Mixed-level Embedding and JIT Compilation for an Iteratively Staged DSL

George Giorgidze and Henrik Nilsson

Functional Programming Laboratory
University of Nottingham
United Kingdom
{ggg,nhn}@cs.nott.ac.uk

Abstract. This paper explores how to implement an *iteratively staged* domain-specific language (DSL) by embedding into a functional language. The domain is modelling and simulation of physical systems where models are expressed in terms of non-causal differential-algebraic equations; i.e., sets of constraints solved through numerical simulation. What sets our language apart is that the equational constraints are *first class entities* allowing for an evolving model structure characterised by *repeated generation* of updated constraints. Hence iteratively staged. Our DSL can thus be seen as a combined functional and constraint programming language, albeit a two-level one, with the functional language chiefly serving as a meta language. However, the two levels do interact throughout the simulation. The embedding strategy we pursue is a mixture of deep and shallow, with the deep embedding enabling just-in-time (JIT) compilation of the constraints as they are generated for efficiency, while the shallow embedding is used for the remainder for maximum leverage of the host language. The paper is organised around a specific DSL, but our implementation strategy should be applicable for iteratively staged languages in general. Our DSL itself is further a novel variation of a declarative constraint programming language.

1 Introduction

Embedding is a powerful and popular way to implement domain-specific languages (DSLs) [8]. Compared with implementing a language from scratch, extending a suitable general-purpose programming language, the *host language*, with notions and vocabulary addressing a particular application or problem domain tends to save a lot of design and implementation effort.

There are two basic approaches to language embeddings: *shallow* and *deep*. In a shallow embedding, domain-specific notions are expressed directly in host-language terms, typically through a higher-order combinator library. This is a light-weight approach that makes it easy to leverage the facilities of the host language. However, the syntactic freedom is limited, and the static semantics of the embedded language must be relatively close to that of the host language for an embedding to be successful. In contrast, a deep embedding is centred around

a *representation* of embedded language terms that then are given meaning by interpretation or compilation. This is a more heavy-weight approach, but also more flexible. In particular, for optimisation or compilation, it is often necessary to inspect terms, suggesting a deep embedding. The two approaches can be combined to draw on the advantages of each. This leads to *mixed-level* embedding.

In this paper, we explore how to embed a language, Hydra [13], for non-causal modelling and simulation of physical systems into a functional programming language. In this application domain, systems are modelled by constraints expressed as undirected Differential Algebraic Equations (DAEs). These equations are solved by specialised combined symbolic and numerical simulation methods. A defining aspect of Hydra is that the equations are *first-class entities* in a functional language layer, providing very flexible means for expressing model composition and evolving model structure. Specifically, in response to *events*, which occur at discrete points in time, the simulation is stopped and, *depending* on results thus far, (partly) new equations are *generated* describing a (partly) new problem to be solved. We refer to this kind of DSL as *iteratively staged* to emphasise that the *domain* is characterised by repeated program generation and execution. Iterative staging makes it possible to model classes of systems in Hydra that current main-stream non-causal modelling and simulation languages cannot handle [6]. Section 2 exemplifies one such system.

Hydra can be seen as a functional and constraint or logical programming language in that it combines a functional and relational approach to programming. However, the integration of the two approaches is less profound than in, say, functional logic languages based on residuation or narrowing [7]. Hydra is a two-level language, where the functional part to a large extent serves as a meta language. However, the two layers do interact throughout the simulation.

We have chosen Haskell as the host language, or, more precisely, Haskell with Glasgow Haskell Compiler (GHC) extensions, GHC's quasiquoting facility [10, 11] being one reason for this choice. Because performance is a primary concern in the domain, the simulation code corresponding to the current equations has to be compiled. As this code is determined *dynamically*, this necessitates *just-in-time* (JIT) compilation. We use a deep embedding for this part of the language along with the Low-Level Virtual Machine (LLVM)[1]: a language-independent, portable, optimising, compiler back-end with JIT support. In contrast, we retain a shallow embedding for the parts of the embedded language concerned with high-level, symbolic computations to get maximum leverage from the host language. Note that we are not concerned with (hard) *real-time* performance here: we are prepared to pay the price of brief "pauses" for symbolic processing and compilation in the interest of minimising the computational cost of the actual simulation that typically dominates the overall costs by a wide margin.

An alternative might have been to use a *multi-staged* host language like MetaOCaml [15]. The built-in run-time code generation capabilities of the host language would then have been used instead of relying on an external code generation framework. We have so far not explored this approach as we wanted to

---
[1] http://llvm.org/

have tight control over the generated code. Also, not predicating our approach on a multi-staged host language means that some of our ideas and implementation techniques can be more readily deployed in other contexts, for example to enhance the capabilities of existing implementations of non-causal languages.

Compilation of Embedded DSLs (EDSLs) is today a standard tool in the DSL-implementer's tool chest. The seminal example is the work by Elliott et al. on compiling embedded languages, specifically the image synthesis and manipulation language Pan [3]. Pan, like our language, provides for program generation by leveraging the host language combined with compilation to speed up the resulting performance-critical computations. However, the program to be compiled is generated once and for all, meaning the host language acts as a powerful but fundamentally conventional macro language: program generation, compilation, and execution is a process with a fixed number of stages.

As Hydra is iteratively staged, the problems we are facing are in many ways different. Also, rather than acting merely as a powerful meta language that is out of the picture once the generated program is ready for execution, the host language is in our case part of the dynamic semantics of the embedded language through the shallow parts of the embedding. With this paper, we thus add further tools to the DSL tool chest for embedding a class of languages that hitherto has not been studied much. Specifically, our contributions are:

– a case study of mixed-level embedding of iteratively staged DSLs;
– using JIT compilation to implement an iteratively staged EDSL efficiently.

