Higher-Order Non-Causal Modelling and Simulation of Structurally Dynamic Systems

George Giorgidze Henrik Nilsson

Functional Programming Laboratory
School of Computer Science
University of Nottingham
United Kingdom

{ggg,nhn}@cs.nott.ac.uk

Abstract

This paper explores a novel approach to the implementation of non-causal modelling and simulation languages supporting highly structurally dynamic systems. One reason the support for structural dynamics is limited in present mainstream non-causal modelling and simulation languages is that they are designed and implemented on the assumption that symbolic processing of models and ultimately compilation of simulation code takes place prior to simulation. We seek to lift that restriction, without sacrificing efficiency, by exploiting just-in-time (JIT) compilation to allow new simulation code, reflecting structural changes, to be generated as the simulation progresses. Our work is carried out in a framework called Functional Hybrid Modelling that supports *higher-order modelling*, as higher-order modelling lends itself naturally to expressing structural dynamism. However, the central ideas of the paper should be of general interest in the area of structural dynamism. The paper provides an in-depth description of the implementation techniques we have developed as well as a performance evaluation.

Keywords: Non-causal Modelling and Simulation, Structurally Dynamic Systems, Functional Programming, Just-In-Time Compilation, Symbolic/Numerical Methods

1 Introduction

When developing dynamic models of physical systems, it is often desirable to model major changes in system behaviour by changing the *differential algebraic equations* (DAEs) that describe the dynamics of the system. These major changes can be due to the modelled system itself exhibiting structural changes, due to a need to change to simplified models of parts of a system for periods of time, and so on [12]. Models whose equational description change over time are called *structurally dynamic*, and each structural configuration is known as a *mode* of operation. Structurally dynamic systems are an example of the more general notion of *hybrid* systems, systems that exhibit both continuous and discrete behaviour.

Unfortunately, the support offered by current modelling languages for expressing structurally dynamic systems (as well as hybrid systems in general) is somewhat limited [13, 20, 22]. This is true in particular for *non-causal* modelling languages, which is the class of modelling languages with which we are concerned in this paper. There are a number of reasons for this limited support, many of them related to the technical difficulties of simulating structurally dynamic models, such as identifying suitable state variables for different modes and proper transfer of the state between modes [12, 13].

However, there is also one less fundamental reason, namely the common assumption that most or all processing to put a model into a form suitable for simulation will take place *prior* to simulation [18, 21]. By enforcing this assumption in the *design* of a modelling language, its *implementation* can be simplified as there is no need for simulation-time support for handling structural changes. For instance, a compiler can typically generate static simulation code (often just sequences of assignment statements) with little or no need for dynamic memory management. This results in good performance. But the limitations are also obvious: for example, the number of modes must be modest as, in general, separate code must be generated for each mode. This rules out supporting *highly* structurally dynamic systems: systems where the number of modes is too large to make explicit enumeration feasible, or even a priori unbounded.

There are a number of efforts to design and implement modelling and simulation languages with improved support for structural dynamics. Examples include HYBRSIM [14], MOSILAB [19], and Sol [22]. Of these, Sol is likely the most flexible. However, thus far, implementations have either been interpreted (HYBRSIM and Sol), or the language has been restricted so as to limit the number of modes to make it feasible to compile code for all modes prior to simulation (MOSILAB).

DOI: 10.3384/ecp09430137

This paper contributes towards the design and implementation of modelling and simulation languages by demonstrating support *both* for modelling of highly structurally dynamic systems *and* for compilation of simulation code for efficiency. We present a prototype implementation of a non-causal language allowing arbitrary structural changes during simulation. Central to this capability, and the focus of this paper, is that the equations that describe the current operating mode are compiled into simulation code at each structural change using a code-generation framework supporting just-in-time (JIT) compilation: the Low Level Virtual Machine (LLVM) [7]. We describe the compilation process as well as the necessary supporting run-time machinery, and we provide small but detailed benchmarks that demonstrate that the generated simulation code is fairly efficient and that the overhead of the processing of structural changes is not unreasonable, particularly not for an early prototype. As far as we know, this is the first time JIT compilation has been used for dynamic compilation of simulation code to enable efficient simulation of highly structurally dynamic models in the context of non-causal modelling. The implementation is available on-line[1] under the open source BSD license.

This work has been carried out in the context of our research on Functional Hybrid Modelling (FHM) [17, 18], a novel approach to purely declarative languages for non-causal, hybrid modelling and simulation. A central aspect of FHM is that models are *first-class entities*. This means they can be manipulated programmatically (past as arguments to functions, returned as results of functions, etc.), just like any other type of value such as integers or Booleans. This is called *higher-order* modelling [3].[2]

Higher-order modelling, with just a minimum of additional language constructs, lends itself very well to *expressing* highly structurally dynamic systems: all that is needed is the means to allow new model fragments to be computed not only before simulation starts, but also *during* simulation, at events, and to be integrated into the simulated system at those points. This is the approach taken by FHM. Indeed, the ease by which higher-order modelling can express structural dynamics was partly what motivated our research into FHM in the first place [17].

However, at its core, this paper is concerned with techniques for *implementing* languages supporting modelling of highly structurally dynamic systems, and we would thus like to emphasise that the ideas and results presented in this paper are not limited to the setting of FHM, but are, on the whole, applicable to modelling and simulation languages supporting structurally dynamic systems in general. We would also like to reiterate that the focus of this paper is squarely on the mechanics of integrating dynamic code generation into the implementation of a non-causal modelling language: many of the other technical problems briefly mentioned above remain to be solved.

[1] `http://cs.nott.ac.uk/~ggg/`

[2] As it is reminiscent of *higher-order functions*. A function is higher-order if some of its arguments or result is function-valued. Not to be confused with other meanings of *higher-order*.

The rest of this paper is organised as follows. Section 2 provides background on FHM and LLVM. FHM is introduced by means of an example that is also used in the remainder of the paper, so we recommend that all readers, even if already familiar with FHM, take at least a quick look at this section. Section 3 explains how our language is implemented, with a particular emphasis on the methods we use to support highly structurally dynamic systems. The performance of our prototype implementation is evaluated in Section 4. Related work is discussed in Section 5. Finally, Section 6 considers future work and conclusions are given in Section 7.

