# Making Modelica Applicable for Formal Methods

**Matthew Klenk  Daniel G. Bobrow  Johan de Kleer  Bill Janssen**
Palo Alto Research Center
3333 Coyote Hill Rd, Palo Alto, CA, 94304
contact email: Matthew.Klenk@parc.com

## Abstract

Engineers need to perform many different types of analyses as they design systems. Modelica has become a leading language to support numerical simulation. As a consequence there is widespread understanding of Modelica and a large number of Modelica model libraries available. This paper addresses the task of using formal methods to derive system properties such as whether a design meets its requirements for all possible inputs. We report on our experience building a qualitative reasoner operating on Modelica models. In this paper, we highlight five Modelica modeling practices that impede the application of formal methods.

## 1 Introduction

Modelica [Fritzson, 2004] is a powerful language for specifying the behaviors of components represented by declarative constructs connected through power ports. Modelica provides designers with large libraries of standard models and compile time computation to create large models. These features attract designers interested in numeric simulation and researchers developing new analyses. In contrast, the languages qualitative reasoning [Weld and de Kleer, 1989] and other hybrid system verification methods (e.g., HybridSAL [Tiwari, 2012]) require equation-based models. Engineers use these formal methods to prove that systems will never reach critical states for all possible parameter values in a section of the design space. In previous work, we have discussed how qualitative reasoning can be used on Modelica models consisting only of a subset of the language [Klenk *et al.*, 2012].

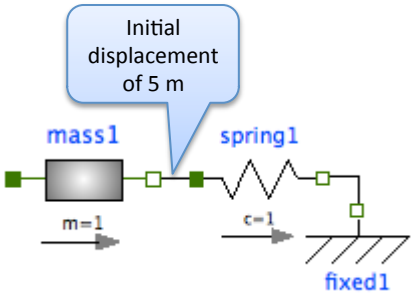
In addition to this common core, Modelica allows designers to specify behavior using algorithms and provide advice for simulation engines. While designers desire this flexibility and control, these features make qualitative reasoning and other formal methods difficult to apply to Modelica models. In addition to simulation advice and explicit algorithms, we identify three other modeling practices that hinder the application of formal methods: unnecessary component model complexity, use of computational state, and incomplete models.

This paper is structured as follows. We begin with a brief overview of qualitative reasoning. Then, we discuss how the Modelica compiler may be used by formal methods. After which, we provide examples of each class of hindrances along with suggestions for improving the applicability of Modelica models for formal methods. We close with a discussion of related work and some general reflections on modeling.

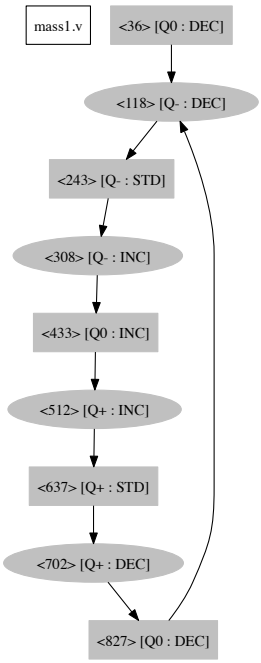
## 2 Qualitative reasoning and Design

Qualitative reasoning has its roots in automating reasoning about physical systems [Forbus, 1984][Kuipers, 1994][de Kleer and Williams, 1991]. Based on the intuition that engineers employ qualitative reasoning extensively throughout the design process, numerous researchers have sought to apply qualitative reasoning to design problems including functional reasoning [Everett, 1999][Wetzel and Forbus, 2009], diagnosis [Struss and Price, 2004], and automated FMEA generation [Snooke and Price, 2012]. An important subtask is *qualitative simulation* [Forbus, 1984][Kuipers, 1994], which provides an abstract description of the possible behaviors of a mathematical model. We illustrate qualitative simulation as well as some uses in the design context with a series of examples.

