approach is to enhance relational abstractions with qualitative abstractions. There are examples where the combination gives a much better quality abstraction than either components. Also, more precise relational abstractions can be potentially generated by using mode invariants.

Numerical Issues. The computation of relational abstraction is performed using floating point arithmetic computation. It is well known that floating point computations can lead to unsound results. In future versions of the tool, numerical errors due to floating point arithmetic need to be handled properly.

Nonlinearity. Our tool is restricted to handling linear differential equations. It can not handle nonlinear differential equations presently. There is a need to develop support for nonlinear differential equations to analyze more complex hybrid models.

We remark here that scalability is not an issue for computing relational abstractions. It can be shown that relational abstractions can be computed in polynomial (cubic) time. The process of computing the abstraction is generally fast. It is the verification of the abstraction, discussed more in the next section, that causes scalability concerns.

4 The SAL Model Checking Tools

The SAL model generated by the relational abstraction tool is analyzed using the SAL model checkers. In this section, we briefly describe the tools and their available options for analyzing the output of the relational abstraction tool.

Recall that the relational abstraction model has the *same* state space as the original model. Only the transitions are abstracted. Hence, the SAL model is an infinite state system. Standard model checkers, which can handle only finite state systems, can not be used to verify the abstract models.

The SAL tool suite provides two techniques for verifying infinite state systems.

Infinite bounded model checking. Infinite bounded model checking is used to verify the safety property upto a fixed finite depth. If the safety property is not true, and there is a violation of the property that is exhibited by a trajectory making at most d discrete mode switches, then preforming infinite bounded model checking with depth d will discover the violation. However, infinite bounded model checking can not, by itself, prove the safety property. It can only show that there are no violations of the safety property for certain bounded behaviors. The tool `sal-inf-bmc` is an infinite bounded model checker for SAL.

k-induction. A safety property can be proved of a model using k-induction. The principle of k-induction is a generalization of the standard 1-step induction. A successful proof using k-induction can demonstrate that there is no violation of the safety property for any *arbitrary* depth. With the `-i` flag, the tool `sal-inf-bmc -i` becomes a k-induction prover.

4.1 `sal-inf-bmc`: Limitations of SAL Verification Tools

There are certain limitations of the SAL verification tools.

Scalability. Relational abstraction generates verification problem on a discrete, infinite state space system, which are difficult to verify automatically. In this project, we have used k-induction and infinite bounded model checking. The scalability of these techniques is limited since the SMT formulas generated (from the SAL model by the infinite bounded model checker or the k-induction prover) can be very large. The use of k-induction can require the need for auxiliary lemmas. In the future, we plan to develop dedicated methods for generating invariants and for performing abstract interpretation that are tailored to analyzing relational abstractions.

Expressiveness. The approach of verification based on abstraction and model checking can only be used to verification of *safety* properties. While safety properties constitute the majority of the requirements, there are other properties (liveness) that are some times useful, which presently can not be analyzed using our approach.

5 Drivetrain Case Study

In this section, we present the HybridSAL model of the drivetrain case study. The drivetrain was originally modeled in Modelica. The Modelica file was translated by hand into a HybridSAL model shown in Figure 6 and Figure 7.

We briefly describe the HybridSAL models in Figure 6 and Figure 7. First, note that the drivetrain `system` is modeled as a composition of two submodules, namely `plant` and `control`. This is captured in Figure 7 as:

```
system: MODULE = plant || control ;
```

Figure 6 contains the description of module `plant` and Figure 7 contains the description of module `control`.

The state space of the `plant` module is described by four real-valued variables, f5, f7, f17 and e11. Note that, in the original Modelica model, these variables had appeared within the derivative (`der`) operator. All the other variables in the Modelica model were eliminated during the translation to HybridSAL. The `plant` module has as input the variable `ratio`. The `control` module outputs (writes) `ratio` and the `plant` module inputs (reads) it. Similarly, the `control` module reads the value of `f5` and `f17`, whereas the `plant` module writes it. After the variable declarations, a HybridSAL module has two other important declarations: `Initialization` and `Transition` sections. All state variables are initialized to zero thus,

```
INITIALIZATION f5 = 0; f7 = 0; e11 = 0; f17 = 0
```

In the `Transition` section, we give one set of ordinary differential equations (ODEs) for each mode of the system. A set of ODEs consists of a differential equation for each of the four variables. Since ratio can take six different values, there are six modes and six sets of ODEs in the module `Plant`. The formula before the `-->` in a guarded command is a guard that specifies what condition must hold at the beginning (unprimed variables) and at the end (primed variables) of the transition. For example, the differential equations describing the plant when `ratio` is 1/4 is written as

```
ratio = 1/4 AND (ratio' = 1/3 OR ratio' = 1) -->
   f5dot' = e11*10000/2400 - f5/20 ;
   f7dot' = 40 + 20 * f17 - 20 * f7 ;
   e11dot' = f17 * 1000 * 1/4 - f5 * 10000/6 ;
   f17dot' = 100*f7 - 100*f17 - e11* 10000/40
```

The guard captures the fact that if ratio is 1/4 now, it will either be 1/3 or 1 in the *next* step. This forces a transition to necessarily change `ratio` in each step.