2 Background

2.1 Functional Hybrid Modelling

In the following, we give a brief overview of Functional Hybrid Modelling (FHM) to explain the notation used in the rest of the paper and to provide some general background. In particular, we introduce Hydra, the FHM language we are currently working on. For details, see earlier papers on FHM [17, 18]. However, we again remind the reader that in the present context, FHM and Hydra should mostly be seen as a particular syntax that is convenient for expressing structurally dynamic systems; neither is central to the contributions of this paper.

With FHM, we seek to develop a small but expressive, purely declarative, non-causal and hybrid modelling language. A key motivation is to develop simple and clear semantical foundations for this class of modelling languages, with a view to paving the way for improvements such as more flexible support for hybrid modelling and type systems exploiting domain knowledge in new ways [15].

Our hypothesis is that the aims of FHM can be realised by identifying the core semantical concepts of non-causal and hybrid modelling and embedding these as first-class entities in a declarative host language. This achieves a separation of concerns that we believe is both sound and expedient, as it highlights similarities and differences with other classes of languages and allows us to focus our research on what is specific to non-causal, hybrid modelling languages.

2.2 Signal Relations

FHM was inspired by Functional Reactive Programming (FRP) [4] and then in particular Yampa [16]. FRP is an approach to reactive programming that in many ways can be seen as *causal* modelling. It is realised by enriching a functional language with a first-class notion of functions operating on signals, *signal functions*, where a signal is a time-varying value. In other words, signal functions are very much like "blocks" in a causal modelling language like Simulink, except having first-class status, which means that ordinary functions can operate on signal functions achieving what in many modelling languages would be considered meta-modelling capabilities. In particular, new signal functions can be computed as a system is running and

then "switched in" to become part of that system, allowing highly structurally dynamic systems to be modelled [16].

What distinguishes non-causal from causal modelling languages is that models are described in terms of undirected equations over time-varying entities. In other words, *relations* on signals as opposed to functions on signals. The essence of FHM is thus the addition of first-class relations on signals, *signal relations*, to a functional language.

There are consequently *two levels* to FHM: the *functional level*, concerned with defining ordinary functions operating on time-invariant values, and the *signal level*, concerned with the definition of relations between signals (time-varying values), and, *indirectly*, the definition of the signals themselves as solutions satisfying the constraints imposed by the signal relations. The definitions at the signal level may freely refer to entities defined at the functional level, which is crucial in the following. Also, defined signal relations are ordinary time-invariant values at the functional level (first-class entities). Signals, on the other hand, are *not* first-class entities at the functional level. However, as discussed in the following, *instantaneous values* of signals can be propagated back to the functional level allowing e.g. the future system structure to depend on signal values at discrete points in time.

For those familiar with languages like Modelica, note that a signal relation has many similarities to a class: fundamentally a signal relation is just an encapsulated set of equations that constrain a number of signal variables. Also like a class, signal relations can be *instantiated*, creating copies of the encapsulated equations imposing their constraints on some variables in the current context. This is called signal relation *application*. Unlike Modelica, there is no class hierarchy and thus no inheritance, so the *only* way to reuse equations is by signal relation application. On the other hand, signal relations are first class entities, giving a lot of additional expressive power, whereas classes are not.

2.3 Hydra by Example: The Breaking Pendulum

Let us illustrate the key aspects of FHM and Hydra through an example. As this paper is concerned with structural dynamism, we chose a variation of a breaking pendulum [11, pp. 31–33] as it is representative yet small. The pendulum is modelled as a point mass m at the end of a rigid, massless rod, subject to gravity $m\vec{g}$; see Figure 1. Additionally, the rod could break at some point in time, causing the mass to fall freely. The breaking of the pendulum causes a significant change in the structure of the system, so much that languages like Modelica do not support non-causal simulation of a pendulum that breaks during simulation [18].

We start by modelling a free-falling body in Hydra. Let us begin by defining some type abbreviations and constants for convenience. Hydra is implemented as an *embedding* in Haskell [5], the *host language*. The functional level discussed above is thus provided by Haskell, saving us the work of implementing a functional language from scratch. Type abbreviations and constants are ordinary time-invariant definitions and defined at the functional level; i.e., directly in Haskell. We define the position and velocity to be pairs of reals, and the state of a body in motion to be given by its position and velocity. An entity of type $Body$ is thus an aggregate of four real-valued fields. We also define the gravitational acceleration g to a suitable value:

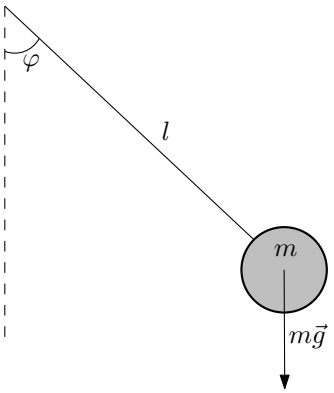

Figure 1: A pendulum subject to gravity.

$$\begin{aligned}
&\textbf{type } Position = (Double, Double)\\
&\textbf{type } Velocity = (Double, Double)\\
&\textbf{type } Body \quad = (Position, Velocity)\\
&g :: Double\\
&g = 9.81
\end{aligned}$$

A model of a free-falling body is a signal relation parametrised on the initial state of the body. In Hydra, exploiting that signal-relations are first-class entities, a parametrised signal relation is just a function that *computes* the appropriate signal relation from the parameters and returns this computed signal relation as its result. This is thus an example of higher-order modelling. In our case, we obtain a function that computes a signal relation constraining a variable of type $Body$ (or rather, its four real-valued fields) given an initial value of the state, also of type $Body$ ($\rightarrow$ is the type former for functions in Haskell):

$$freeFall :: Body \rightarrow SR\ Body$$

The function *freeFall* is defined by *pattern matching* on its one argument of type $Body$, binding (functional-level) variables $x0$, $y0$, $vx0$, and $vy0$ to the initial position and velocity:

$$freeFall\ ((x0, y0), (vx0, vy0)) = \ldots$$

The bound variables scope over the body of the function, here indicated by an ellipsis.

Note that the Haskell syntax for function application is simply juxtapositioning; e.g., $f\ 1$ denotes the application of the function f to the argument 1, $g\ 2\ 3$ the application of the function g to the two arguments 2 and 3, and so on. Parentheses are not part of the syntax of function application as such, although they are used for grouping and, as here, to denote tuples. The syntax of function definition mirrors the

syntax of function application. The function *freeFall* is thus syntactically a function of a single argument: a pair of pairs of reals.