First, consider the spring block system in Figure 1 with the initial condition of a displacement of 5 meters compressing the spring. Given a set of numeric parameters and a simulation duration, Modelica produces a numeric simulation (i.e., a sequence of real values for each variable). On the other hand, qualitative simulation begins by creating a set of abstractions for each variable. The simplest set is three qualitative values (Q-, Q0, and Q+) corresponding to the sign of the real valued quantity. Each continuous variable can have as many higher-order derivatives as necessary, each of which specifies a direction of change ($\downarrow$(DEC), $\rightarrow$(STD), $\uparrow$(INC)) at that derivative order. Qualitative simulation determines all possible sequences of qualitative states a system can go through over time, called an *envisionment*. Changes in the qualitative state occur when a variable or its derivative reaches a *landmark* (e.g., the displacement of the block crosses zero, the velocity of the block crosses zero). Crossing a landmark occurs in an instant that has no duration (represented as a rectangle) and approaching or departing a landmark occurs over an interval of time (represented as an oval). The numeric simulation produces a sequence of numbers that must be interpreted by the designer to understand the behavior. On the other hand, the envisionment illustrates directly that this system is oscillatory because the graph is a loop (i.e., the system returns to the same qualitative state). Furthermore, while the numeric simulation results apply to specific values for the mass and compliance of




(a) Spring block system



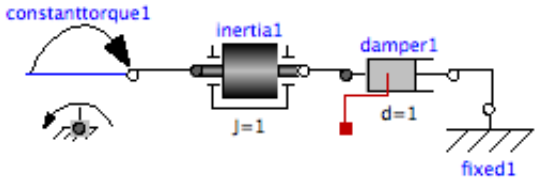

(b) Envisionment with qualitative values for the velocity and its derivative are shown. Each state is assigned a unique id. Ovals represent qualitative states that exist over an interval of time and rectangles represent qualitative states that exist only for an instant.

Figure 1: Qualitative simulation example

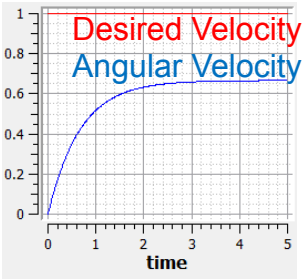
the spring, the envisionment illustrates that the behavior of the system in Figure 1 will be oscillatory for every set of parameters.

One reason for performing simulations is to determine if a system will meet some specified requirement. Figure 2 illustrates a simple rotational mechanical system with a specified requirement that the angular velocity must exceed 1 rad/s. This requirement can be encoded in Modelica using an enumerated type: `Unknown, Success, Violated`. The value of this variable begins as `Unknown` and changes to `Success` if the angular velocity of the flywheel exceeds 1 rad/s. As shown in the Modelica simulation (Figure 2), the system does not meet the requirement. Meanwhile, the envisionment (Figure 2) contains an interval in which the flywheel is accelerating followed by two branches: one where the requirement is met (shown as a green uparrow) and one where the inertia reaches its asymptote (shown as a blue rectangle). This multi-trajectory simulation illustrates the range of behaviors that are possible given underspecified parameters (e.g., the moment of inertia, applied torque, and damping coefficient are known only to be positive). The Modelica simulation in Figure 2 corresponds to the the following trajectory of qualitative states: 48 → 122 → 313. Because this envisionment includes a trajectory in which the requirement's value is `Success`, the designer knows that this topology may satisfy the requirement with different parameter values (e.g., increasing the torque or decreasing the damping factor).

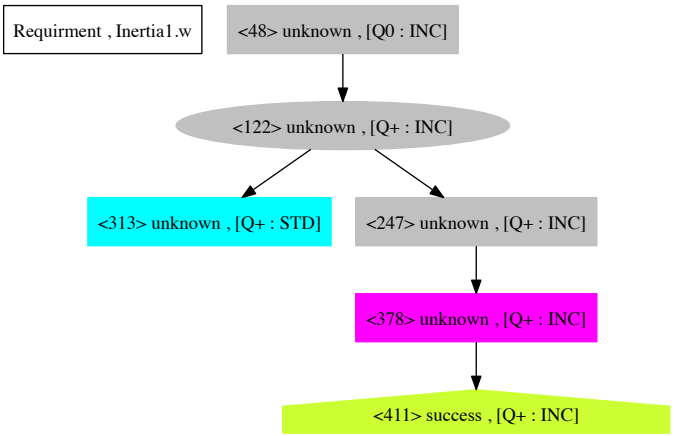
Figure 2: Qualitative Simulation and Requirements




(a) Modelica configuration with requirement that the flywheel reaches 1rad/s.




(b) Modelica simulation demonstrating that the current set of parameters does not meet the requirements within 5 seconds.




(c) The envisionment shows that this configuration could meet requirements with a different set of parameters. Cyan nodes represent terminal states, magenta nodes represent discrete events, and green uparrow nodes represent states that meet requirements.