The `control` module is shown in Figure 7. It is a purely discrete state transition system – it involves no differential equations. The logic for changing the value of `ratio` is self evident from Figure 7.

It is also possible to model a `time triggered` version of the plant and control. This is shown in Figure 8. The only difference between the timed and untimed model is in the guards of `plant`. In the untimed version, we set the guards so that the value of `ratio` changes in every single step. In the timed version, we can not force the value of `ratio` to change in every single step (since it may not depending on the sampling period), and hence we simplify the guards.

The HybridSAL model(s) can be abstracted using the relational abstraction tools. The usage of these tools is described in Appendix A. The relational abstraction tool eliminates the differential equations and outputs a SAL file that contains no differential equations. The following two SAL tools can be used to analyze the generated abstract file.

`sal-inf-bmc`: SAL infinite bounded model checker
`sal-inf-bmc -i`: SAL k-induction prover

We note that these two tools use Yices as their background satisfiability modulo theory (SMT) constraint solver.

6 Conclusion

We presented the three components of the HybridSal relational abstraction verification tool:
(a) A translator that converts Modelica models into HybridSal models
(b) A relational abstraction tool that converts a HybridSal model into a SAL model and
(c) An infinite bounded model checking based verification tool for analyzing infinite state SAL models.
Each of these three components can be used independently. However, our verification tool internally composes these three parts and directly verifies safety properties of Modelica models. It currently runs on all platforms, Linux, Mac, and Windows.

References

1. R. Alur, T. Dang, and F. Ivancic. Counter-example guided predicate abstraction of hybrid systems. In *TACAS*, volume 2619 of *LNCS*, pages 208–223, 2003.
2. A. Chutinan and K. R. Butts. *SmartVehicle baseline report: Dynamic analysis of hybrid system models for design validation.* Ford Motor Co., 2002. Tech. report, Open Experimental Platform for DARPA MoBIES, Contract F33615-00-C-1698.
3. E. M. Clarke, A. Fehnker, Z. Han, B. H. Krogh, O. Stursberg, and M. Theobald. Verification of hybrid systems based on counterexample-guided abstraction refinement. In *TACAS*, volume 2619 of *LNCS*, pages 192–207, 2003.
4. G. Frehse, C. L. Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler. Spaceex: Scalable verification of hybrid systems. In *CAV*, volume 6806 of *LNCS*, pages 379–395, 2011.
5. Hybridsal: Modeling and abstracting hybrid systems. `http://www.csl.sri.com/users/tiwari/HybridSalDoc.ps`.
6. S. Prajna, A. Papachristodoulou, and P. A. Parrilo. SOSTOOLS: Sum of Square Optimization Toolbox for MATLAB, 2002. Available from `http://www.cds.caltech.edu/sostools` and `http://www.aut.ee.ethz.ch/~parrilo/sostools`.
7. The SAL intermediate language, 2003. Computer Science Laboratory, SRI International, Menlo Park, CA. `http://sal.csl.sri.com/`.
8. S. Sankaranarayanan and A. Tiwari. Relational abstractions for continuous and hybrid systems. In *CAV*, volume 6806 of *LNCS*, pages 686–702, 2011.
9. T. Sturm and A. Tiwari. Verification and synthesis using real quantifier elimination. In *ISSAC*, pages 329–336, 2011.
10. A. Tiwari. Hybridsal relational abstracter. `http://www.csl.sri.com/~tiwari/relational-abstraction/`.