The body of *freeFall* defines the parametrised signal relation. To do this, we need to work at the signal level. The Hydra embedding is achieved through *quasiquoting*, a Haskell extension provided by the Glasgow Haskell Compiler (GHC) [10]. Quasiquoting allows custom syntax to be realised by arranging for source text delimited by quasiquotes to be passed to a custom-written function that parses the text and translates it into abstract syntax of the *host* language. In GHC, this is done prior to type checking. Quasiquoting is thus rather similar to what can be achieved through a pre-processor, except more tightly integrated with the compiler and less work to implement. The Hydra opening quasiquote is $[\$hydra|$, and the closing quote is $|]$. Between them, we have signal-level definitions expressed in our custom syntax. The complete definition of *freeFall* is as follows:

$$
\begin{array}{l}
freeFall\ ((x0, y0), (vx0, vy0)) = [\$hydra| \\
\quad \textbf{sigrel}\ ((x, y), (vx, vy))\ \textbf{where} \\
\qquad \textbf{init}\ (x, y) = (\$x0\$, \$y0\$) \\
\qquad \textbf{init}\ (vx, vy) = (\$vx0\$, \$vy0\$) \\
\qquad (der\ x, der\ y) = (vx, vy) \\
\qquad (der\ vx, der\ vy) = (0, -\$\ g\$) \\
|]
\end{array}
$$

The keyword **sigrel** starts the definition of a signal relation. It is followed by a pattern that introduces *signal variables* giving local names to the signals that are going to be constrained by the signal relation. This pattern thus specifies the *interface* of a signal relation, similarly to how function arguments specify the interface of a function. After the keyword **where** follow the equations that define the relation. These equations may introduce additional signal variables as needed. Equations marked by the keyword **init** are initialisation equations used to specify initial conditions.

To refer to *functional-level* entities at the signal level, inside the quasiquotes, such references need to be *antiquoted* by enclosing them between $-signs. Arbitrary Haskell expressions, using any *functional-level* variable in scope outside the quasiquoted block, are allowed between the antiquotes. However, none of the *signal-level* variables are in scope in antiquoted code. The abstract syntax for each antiquoted Haskell expressions is then spliced in at the point of the antiquote. For example, note how the functional-level parameters defining the initial position and velocity ($x0$, $y0$, $vx0$, $vy0$) are referenced in the initialisation equations. A signal relation may thus depend on functional-level values, thus making it parametrised. As we will see, these values can be *any* kind of functional-level values, including signal relations, thus achieving higher-order modelling without further ado.

Of course, the quasiquotes and antiquotes are just an artifact of our specific approach to implementing Hydra. A less "noisy" syntax would certainly be possible (with some implementation effort). However, the explicit quoting makes the distinction between the functional level and the signal level manifest, which is helpful at least for explanatory purposes.

Readers who are familiar with Modelica might find it illuminating to compare the Hydra model above with a Modelica version. To facilitate comparison, we have kept the Modelica version as close as possible to the Hydra version, rather than trying to provide the "most natural" Modelica model:

```
model FreeFall
  parameter Real x0, y0, vx0, vy0;
  Real x(start=x0), y(start=y0);
  Real vx(start=vx0), vy(start=vy0);
equation
  der(x) = vx;
  der(y) = vy;
  der(vx) = 0;
  der(vy) = -g;
end FreeFall;
```

Here, `g` is assumed to be a constant defined elsewhere. Note how Modelica parameters, that denote entities that remain constant during continuous integration, correspond to functional-level variables in Hydra, while Modelica variables that change during continuous integration correspond to signal-level Hydra variables. A difference, though, is that Modelica parameters can only change at the very start of a simulation, while Hydra functional-level variables can change at every event occurrence.

In a completely analogous way to the free-falling mass, we define a parametrised signal relation that models the pendulum in its unbroken mode. The parameters are the length of the rod l and the initial angle of deviation $phi0$:

$$
\begin{array}{l}
pendulum :: Double \rightarrow Double \rightarrow SR\ Body \\
pendulum\ l\ phi0 = [\$hydra| \\
\quad \textbf{sigrel}\ ((x, y), (vx, vy))\ \textbf{where} \\
\qquad \textbf{init}\ phi = \$phi0\ \$ \\
\qquad \textbf{init}\ der\ phi = 0 \\
\qquad \textbf{init}\ vx = 0 \\
\qquad \textbf{init}\ vy = 0 \\
\qquad (x, y) = (\$l\ \$ * sin\ phi, -\$\ l\ \$ * cos\ phi) \\
\qquad (vx, vy) = (der\ x, der\ y) \\
\qquad der\ (der\ phi) + (\$g\ \$\ /\ \$\ l\$) * sin\ phi = 0 \\
|]
\end{array}
$$

Again, for comparison, we give a Modelica version:

```
model Pendulum
  parameter Real l, phi0;
  Real x, y, vx, vy;
  Real phi(start=phi0), phid;
equation
  x = l * sin(phi);
  y = -l * cos(phi);
  vx = der(x);
  vy = der(y);
  phid = der(phi);
  der(phid) + (g/l) * sin(phi) = 0;
end Pendulum;
```

We proceed to extend the above definition into a signal relation that also provides an *event signal* defining when the pendulum is to break. An event signal is a signal that is only defined at discrete points in time, *events*. In this case, an event is generated at a specified point in time, and the value of the event signal is the state (position and velocity) of the pendulum at that point. Note how the new signal relation is defined by extending the previous one. The parametrised signal relation $pendulum$ is applied to the length of the pendulum and the initial angle of deviation at the *functional level* (within antiquotes), thus computing a signal relation. This relation is then applied at the signal level, through *signal relation application* (the operator $\diamond$), instantiating the equations of $pendulum$ in the context of $breakingPendulum$ and thus effectively extending the definition of $pendulum$:

$$
\begin{aligned}
&breakingPendulum :: Double \rightarrow Double \rightarrow Double \\
&\qquad\qquad \rightarrow SR\ (Body, E\ Body) \\
&breakingPendulum\ t\ l\ phi0 = [\$hydra| \\
&\quad \mathbf{sigrel}\ (((x, y), (vx, vy)), \\
&\qquad\quad event\ e@((\_, \_), (\_, \_)))\ \mathbf{where} \\
&\qquad \$\ pendulum\ l\ phi0\ \$\ \diamond ((x, y), (vx, vy)) \\
&\qquad event\ e = ((x, y), (vx, vy))\ when\ time = \$t\ \$ \\
&|]
\end{aligned}
$$

The process of unfolding signal relation applications is called *flattening* and is in many ways similar to the transformation of hierarchical models in languages like Modelica into a flat system of equations, a process usually also referred to as flattening. Unfolding signal relation application in Hydra is straightforward: the actual arguments (signal-valued expressions) to the right of the signal relation application operator $\diamond$ are simply substituted for the corresponding formal arguments (signal variables) in the body of the signal relation to the left of $\diamond$. The only real issue is that *name capture*, accidental clashes of variable names[3], must be avoided by an appropriate renaming strategy.