Our intuition is that designers use this qualitative under-


DOI 10.3384/ECP14096205

standing of the design space to make decisions about components, parameters and configurations. Qualitative simulation is applicable early in the design process because it operates without completely specified parameters. Furthermore, qualitative simulation can be used to automate a number of the reasoning tasks designers perform during early design, answering such questions as:

- Could this configuration of components perform the desired function?
- What kinds of failures might this design have?
- How would this system behave when increasing a particular parameter?

Unfortunately, current qualitative simulation approaches are unable to make use of many existing Modelica models. In the the rest of this paper, we discuss features of the Modelica language and practice that hinders the reuse of Modelica models by formal methods such as qualitative reasoning.

## 3 Modelica models for formal methods

Modelica's reuse and flexibility are central to its appeal among designers, engineers, and researchers. Unfortunately, these features create difficulties for applying formal methods. Some problematic features are: compiler interaction, artifacts for numeric simulation, unnecessary component model complexity, algorithms, sequential states, and incomplete models. For each issue, we will attempt to answer three questions:

- Why do designers use it?
- Why is this difficult for formal methods?
- What should be done to enable formal verification?

Before we discuss Modelica language issues, we discuss how Modelica compilers assist in our efforts to perform qualitative reasoning with Modelica models.

### 3.1 Using the compiler to facilitate other analyses

Modelica tools include a compiler that takes as input a Modelica model and through a sequence of processes produces executable code to perform numeric simulation. Access to intermediate results during the compilation process facilitates other analyses. Here we highlight three aspects of the compilation process we have found useful.

**Model construction language**

Modelica has a powerful model construction language including iteration and conditional declarations. For example, the `Damper` model (shown in Figure 3) includes a conditional heat port, which allows the same damper model to be used in systems that consider thermal connections and those that do not. Also, compilers perform a number of optimizations on the model including index reduction and removal of redundant variables. These optimizations are applicable to qualitative reasoning as well. Therefore, our approach uses an XML representation of the hybrid-DAE produced by OpenModelica [Parrotto *et al.*, 2010]. Furthermore, we encourage the ongoing efforts to standardize an XML representation of the compiled model across Modelica tools.

**System initialization**

System initialization is a well-known difficult problem, and Modelica provides a number of language constructs to direct the solver to the initial state [Mattsson *et al.*, 2002]. These include the use of the `:start` and `:fixed` keywords, initial equations and initial algorithms. Qualitative simulation also requires knowing the initial values of the system variables. Therefore, we use OpenModelica to solve the initial equation system for a consistent set of initial values from which to begin our analysis.

**Function inlining**

Many Modelica functions are merely mathematical relationships between input and output variables. Consider the `from_kmh` function in the Modelica Standard Library shown in Figure 4. Function inlining is performed by many Modelica compilers to replace calls to these functions by their equivalent equations. The problems with analyzing arbitrary functions will be discussed in Section 3.4. Therefore, having the compiler perform these optimizations assists in translating Modelica models for use in formal methods.

Figure 4: Function that converts km/h to m/s

```
function from_kmh;
  input NonSIunits.Velocity_kmh kmh ;
  output Velocity ms "metre per second value";
algorithm
  ms := kmh/3.6;
end from_kmh;
```

### 3.2 Artifacts of numeric simulation

Modelica is primarily a language for modeling and simulating mathematical models that evolve as a function of time. Consequently, there exist many constructs to assist with issues that arise in numeric simulation. While some are irrelevant for formal methods (e.g., `noEvent` and `smooth`), in this section, we discuss three that complicate formal methods analyses.

**Equality involving continuous time variables**

Modelica events occur at zero crossings. Therefore, it is not possible to have a condition testing equality of real valued variables, $s1$ and $s2$. Instead, the `Modelica.math.IsEqual` function from the Modelica Standard Library which is computing using Equation 1.

$$result := abs(s1 - s2) <= eps; \tag{1}$$

While Equation 1 can be translated directly for formal methods, it both needlessly adds complexity to formal analysis and may yield unexpected results. Intuitively, `IsEqual` is testing whether $s1$ and $s2$ are equal. If the Equation 1 is directly encoded, the formal analysis will have to consider 7 cases: (1) $s1 - s2$ is negative and more than $eps$ from 0, (2) $s1 - s2 = -eps$, (3) $s1 - s2$ lies between $-eps$ and 0, (4) $s1 - s2 = 0$, (5) $s1 - s2$ is positive and less than $eps$, (6) $s1 - s2 = eps$, (7) $s1 - s2 > eps$. In effect, it treats $eps$ as an important parameter the system. As a consequence the number of states needed to be analyzed grows exponentially in the number of `IsEqual`'s translated in this way. Finally, the formal analysis could