A HybridSAL relational abstracter man pages

```
NAME
        bin/hsal2hasal - construct relational abstraction of
                         HybridSAL models

SYNOPSIS
        bin/hasal [OPTION]... [FILE]

DESCRIPTION
        Construct a relational abstraction of the model in [FILE].
        Create a new SAL file containing the abstract model.
        Input file is expected to be in HybridSAL (.hsal) syntax,
        or HybridSAL's XML representation (.hxml).
        The new file will have the same name as [FILE], but
        a different extension, .sal

        Options include:
        -c, --copyguard
            Explicitly handle the guards in the continuous dynamics
            as state invariants
        -n, --nonlinear
            Create a nonlinear abstract model
            Note that freely available model checkers are unable
            to handle nonlinear models, hence this option is
            useful for research purposes only
        -t <T>, --timed <T>
            Create a timed relational abstraction assuming that
            the controller is run every <T> time units.
            <T> should be a number (such as, 0.01)
        -o, --optimize
            Create an optimized relational abstraction.
            Certain transient's are unsoundly eliminated from the
            abstract SAL model to improve performance of the model
            checkers on the generated SAL model

AUTHOR
        Written by Ashish Tiwari

REPORTING BUGS
        Report bugs to ashish_dot_tiwari_at_sri_dot_com

COPYRIGHT
        Copyright  2011 Ashish Tiwari, SRI International.

-------------------------------------------------------------------
```

```
NAME
        hybridsal2xml - convert hybridsal into XML format

SYNOPSIS
        hybridsal2xml [OPTION]... [FILE]

DESCRIPTION
        Parse the HybridSAL file [FILE] (with .hsal extension) and
        create a new file containing its XML representation.
        The new file will have the same name as [FILE], but
        a different extension, namely .hxml.

        Options include:
        -o <filename>
            Save the XML in file <filename>

AUTHOR
        Written by Ashish Tiwari

REPORTING BUGS
        Report bugs to ashish_dot_tiwari_at_sri_dot_com

COPYRIGHT
        Copyright  2011 Ashish Tiwari, SRI International.

-------------------------------------------------------------------

NAME
        bin/hxml2hsal - convert hybridsal XML to standard notation

SYNOPSIS
        bin/hxml2hsal [FILE]

DESCRIPTION
        Pretty print the XML file [FILE] as a HybridSAL file.
        The input is assumed to be a file with extension .hxml
        The output is written in a new file that has the same
        name as [FILE], but a different extension, namely .hsal.

        This is the ``inverse'' of hybridsal2xml tool.

AUTHOR
        Written by Ashish Tiwari

REPORTING BUGS
```

```
        Report bugs to ashish_dot_tiwari_at_sri_dot_com

COPYRIGHT
        Copyright  2011 Ashish Tiwari, SRI International.

------------------------------------------------------------------

NAME
        bin/hasal2sal - Extract SAL from HybridSal

SYNOPSIS
        bin/hasal2sal [FILE]

DESCRIPTION
        Extract the discrete part of the transition system
        contained in the hybrid abstract HybridSAL file [FILE],
        with .hasal extension, and pretty print it.
        The input is assumed to be a file with extension .hasal
        The output is written in a new file that has the same
        name as [FILE], but a different extension, namely .sal.

        There is no analysis performed; it is a purely
        syntactic extraction.

AUTHOR
        Written by Ashish Tiwari

REPORTING BUGS
        Report bugs to ashish_dot_tiwari_at_sri_dot_com

COPYRIGHT
        Copyright  2011 Ashish Tiwari, SRI International.
------------------------------------------------------------------
modelica2hsal -- a converter from Modelica to HybridSal

NAME
        bin/modelica2hsal - convert Modelica XML to Hybridsal

SYNOPSIS
        bin/modelica2hsal <Modelica.xml> [<context-property-file>]
                [--addTime|--removeTime]

DESCRIPTION
        Converts Modelica model in <Modelica.xml> to HybridSal.
        The optional argument <context-property-file> is a file
```

```
        containing the context model and property: in json or xml.
        The optional argument --addTime assumes time as an
        implicitly defined variable.
        The optional argument --removeTime removes the time
        variable and all equations containing time before
        performing the translation to HybridSal.

AUTHOR
        Written by Ashish Tiwari

COPYRIGHT
        Copyright  2011 Ashish Tiwari, SRI International.
-------------------------------------------------------------------
```

```
drivetrain: CONTEXT =
BEGIN
plant: MODULE =
BEGIN
OUTPUT f5, f7, e11, f17:REAL
INPUT ratio: REAL
INITIALIZATION
   f5 = 0; f7 = 0; e11 = 0; f17 = 0
TRANSITION
[
ratio = 1/4 AND (ratio' = 1/3 OR ratio' = 1) -->
   f5dot' = e11*10000/2400 - f5/20 ;
   f7dot' = 40 + 20 * f17 - 20 * f7 ;
   e11dot' = f17 * 1000 * 1/4 - f5 * 10000/6 ;
   f17dot' = 100*f7 - 100*f17 - e11* 10000/40
[]
ratio = 1/3 AND (ratio' = 1/3 OR ratio' = 3/7) -->
   f5dot' = e11*10000/2400 - f5/20 ;
   f7dot' = 40 + 20 * f17 - 20 * f7 ;
   e11dot' = f17 * 1000 * 1/3 - f5 * 10000/6 ;
   f17dot' = 100*f7 - 100*f17 - e11* 10000/30
[]
ratio = 3/7 AND (ratio' = 1/3 OR ratio' = 3/5) -->
   f5dot' = e11*10000/2400 - f5/20 ;
   f7dot' = 40 + 20 * f17 - 20 * f7 ;
   e11dot' = f17 * 1000 * 3/7 - f5 * 10000/6 ;
   f17dot' = 100*f7 - 100*f17 - e11*10000* 3/70
[]
ratio = 3/5 AND (ratio' = 3/7 OR ratio' = 1) -->
   f5dot' = e11*10000/2400 - f5/20 ;
   f7dot' = 40 + 20 * f17 - 20 * f7 ;
   e11dot' = f17 * 1000 * 3/5 - f5 * 10000/6 ;
   f17dot' = 100*f7 - 100*f17 - e11*10000* 3/50
[]
ratio = 1 AND (ratio' = 3/5 OR ratio' = 7/5) -->
   f5dot' = e11*10000/2400 - f5/20 ;
   f7dot' = 40 + 20 * f17 - 20 * f7 ;
   e11dot' = f17 * 1000 * 1 - f5 * 10000/6 ;
   f17dot' = 100*f7 - 100*f17 - e11* 10000/10
[]
ratio = 7/5 AND ratio' = 1 -->
   f5dot' = e11*10000/2400 - f5/20 ;
   f7dot' = 40 + 20 * f17 - 20 * f7 ;
   e11dot' = f17 * 1000 * 7/5 - f5 * 10000/6 ;
   f17dot' = 100*f7 - 100*f17 - e11*10000* 7/50
]
END;
```