Finally, we simulate the actual breaking of the pendulum by switching from the pendulum equations to the free fall equations at the point where the event is generated. This is accomplished by the $switch$-combinator[4] with the following type signature:

$$switch :: SR\ (a, E\ b) \rightarrow (b \rightarrow SR\ a) \rightarrow SR\ a$$

The as and bs in the above type signature are *polymorphic type variables* that can be instantiated to *any* specific type. For example, a would be a pair of reals for a binary signal relation.

The $switch$-combinator is another example of higher-order modelling: it takes two signal relations as arguments (one plain signal relation and one parametrised signal relation; i.e. a function returning a signal relation) and combines them into a new signal relation that initially imposes constraints according to the first relation, and after the switch according to the second relation. Again, this is in many ways just syntax that happens to fit well in our FHM setting. One could envision alternative ways of expressing switching from one set of equations to another, such as conditionals where each branch give one set of equations.

In more detail, the $switch$-combinator expects an event signal output from its first signal relation argument. The first time this event signal is defined, the $switch$ combinator *applies* its second argument (the parameterised signal relation) to the value of the event signal at this point, computing a new signal relation of the same type as the first signal relation argument, and then replacing the equations from the first signal relation with those of the newly computed signal relation:

$$
\begin{aligned}
&mainSR :: SR\ Body \\
&mainSR = switch\ (breakingPendulum\ 10\ 1\ (pi\ /\ 4)) \\
&\qquad\qquad freeFall
\end{aligned}
$$

Note that the switching event carries the state of the pendulum at the breaking point as a value of type $Body$. This value is passed to $freeFall$, resulting in a model of the free-falling mass that is initialised in a way that ensures that the position and velocity of the mass become continuous signals.

2.4 The Low Level Virtual Machine

The Low Level Virtual Machine (LLVM) [7] is a language-independent, portable, optimising, compiler back-end. As its name suggests, it provides a compiler with a virtual machine target and takes care of translating the LLVM code into code for any specific, concrete, architecture supported by the LLVM. There are a number of backends with capabilities similar to LLVM that we could have used instead. However, LLVM is rather typical, so the following discussion, while centred around LLVM, is mostly relevant also for other similar backends.

The LLVM has been very carefully engineered to on the one hand be sufficiently high-level to be truly portable across different architectures, shielding the the compiler from the low-level details of the ultimate target. In that sense, it is a principled alternative to using a language like C as a compiler target for portability. Also, this makes it possible to support generic, target-independent optimisations at the LLVM level, saving the compiler writer from the burden of implementing many standard optimisations from scratch. On the other hand, LLVM has also been engineered to be sufficiently low-level to not get in the way of generating high-performance code, and to not make any assumptions about the source language. Taken together, this means that the LLVM it is an ideal target for compilers for a wide range of different languages.

The LLVM is thus rather unlike what probably is the most common virtual machine, the Java Virtual Machine (JVM). The JVM is very Java-centric, making it a poor fit for any language which isn't similar to Java. Also the JVM is fundamentally a byte-code interpreter, which incurs performance

[3] For example, a local variable in a signal relation body having the same name as a variable in one of the actual arguments.

[4] A *combinator* is a function without free variables. Functions whose main purpose is to combine functions into new functions, are often referred to as combinators to emphasise this purpose.

penalties, even if just-in-time (JIT) compilation often is employed to speed up the interpretation. LLVM code, on the other hand, is designed to be compiled into code for the ultimate target architecture.

That said, the LLVM, like the JVM, does support adding in new code dynamically to running applications. The LLVM achieves this through JIT compilation. An application that needs to invoke code that is generated dynamically is linked with the LLVM JIT compiler. The code generator of the JIT compiler is the same as in LLVM static compiler [9]. The LLVM code generator has been shown to outperform the code generator of GNU Compilers Collection (GCC) in a number of benchmarks, both in speed of code generation and execution of generated code [8].

The JIT compiler can be invoked through the provided API whenever a piece of new code needs compiling. The JIT compiler will return a handle to the generated code. This can then be called, just like any statically compiled function, and equally efficient. It is also possible to dispose of dynamically generated code that is no longer needed. The LLVM JIT capabilities turned out to be a very good fit for supporting structurally dynamic simulation, as will be discussed in the following.

3 Simulation

In this section we describe how simulation is performed in Hydra. The process is illustrated in Figure 2 and is conceptually divided into four stages. In the first stage, a signal relation is flattened and subsequently transformed into a mathematical representation suitable for numerical simulation. In the second stage, this representation is just-in-time compiled into efficient machine code. In the third stage, the compiled code is passed to a numerical solver that simulates the system until the end of simulation or an event occurrence. In the fourth stage, in the case of an event occurrence, the event is analysed, a corresponding new signal relation is computed and the process is repeated from the first stage. In the following, each stage is described in more detail.