Figure 3: Damper model from the Modelica Standard Library includes a conditional heat port connection that is set during model instantiation

```
model Damper "Linear 1D translational damper"
  extends Translational.Interfaces.PartialCompliantWithRelativeStates;
  parameter SI.TranslationalDampingConstant d(final min = 0, start = 0) "Damping constant";
  extends Modelica.Thermal.HeatTransfer.Interfaces.PartialElementaryConditionalHeatPortWithoutT;
equation
  f = d * v_rel;
  lossPower = f * v_rel;
end Damper;
```

produce incorrect results because it will accept as possible states in which s1 is not equal to s2 which is clearly against modeler's intent. Because formal methods allow for equality between continuous-time variables, the best solution is to simply translate `IsEqual(s1,s2)` to $s1 == s2$.

**Smoothing functions**

Another piece of advice supplied by the model to the simulation engine concerns smoothing functions. For example, the function `Modelica.Fluid.Utilities.regStep` approximates a step function with a second order polynomial that is continuous and differentiable. Unless the transient behavior is the focus of the model, formal methods are more applicable to the idealized behaviors.

### 3.3 Unnecessary component model complexity

Modelers should be as concise and clear as possible when authoring models.

**Optional model parameters**

The inheritance features in Modelica make it easy to provide different variants of components that account for different behaviors. Therefore, in each model, every parameter should affect the behavior of the model. When this is not the case, the modeler has increased the complexity of the model unnecessarily. Consider the `w_small` parameter `PartialFriction` model. The default value of this parameter is $1e10$ and the comment directs the engineer to set this to a small value if particular discontinuities are expected. This absurdly high value is to prevent it from affecting the simulation. Making these two separate models that inherit from the same model would facilitate formal methods by considering the `w_small` parameter only when it is necessary. Otherwise formal analysis will have to needlessly analyze the distinction between $w_small$ and $w$.

**Component modes**

The evolutionary development of the Modelica language is apparent in the models of the Modelica Standard Library. For instance, many models use `Integer` variables to define a mode of operation for the model. However, these variables are typically unbounded, and often the default variable value of zero is not an applicable mode. Using enumerations would provide a definite set of modes of operation for these variables. However, even this is not sufficient. In some tool systems, such as OpenModelica, parameter variables of enumerated types are not required to be initialized to any particular value, and in that case they default to the integer default value of zero, which is an invalid integer value for that enumerated type! In general, the semantics of operating modes, and more specifically enumerations (and parameters), seems to need more work in Modelica.

### 3.4 Imperative Code

Modelica algorithms can be executed at two times: flattening and simulation. All of the former algorithms pose no difficulty to formal analysis as they are executed before the DAE is created. Imperative code embedded in the DAE, typically in functions, to be executed at run time presents a fundamental challenge. Imperative code is important to model designers because certain numerical computations are easier to express as algorithms as opposed to equations. The Modelica language is Turing-complete, so proving properties of arbitrary Modelica programs is as hard as proving properties of any program. And proving properties of programs is a challenging intellectual field all to its own. Formal methods cannot be expected to analyze such algorithms. For common functions, we have created qualitative equivalents. For example, interpolation tables are essential to modeling complex physical systems, and, therefore, we have created an analogous concept for qualitative reasoning. There is independent interest in the Modelica community in elimination of imperatives when possible. For example, function inlining converts some imperatives to constraints automatically [Papadopoulos *et al.*, 2012] .

We have applied our analysis technique to a wide variety of models. Too often we encounter needless imperatives. For example, consider:

```
Model Single_Clutch_Controller
  Output Real y;
  Input Integer u;
  parameter Integer num_gears = 5
  parameter
   Integer gear_nums[num_gears] = {-1,1,2,3,4}
  parameter Real engagement[num_gears]
     = {0.0, 1.0, 1.0, 0.0, 0.0};
algorithm
  y := 0.0;
  for i in 1:num_gears loop
      if u == gear_nums[i] then
        y := engagement[i];
      end if;
  end for;
end Single_Clutch_Controller;

model GBX_5_clutch_controller
   ...
```