Fig. 6. HybridSAL model of the drivetrain case study. The `plant` module describes the dynamics of the physical plant. The rest of the HybridSAL model is shown in Figure 7.

```
control: MODULE =
BEGIN
OUTPUT ratio: REAL
LOCAL G1, G2, G3, G4, G5, G6: REAL
LOCAL G2Low, G3Low, G4Low, G5Low, G6Low: REAL
LOCAL G1High, G2High, G3High, G4High, G5High, G6High: REAL
INPUT f17, f5: REAL
LOCAL OutRPM: REAL
INITIALIZATION ratio = 1/4
DEFINITION G1 = 1/4; G2 = 1/3; G3 = 3/7; G4 = 3/5; G5 = 1; G6 = 7/5;
   G2Low = G1High; G3Low = G2High;
   G4Low = G3High; G5Low = G4High;
   G6Low = G5High;
   G1High = G1 * 1800; G2High = G2 * 1800;
   G3High = G3 * 1800; G4High = G4 * 1800;
   G5High = G5 * 1800; G6High = G6 * 1800;
   OutRPM = f5 * 16
TRANSITION
[
   OutRPM' < G1High --> ratio' = G1
   []
   OutRPM' >= G2Low AND OutRPM' < G2High --> ratio' = G2
   []
   OutRPM' >= G3Low AND OutRPM' < G3High --> ratio' = G3
   []
   OutRPM' >= G4Low AND OutRPM' < G4High --> ratio' = G4
   []
   OutRPM' >= G5Low AND OutRPM' < G5High --> ratio' = G5
   []
   OutRPM' >= G6Low AND OutRPM' < G6High --> ratio' = G6
   []
   ELSE --> ratio' = 1
]
END;

system:MODULE = plant || control ;

correct : THEOREM system |- G( f7 >= 0 OR f17 >= 0);

reach : THEOREM system |- G( ratio <= 1/2 );
END
```

Fig. 7. HybridSAL model of the drivetrain case study. The `control` module describes the logic for switching between the different modes of the plant. The first part of the HybridSAL model is shown in Figure 6. We have also shown some (dummy) properties.

```
timedDrivetrain: CONTEXT =
BEGIN
plant: MODULE =
BEGIN
OUTPUT f5, f7, e11, f17:REAL
INPUT ratio: REAL
INITIALIZATION
   f5 = 0; f7 = 0; e11 = 0; f17 = 0
TRANSITION
[
ratio = 1/4 -->
   f5dot' = e11*10000/2400 - f5/20 ;
   f7dot' = 40 + 20 * f17 - 20 * f7 ;
   e11dot' = f17 * 1000 * 1/4 - f5 * 10000/6 ;
   f17dot' = 100*f7 - 100*f17 - e11* 10000/40
[]
ratio = 1/3 --> ...  same as before
[]
ratio = 3/7 --> ...  same as before
[]
ratio = 3/5 --> ...  same as before
[]
ratio = 1 --> ...  same as before
[]
ratio = 7/5 --> ...  same as before
]
END;
control: MODULE =
BEGIN
   same as before
]
END;

system:MODULE = plant || control ; END
```

Fig. 8. HybridSAL model of the time triggered implementation of the drivetrain case study. But for the guards of `plant`, this model is identical to the model in Figure 6.