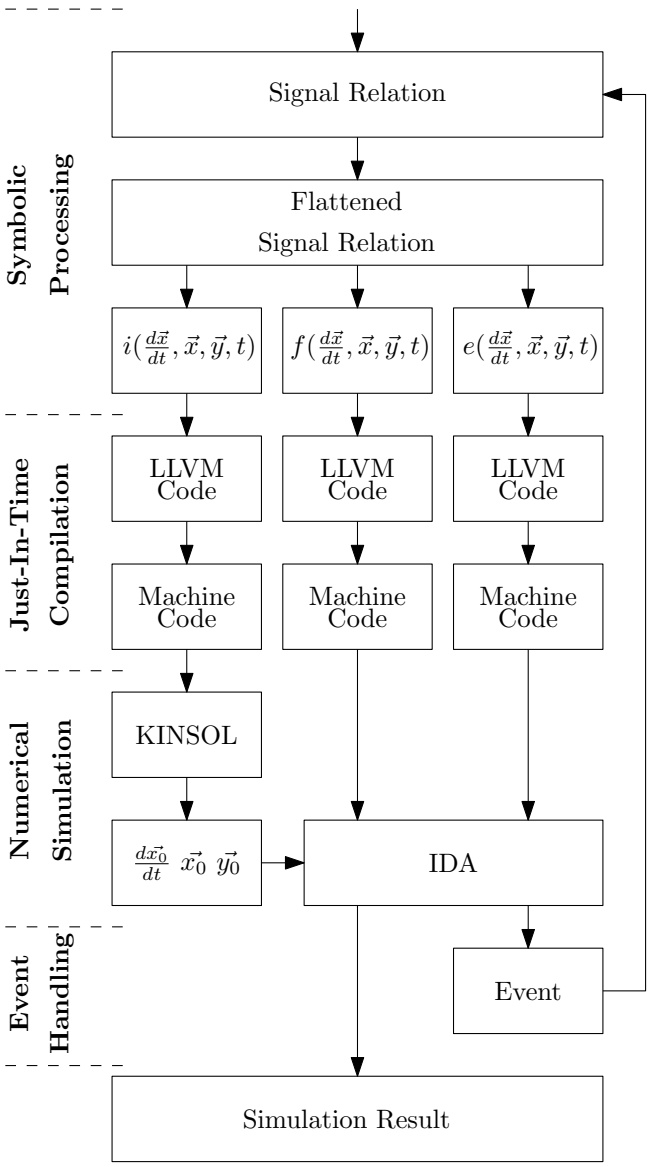

Figure 2: Simulation run-time system of Hydra

3.1 Symbolic Processing

As a first step, all signal variables are renamed to give them distinct names. This is to simplify the process of *flattening*, signal relation application unfolding (see section 2.3). All event variables are also given distinct names to allow the event handler to identify a corresponding event variable in the original unflattened signal relation at the moment of an event occurrence (see section 3.4). Having carried out this preparatory renaming step, all signal relation applications are unfolded until the signal relation is completely flattened.

Further symbolic processing is then performed to transform the flattened signal relation into a form that is suitable for numerical simulation. In particular, derivatives of compound signal expressions are computed symbolically. In the case of higher-order derivatives, extra variables and equations are introduced to ensure that all derivatives in the flattened system are first order. While the numerical solver used in the current implementation handles higher-index systems of equations, it is desirable to perform index reduction symbolically at this stage as well [1, 23]. Hydra does not yet do this, but we intend to implement symbolic index reduction in the future.

Finally, the following equations are generated at the end of the stage of symbolic processing:

$$i(\frac{d\vec{x}}{dt}, \vec{x}, \vec{y}, t) = 0, t = t_0 \quad (1)$$

$$f(\frac{d\vec{x}}{dt}, \vec{x}, \vec{y}, t) = 0 \quad (2)$$

$$e(\frac{d\vec{x}}{dt}, \vec{x}, \vec{y}, t) = 0 \quad (3)$$

Here, $\vec{x}$ is a vector of differential variables, $\vec{y}$ is a vector of algebraic variables, t is time, and t_0 is the starting time for the current set of equations. Equation 1 determines the initial conditions for Equation 2 (i.e., the values of $\frac{d\vec{x}}{dt}$,$\vec{x}$

and $\vec{y}$ at time t_0). Equation 2 is the main DAE of the system that needs to be integrated in time starting from the initial conditions. Equation 3 specifies the event conditions.

3.2 Just-in-time Compilation

The generated equations are implicitly formulated ones: the mathematical representation of non-causal signal relations. In general, it is not possible to transform these implicit equations into explicit ones; i.e., to completely causalise them [1]. Consequently, a system of implicit equations needs to be solved at the start of the simulation of each structural configuration mode and at every integration step. For example, a numerical solution of an implicitly formulated DAE (Equation 2) involves execution of the function f, a number of times (sometimes hundreds or more at each integration step), with varying arguments, until it converges to zero. The number of executions of f depends on various factors including the required precision, the initial guess, the degree of non-linearity of the DAE, etc.

To enable efficient simulation, it is important to compile the functions i, f, and e to a representation that can be executed efficiently. In addition, as Hydra allows the equations to change completely during simulation, it follows that compilation cannot be performed only before the simulation starts, but has to be performed during the simulation as well, whenever the equations change.

The simulation run-time system of Hydra supports JIT machine code generation using the compiler infrastructure provided by LLVM. The functions i, f and e are compiled into LLVM instructions that in turn are compiled by the LLVM JIT compiler into machine code for the processor architecture the simulation is running on. As noted above, the process of JIT compilation is triggered by the simulation run-time system at every discrete event that changes the equations of the system. The generated machine code is then passed to the numerical solver.

3.3 Numerical Simulation

The numerical suite used in the current implementation of Hydra is called SUNDIALS [6]. The following components of SUNDIALS are used:

- KINSOL: nonlinear algebraic equation systems solver
- IDA: differential algebraic equation systems solver

The code for the function i is passed to KINSOL that numerically solves the system and returns initial values (at time t_0) of $\frac{d\vec{x}}{dt}$,$\vec{x}$ and $\vec{y}$. These vectors together with the code for the functions f and e are passed to IDA that proceeds to solve the DAE by numerical integration. This continues until either the simulation is complete or until one of the events defined by the function e occurs. Precise event detection facilities are provided by IDA.

3.4 Event Handling

At the moment of an event occurrence, the numerical simulator terminates and presents the following information to an event handler:

- Name of the event signal variable for which an event occurrence has been detected
- Time t_e of the event occurrence
- Instantaneous values of the signal variables (i.e., values of $\frac{d\vec{x}}{dt}$, $\vec{x}$ and $\vec{y}$ at time t_e)

In the case of the breaking pendulum model, the name of the detected event signal variable is e. In addition, $t_e = 10$, $x \approx 0.21$, $y \approx -0.98$, $vx \approx 2.25$ and $vy \approx 0.49$. Here, x, y, vx and vy are the signal variables that are constrained by the $pendulum$ signal relation.