DOI 10.3384/ECP14096205

```
   Modelica.Blocks.Interfaces.RealOutput clutch_1;
   Single_Clutch_Controller controller_c1
     (num_gears=num_gears,
      gear_nums=gear_nums,
      engagement=c1_eng);
...
equations
   connect(controller_c1.u, gear_selected);
   connect(controller_c1.y , clutch_1 );
   ...
end GBX_5_clutch_controller;
```

Given a desired gear, the controller selects the clutch to activate. The algorithm block simply looks up the array index of the desired gear and reads off the clutch needed to engage. The clutch can be modeled as a table:

| x | y |
|---|---|
| -1 | 0.0 |
| 1 | 1.0 |
| 2 | 1.0 |
| 3 | 0.0 |
| 4 | 0.0 |
| X | 0.0 |

Modelica has a table primitive (which begs the question why it wasn't used in this model). However, this particular table can be expressed as a Modelica equation:

```
y = if (x=1 or x=2) then 1.0 else 0.0;
```

To summarize, our approach to imperative code is:

- For widely used MSL functions such as interpolation we design them as primitives for the qualitative analysis. These function names appear in the XML DAE and thus can be treated as primitives.
- In limited cases, imperative code can automatically be translated to declarative code by function inlining.
- Key MSL models containing imperative code are being rewritten to be purely declarative (or use only known imperative primitives).
- User created functions and algorithms are currently not allowed. One unexplored possibility is the user must annotate the model specifying a piecewise linear approximation of the imperative code's behavior.

### 3.5 Sequential States

Many models include sources that iterate through a sequence of states (`Modelica.Blocks.Sources.SawTooth`) or components that exhibit delayed effects (`Modelica.Blocks.Logical.TriggeredTrapezoid`). The primary way this is handled in Modelica is by triggered Modelica events and setting a discrete-time real value representing the time at which the next state should change. Consider the variable $T$ in the `TriggeredTrapezoid` model shown in Figure 5.

These models are difficult to analyze due to the non-local effects of the setting of the discrete-time variable. We propose to rewrite them without explicitly referencing time. This is done by using the events to set the rate and then the delayed effects occur when one of the continuous variables reaches a limit.

### 3.6 Incomplete models

Another complicating issue is that model authors frequently build models until their needs are met. While these models produce the simulation results the modeler expects, other analyses may have trouble using them due to untyped variables and operating regions.

**Untyped variables**
Modelica allows modelers to give types to variables, but modelers frequently use `Real` instead of more specific types. The `Brake` model in MSL defines `mue0` as type `Real` instead of `CoefficientOfFriction`. Automated fault modeling techniques (e.g., FAME [de Kleer *et al.*, 2013]) construct better fault models if variables are typed correctly.

## 4 Related work

The differences between the modeling languages used by engineers (e.g., Modelica, C++) and those used by model checking tools (e.g., finite state automata) hinder the deployment of formal methods in the design process. Researchers have taken both top-down and bottom-up approaches to overcoming this hurdle. Carloni *et al.* exemplify the top-down approach by arguing for a *semantic-aware interchange format* to make formal methods applicable across languages [Carloni *et al.*, 2006]. As an alternative to attempting to unify all hybrid systems languages, bottom-up approaches define automatic translations between subsets of pairs of languages. For example, Lundvall *et al.* translate a portion of Modelica models into hybrid automata for verification [Lundvall *et al.*, 2004]. Our approach follows the bottom-up tradition, and the contribution of this paper is a discussion of five Modelica modeling practices that hinder the automated analyses of Modelica models by formal methods.

## 5 Reflections on modeling

Modelica makes some fundamental semantic choices which are at odds with the formal methods, qualitative reasoning and cyber-physical systems communities. For example, Modelica allows one event to cause another — that is, two instants immediately following one another. Also, formal methods typically model behavior by modes, guards, and constraints. Modelica's guards and modes exist at the system level as conditions, but are not directly accessible from the models. So the common-sense engineering notion of mode has to be expressed by extra boolean variables and conditions. This can lead to very counterintuitive (to an engineer) models (e.g., the transistor models in MSL). For the purposes of our research, we have to accept Modelica's semantics. We do not know yet what problems this will cause.