Next, the event handler traverses the original unflattened signal relation and finds the event value expression (i.e., a signal-level expression) that corresponds to the aforementioned event signal variable. In the case of the breaking pendulum model, the expression is $((x, y), (vx, vy))$. This expression is evaluated by substituting the instantaneous values of the corresponding signals for the variables. In the case of the breaking pendulum model, the computed value is $((0.21, -0.98), (2.25, 0.49))$. However, note that this now is a functional-level expression. This is the only place in Hydra where instantaneous values of signals are passed back to the functional level.

As a final step, the event handler applies the second argument of the switch combinator (i.e., the function to compute the new signal relation to switch into) to the functional-level event value. In the case of the breaking pendulum model, the function $freeFall$ is applied to $((0.21, -0.98), (2.25, 0.49))$.

The result of this application is a new signal relation. The simulation process continues from the first stage of symbolic processing and onwards by discarding the old signal relation and simulating the new one. In the case of the breaking pendulum model, the $pendulum$ signal relation is discarded together with the machine code that was generated for it by the LLVM JIT compiler.

In the current implementation, the new signal relation is flattened and new equations generated without reusing old ones from previous modes. In other words, events are not treated locally. In addition, the state of the whole system needs to be transferred for global and explicit reinitialisation of the entire system at every event using a *top level* switch, like in the breaking pendulum example. We hope to address these issues in the future: see Section 6.

4 Performance

In this section we provide an initial performance evaluation of the current prototype implementation of Hydra. We are mainly concerned with the overheads of mode switching (computing new structural configurations at events, symbolic processing of the equations, and JIT compilation) and

how this scales when the size of the models grow in order to establish the feasibility of our approach. The time spent on numerical simulation is of less interest at this point: as we are using standard numerical solvers, and as our model equations are compiled down to native code with efficiency on par with statically generated code (see section 2.4), this aspect of the overall performance should be roughly similar to what can be obtained from other compilation-based modelling and simulation language implementations. For this reason, and because other compilation-based, non-causal modelling and simulation language implementations do not carry out dynamic reconfiguration, we do not compare the performance to other simulation software. The results would not be very meaningful.

The evaluation setup is as follows. The numerical simulator integrates the system using variable-step, variable-order BDF (Backward Differentiation Formula) solver [1]. Absolute and relative tolerances for numerical solution are set to 10^{-6} and trajectories are printed out at every point where $t = 10^{-3} * k, k \in \mathbb{N}$. For static compilation and JIT compilation we use GHC 6.10.4 and LLVM 2.5 respectively. Simulations are performed on a 2.0 GHz x86-64 Intel® Core™2 CPU. However, presently, we do not exploit any parallelism, running everything on a single core.

Let us first consider the model of the breaking pendulum from Section 2.3. We simulate it over the time interval $t \in [0, 20]$, letting the pendulum break at $t = 10$. Table 1 shows the amount of time spent simulating each mode of the system, and within that how much time that is spent on each of the four conceptual simulation process stages (see Section 3). As can be seen, most time (80–90 %) is spent on numerical simulation, meaning the overheads of our dynamic code generation approach was small in this case. Also, in absolute terms, it can be seen that the amount of time spent on symbolic processing, JIT compilation, and event handling was small, just fractions of a second.

	Pendulum $t \in [0, 10)$		Free Fall $t \in [10, 20]$	
	CPU Time		CPU Time	
	s	%	s	%
Symbolic Processing	0.0001	0.2	0.0000	0.0
JIT Compilation	0.0110	18.0	0.0077	9.1
Numerical Simulation	0.0500	81.8	0.0767	90.9
Event Handling	0.0000	0.0	-	-
Total	0.0611	100.0	0.0844	100.0

Table 1: Time profile of the breaking pendulum simulation

However, the breaking pendulum example is obviously very small (just a handful of equations), and it only needs to be translated to simulation code twice: at simulation start and when the pendulum breaks. To get an idea of how the performance of the prototype implementation scales with an increasing number of equations, we constructed a hybrid model of an RLC circuit (i.e., a circuit consisting of resistors, inductors and capacitors) with dynamic structure. In the first mode the circuit contains 200 components, described by 1000 equations in total (5 equations for each component). Every time $t = 10 * k$, where $k \in \mathbb{N}$, the number of circuit components is increased by 200 (and thus the number of equations by 1000) by switching the additional components into the circuit.

	200 Components 1000 Equations $t \in [0, 10)$		400 Components 2000 Equations $t \in [10, 20)$		600 Components 3000 Equations $t \in [20, 30)$	
	CPU Time		CPU Time		CPU Time	
	s	%	s	%	s	%
Symbolic Processing	0.063	0.6	0.147	0.6	0.236	0.5
JIT Compilation	1.057	10.2	2.120	8.3	3.213	6.6
Numerical Simulation	9.273	89.2	23.228	91.1	45.140	92.9
Event Handling	0.004	0.0	0.006	0.0	0.008	0.0
Total	10.397	100.0	25.501	100.0	48.598	100.0

Table 2: Time profile of structurally dynamic RLC circuit simulation, part I

	800 Components 4000 Equations $t \in [30, 40)$		1000 Components 5000 Equations $t \in [40, 50)$		1200 Components 6000 Equations $t \in [50, 60]$	
	CPU Time		CPU Time		CPU Time	
	s	%	s	%	s	%
Symbolic Processing	0.328	0.4	0.439	0.4	0.534	0.3
JIT Compilation	4.506	4.9	5.660	5.1	6.840	4.3
Numerical Simulation	86.471	94.7	105.066	94.5	152.250	95.4
Event Handling	0.011	0.0	0.015	0.0	-	-
Total	91.317	100.0	111.179	100.0	159.624	100.0

Table 3: Time profile of structurally dynamic RLC circuit simulation, part II

Tables 2 and 3 show the amount of time spent in each mode of the system and in each conceptual stage of simulation of the structurally dynamic RLC circuit. In absolute terms, it is evident that the extra time spent on the mode switches becomes significant as the system grows. However, in relative terms, the overheads of our dynamic code generation approach remains low at about 10 % or less of the overall simulation time.

While JIT compilation remains the dominating part of the time spent at mode switches, Figure 3 demonstrates that the performance of the JIT compiler scales well. In particular, compilation time increases roughly linearly in the number of equations. In addition, it should be noted that the time spent on symbolic processing and event handling remains encouragingly modest (both in relative and absolute terms) and grows slowly as model complexity increases. There are also many opportunities for further performance improvements: see Section 6 for some possibilities.

Our approach offers new functionality in that it allows non-causal modelling and simulation of structurally dynamic systems that simply cannot be handled by static approaches. Thus, when evaluating the feasibility of our approach, one should weigh the overheads against the limitation and inconvenience of not being able to model such systems non-causally.

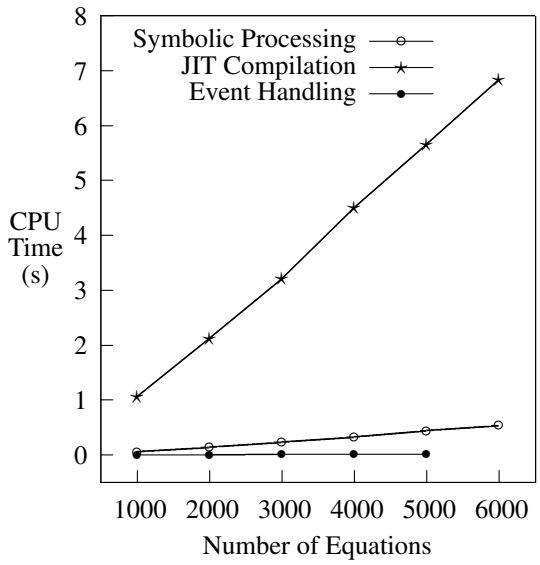

Figure 3: Plot demonstrating how CPU time spent on mode switches grows as number of equations increase in structurally dynamic RLC circuit simulation

5 Related Work

5.1 Sol

Sol is a Modelica-like language [21, 22]. It introduces language constructs that enable the description of systems where objects are dynamically created and deleted, thus aiming at supporting modelling of highly structurally dynamic systems. So far, the research emphasis has been on the design of the language itself along with support for incremental dynamic recausalisation and dynamic handling of structural singularities. An interpreter is used for simulation. The work on Sol is thus complementary to ours: techniques for dynamic compilation would be of interest in the context of Sol to enable it to target high-end simulation tasks; conversely, algorithms for incremental recausalisation is of interest to us to minimise the amount of work needed to regenerate simulation code after structural changes (see Section 6).

5.2 MOSILAB

MOSILAB is an extension of the Modelica language that supports the description of structural changes using object-oriented statecharts [19]. This enables modelling of structurally dynamic systems. It is a compiled implementation. However, the statechart approach implies that all structural modes must be explicitly specified in advance, meaning that MOSILAB does not support *highly* structurally dynamic systems. Even so, if the number of possible configurations is large (perhaps generated mechanically by meta-modelling), techniques like those we have investigated here might be of interest also in the implementation of MOSILAB.

5.3 Modelling Kernel Language

Broman [2, 3] is developing the Modelling Kernel Language (MKL) that is intended to be a core language for non-causal modelling languages such as Modelica. Broman takes a functional approach to non-causal modelling, similar to the FHM approach [17, 18]. One of the main goals of MKL is to provide a formal semantics of the core language. Currently, this semantics is based on an untyped, effectful λ-calculus.

Similarly to Hydra, MKL provides a λ-abstraction for defining functions and an abstraction similar to **sigrel** for defining non-causal models. Both functions and non-causal models are first-class entities in MKL, enabling higher-order, non-causal modelling. The similarity of the basic abstractions in Hydra and MKL leads to a similar style of modelling in both languages.

Thus far, the work on MKL has not specifically considered support for structural dynamics, meaning that its expressive power in that respect is similar to current mainstream, non-causal modelling and simulation languages like Modelica. However, given the similarities between MKL and FHM/Hydra, MKL should be a good setting for exploring support for structural dynamics, which ultimately could carry over to better support for structural dynamics for any higher-level language that has a semantics defined by translation into MKL. Again, the implementation techniques discussed in this paper should be of interest in such a setting.

6 Future Work

In the current implementation of Hydra, a new flat system of equations is generated at each mode switch without reusing the equations of the previous mode. It would be interesting to identify exactly what has *changed* at each mode switch, thus allowing reuse of the equations from the previous mode as much as possible. We hope to benefit from Zimmer's related work on incremental symbolic processing methods for structurally dynamic, non-causal simulation [22, 23]. In particular, information about the equations that remain unchanged during the mode switches provides opportunities for the JIT compiler to reuse the machine code from the previous mode, thus reducing the burden on the JIT compiler and consequently the compilation time during mode switches.

We are considering approaches for further performance improvements by taking advantage of multi-core hardware. In principle, the functions i, f and e (see Figure 2) could be JIT compiled in parallel, which should give a respectable speedup on a two-core system. The functions could also be broken down into smaller functions, each independently compilable, thus potentially keeping a fair number of cores busy simultaneously.

Another, or possibly complementary, idea, is a mixed interpreter and JIT compiler approach. Numerical simulation would start directly after the symbolic processing stage by *interpreting* the simulation code. At the same time, a JIT

compilation process would be forked in the background. This seems particularly easy in the context of LLVM as it provides both an interpreter and JIT compiler for LLVM code. Numerical simulation would initially progress with an extra interpretive overhead. However once the JIT compilation process is finished, the interpreter would be substituted by the JIT compiled code. On multi-core hardware, this may lead to significant performance improvements by decreasing the delays during the mode switches and improving overall simulation time.

In the current version of Hydra, transfer of system state between modes and reinitialisation has to be coded explicitly at every switch for the entire system. This quickly becomes infeasible as models grow. Better language support for declaratively specifying implicit state transfer for unchanging parts of a system is thus something that we intend to look into.

7 Conclusions

This paper presents a novel approach to the implementation of non-causal modelling and simulation languages. It allows symbolic processing and code generation to be carried out as the model undergoes structural changes during the simulation, thus enabling non-causal modelling and simulation of highly structurally dynamic systems. Our approach provides an efficient alternative to interpreted implementations of structurally dynamic modelling languages and, at the same time, lifts the restrictions that are associated with pre-simulation compilation of non-causal modelling languages.

Our work is carried out in the framework of Functional Hybrid Modelling (FHM) because, by supporting higher-order modelling, this provides expressive language features for describing structurally dynamic systems. However, our implementation approach can be applied to other modelling languages that aim to support structural dynamism.