Our experiences using Modelica models for other analysis purposes motivates some reflections on good modeling practices. In the course of this research, we have had to study and analyze a great many Modelica models. Some models in the MSL are diamonds, other models are disasters. We often wish that the Modelica community employed more standardized modeling practice. Modelica is such a general language that a modeler can write incredibly stupid models. Maybe that is because it is very hard to write clean, concise models. To summarize we suggest the following modeling principles:

Figure 5: Portion of the `TriggeredTrapezoid` model highlighting the use of discrete-time variables to initialize the timing of state transitions.

```
block TriggeredTrapezoid "Triggered trapezoid generator"
  extends Modelica.Blocks.Icons.PartialBooleanBlock;
  ...
protected
  ...
  discrete Modelica.SIunits.Time T
     "Predicted time of output reaching endValue";
equation
  y = if time < T then
     endValue - (T - time) * rate else
  endValue;
  when {initial(),u,not u} then
    ...
    T = if u and not rising > 0 or not u
              and not falling > 0
              or not abs(amplitude) > 0
              or initial() then
            time else
            time + (endValue - pre(y)) / rate;
  end when;
end TriggeredTrapezoid;
```

Figure 6: Portion of the `TriggeredTrapezoidPARC` model that explicitly states the conditions for state transitions.

```
block TriggeredTrapezoidPARC
  ...
protected
  Real rate;
equation
  when {initial(),u,not u,y>offset+amplitude,y<offset} then
   rate = ...
  end when;
  der(y) = rate;
end TriggeredTrapezoid;
```

1. Models should be easy to understand for both machines and people.
2. Models should be as declarative as possible, even when it's simpler to write imperative code. Such models are easier to understand by both people and machines.
3. Any "advice" to the compiler, especially dealing with small epsilon quantities in models should be done in standardized ways, such as `IsEquals`.
4. All variables should be declared by their physical type.
5. Models should have `assert` statements describing their range of applicability.

## 6 Acknowledgments

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

## References

[Carloni *et al.*, 2006] Luca P Carloni, Roberto Passerone, and Alessandro Pinto. *Languages and tools for hybrid systems design*, volume 1. Now Pub, 2006.

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

[de Kleer *et al.*, 2013] Johan de Kleer, Bill Janssen, Daniel G. Bobrow, Tolga Kurtoglu, Bhaskar Saha, Nicholas R. Moore, and Saravan Sutharshana. Fault augmented modelica models. In *24th International Workshop on Principles of Diagnosis*, pages 71–78, Jerusalem, Israel, 2013.

[Everett, 1999] John Otis Everett. Topological inference of teleology: Deriving function from structure via evidential reasoning 6. *Artificial Intelligence*, 113:149–202, 1999.

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

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

[Klenk *et al.*, 2012] Matthew Klenk, Johan de Kleer, Daniel G. Bobrow, Sungwook Yoon, John Hanley, and Bill Janssen. Guiding and verifying early design using qualitative simulation. In *Proceedings of the ASME 2012 IDETC and CIE*, Chicago, IL, 2012.

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

[Lundvall *et al.*, 2004] Håkan Lundvall, Peter Bunus, and Peter Fritzson. Towards automatic generation of model checkable code from modelica. 2004.

[Mattsson *et al.*, 2002] Sven Erik Mattsson, Hilding Elmqvist, Martin Otter, and Hans Olsson. Initialization of hybrid differential-algebraic equations in modelica 2.0. In *2nd Inter. Modelica Conference 2002*, pages 9–15, 2002.

[Papadopoulos *et al.*, 2012] Alessandro V Papadopoulos, Martina Maggio, Francesco Casella, Johan Åkesson, and AB Modelon. Function inlining in modelica models. In *7th Vienna Conference on Mathematical Modelling*, 2012.

[Parrotto *et al.*, 2010] Roberto Parrotto, Johan Åkesson, and Francesco Casella. An xml representation of dae systems obtained from continuous-time modelica models. In *EOOLT*, pages 91–98, 2010.

[Snooke and Price, 2012] Neal Snooke and Chris Price. Automated {FMEA} based diagnostic symptom generation. *Advanced Engineering Informatics*, 26(4):870 – 888, 2012.

[Struss and Price, 2004] Peter Struss and Chris Price. Model-based systems in the automotive industry. *AI Magazine*, 24(4):17–34, 2004.

[Tiwari, 2012] Ashish Tiwari. Hybridsal relational abstracter. In *CAV*, pages 725–731, 2012.

[Weld and de Kleer, 1989] D.S. Weld and J. de Kleer. *Readings in Qualitative Reasoning about Physical Systems*. Morgan Kaufmann, 1989.

[Wetzel and Forbus, 2009] Jon Wetzel and Ken Forbus. Automated critique of sketched mechanisms. *Proceedings of the 21st Innovative Applications of Artificial Intelligence Conference, Pasadena, CA*, 2009.