Acknowledgements. This work was supported by EPSRC grant EP/D064554/1. We would like to thank the anonymous reviewers for their thorough and constructive feedback that helped improve the paper.

References

[1] Kathryn Eleda Brenan, Stephen La Vern Campbell, and Linda Ruth Petzold. *Numerical solution of initial-value problems in differential-algebraic equations*. SIAM, Philadelphia, 1996.

[2] David Broman. Flow Lambda Calculus for declarative physical connection semantics. Technical Reports in Computer and Information Science 1, Linköping University Electronic Press, 2007.

[3] David Broman and Peter Fritzson. Higher-order acausal models. In Peter Fritzson, François Cellier, and David Broman, editors, *Proceedings of the 2nd International Workshop on Equation-Based Object-Oriented Languages and Tools (EOOLT)*, number 29 in Linköping Electronic Conference Proceedings, pages 59–69, Paphos, Cyprus, 2008. Linköping University Electronic Press.

[4] Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of ICFP'97: International Conference on Functional Programming*, pages 163–173, June 1997.

[5] George Giorgidze and Henrik Nilsson. Embedding a functional hybrid modelling language in Haskell. In *Refereed Proceedings of the 20th International Symposium on the Implementation and Application of Functional Languages (IFL '08)*, University of Hertfordshire, Hatfield, UK, September 2008. To Appear.

[6] Alan C. Hindmarsh, Peter N. Brown, Keith E. Grant, Steven L. Lee, Radu Serban, Dan E. Shumaker, and Carol S. Woodward. Sundials: Suite of nonlinear and differential/algebraic equation solvers. *ACM Trans. Math. Softw.*, 31(3):363–396, 2005.

[7] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.org`.

[8] Chris Lattner. Introduction to the llvm compiler system. In *Proceedings of International Workshop on Advanced Computing and Analysis Techniques in Physics Research*, Erice, Sicily, Italy, 2008.

[9] Chris Lattner and Vikram Adve. The LLVM Compiler Framework and Infrastructure Tutorial. In *LCPC'04 Mini Workshop on Compiler Research Infrastructures*, West Lafayette, Indiana, Sep 2004.

[10] Geoffrey Mainland. Why it's nice to be quoted: quasiquoting for haskell. In *Haskell '07: Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, pages 73–82, New York, NY, USA, 2007. ACM.

[11] The Modelica Association. *Modelica – A Unified Object-Oriented Language for Physical Systems Modeling: Tutorial version 1.4*, December 2000.

[12] Pieter J. Mosterman. *Hybrid Dynamic Systems: A Hybrid Bond Graph Modeling Paradigm and its Application in Diagnosis*. PhD thesis, Graduate School of Vanderbilt University, Nashville, Tennessee, May 1997.

[13] Pieter J. Mosterman. An overview of hybrid simulation phenomena and their support by simulation packages. In *HSCC '99: Proceedings of the Second International Workshop on Hybrid Systems*, pages 165–177, London, UK, 1999. Springer-Verlag.

[14] Pieter J. Mosterman, Gautam Biswas, and Martin Otter. Simulation of discontinuities in physical system models based on conservation principles. In *Proceedings of SCS Summer Conference 1998*, pages 320–325, July 1998.

[15] Henrik Nilsson. Type-based structural analysis for modular systems of equations. In Peter Fritzson, François Cellier, and David Broman, editors, *Proceedings of the 2nd International Workshop on Equation-Based Object-Oriented Languages and Tools*, number 29 in Linköping Electronic Conference Proceedings, pages 71–81, Paphos, Cyprus, July 2008. Linköping University Electronic Press.

[16] Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop (Haskell'02)*, pages 51–64, Pittsburgh, Pennsylvania, USA, October 2002. ACM Press.

[17] Henrik Nilsson, John Peterson, and Paul Hudak. Functional hybrid modeling. In *Proceedings of PADL'03: 5th International Workshop on Practical Aspects of Declarative Languages*, volume 2562 of *Lecture Notes in Computer Science*, pages 376–390, New Orleans, Lousiana, USA, January 2003. Springer-Verlag.

[18] Henrik Nilsson, John Peterson, and Paul Hudak. Functional hybrid modeling from an object-oriented perspective. In Peter Fritzson, François Cellier, and Christoph Nytsch-Geusen, editors, *Proceedings of the 1st International Workshop on Equation-Based Object-Oriented Languages and Tools (EOOLT)*, number 24 in Linköping Electronic Conference Proceedings, pages 71–87, Berlin, Germany, 2007. Linköping University Electronic Press.

[19] Christoph Nytsch-Geusen, Thilo Ernst, André Nordwig, Peter Schwarz, Peter Schneider, Matthias Vetter, Christof Wittwer, Thierry Nouidui, Andreas Holm, Jürgen Leopold, Gerhard Schmidt, Alexander Mattes, and Ulrich Doll. MOSILAB: Development of a modelica based generic simulation tool supporting model structural dynamics. In *Proceedings of the 4th International Modelica Conference*, pages 527–535, Hamburg, Germany, 2005.

[20] Günther Zauner, Daniel Leitner, and Felix Breitenecker. Modelling structural-dynamics systems in Modelica/Dymola, Modelica/MOSILAB, and AnyLogic. In Peter Fritzson, François Cellier, and Christoph Nytsch-Geusen, editors, *Proceedings of the 1st International Workshop on Equation-Based Object-Oriented Languages and Tools (EOOLT)*, number 24 in Linköping Electronic Conference Proceedings, pages 99–110, Berlin, Germany, 2007. Linköping University Electronic Press.

[21] Dirk Zimmer. Enhancing Modelica towards variable structure systems. In Peter Fritzson, François Cellier, and Christoph Nytsch-Geusen, editors, *Proceedings of the 1st International Workshop on Equation-Based Object-Oriented Languages and Tools (EOOLT)*, number 24 in Linköping Electronic Conference Proceedings, pages 61–70, Berlin, Germany, 2007. Linköping University Electronic Press.

[22] Dirk Zimmer. Introducing Sol: A general methodology for equation-based modeling of variable-structure systems. In *Proceedings of the 6th International Modelica Conference*, pages 47–56, Bielefeld, Germany, 2008.

[23] Dirk Zimmer. An application of Sol on variable-structure systems with higher index. In *Proceedings of the 7th International Modelica Conference*, Como, Italy, 2009.