Additionally, we consider static type checking in the context of iterative staging and quasiquoting-based embedding. While Hydra is specialised, we believe the ideas underlying the implementation are of general interest, and that Hydra itself should be of interest to programming language researchers interested in languages that combine functional and relational approaches to programming. The implementation is available on-line[2] under the open source BSD license.

The rest of the paper is organised as follows. In Section 2, we introduce non-causal modelling and our language Hydra in more detail. Section 3 explains the Haskell embedding of Hydra and Section 4 then describes how iteratively staged programs are executed. Related work is discussed in Section 5. Finally, Section 6 gives conclusions.

2 Background

This section provides an introduction to Functional Hybrid Modelling (FHM) [13] and to our specific instance Hydra [5, 6]. We focus on aspects that are particularly pertinent to the present setting. The reader is referred to the earlier papers on FHM and Hydra for a more general treatment.

---

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

2.1 Functional Hybrid Modelling

Functional Hybrid Modelling (FHM) [13] is a new approach to designing *non-causal modelling languages* [2] supporting *hybrid systems*: systems that exhibit both continuous and discrete dynamic semantics. This class of languages is intended for modelling and simulation of systems that can be described by *Differential Algebraic Equations* (DAEs). Examples include electrical, mechanical, hydraulic, and other physical systems, as well as their combinations. *Non-causal*[3] in this context refers to treating the equations as being *undirected*: an equation can be used to solve any of the variables occurring in it. This is in contrast to *causal* modelling languages where equations are restricted to be *directed*: only "known" variables on one side of the equal sign, and only "unknown" variables on the other. Note that the domain of the variables are *time-varying values* or *signals*: functions of continuous time.

The advantages of non-causal languages over causal ones include that models are more *reusable* (the equations can be used in many ways) and more *declarative* (the modeller can focus on *what* to model, worrying less about *how* to model it to enable simulation) [2]. These are crucial advantages in many modelling domains. As a result, a number of successful non-causal modelling languages have been developed. Modelica[4] is a prominent, state-of-the-art example.

However, one well-known weakness of current non-causal languages is that their support for modelling *structurally dynamic systems*, systems where the equations that describe the dynamic behaviour change at discrete points in time, usually is limited. There are a number of reasons for this. A fundamental one is that languages like Modelica, to facilitate efficient simulation, are designed on the assumption that the model is translated into simulation code once and for all, *before* simulation starts.

The idea of FHM is to enrich a purely functional language with a few key abstractions for supporting hybrid, non-causal modelling. In particular, first-class *signal relations*, relations on signals described by undirected DAEs, provide support for non-causal modelling, and *dynamic switching* among signal relations that are *computed* at the point when they are being "switched in" provides support for describing highly structurally dynamic systems [13].

Our hypothesis is that the FHM approach will result in non-causal modelling languages that are more expressive than the current ones, yet have relatively simple, declarative semantics. Results so far have been promising. The capability to compute and use new signal relations during simulation has already allowed us to non-causally model and simulate some systems that e.g. Modelica cannot handle [6]. We present one such example in the following. The dynamic computation of and switching among signal relations is, of course, also what makes FHM iteratively staged.

---

[3] Do not confuse this with *temporal* causality. A system is temporally causal if its output only depends on present and past input, and temporally non-causal if the output depends on future input.
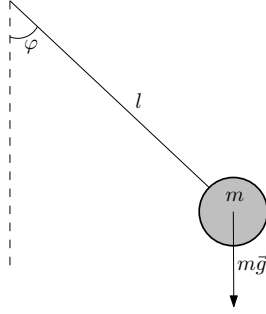
[4] http://www.modelica.org/

Fig. 1. A pendulum subject to gravity.

2.2 Hydra by Example: The Breaking Pendulum

To introduce Hydra, let us model a physical system whose structural configuration changes abruptly during simulation: a simple pendulum that can break at a specified point in time; see Figure 1. The pendulum is modelled as a point mass m at the end of a rigid, mass-less rod, subject to gravity $m\vec{g}$. If the rod breaks, the mass will fall freely. This makes the differences between the two configurations sufficiently large that e.g. Modelica does not support non-causal modelling of this system. Instead, if simulation across the breaking point is desired, the modeller is forced to model the system in a causal, less declarative way.

There are two levels to Hydra: the *functional level* and the *signal level*. The functional level is concerned with the definition of ordinary functions operating on time-invariant values. The signal level is concerned with the definition of relations between signals, the *signal relations*, and, *indirectly*, the definition of the *signals* themselves as solutions satisfying these relations.

Signal relations are *first-class* entities at the functional level. The type of a signal relation is parametrised on a *descriptor* of the types of the signals it relates: essentially a tuple of the types carried by the signals. For example, the type of a signal relation relating three real-valued signals is $SR\,(Real, Real, Real)$.

Signals, in contrast to signal relations, are *not* first-class entities at the functional level. However, crucially, *instantaneous values* of signals can be propagated back to the functional level, allowing the future system structure to depend on signal values at discrete points in time.

The definitions at the signal level may freely refer to entities defined at the functional level as the latter are time-invariant, known parameters as far as solving the equations are concerned. However, the opposite is not allowed: time-varying entities are confined to the signal level. The only signal-level notion that exists at the functional level is the *time-invariant* signal relation.

Hydra is currently implemented as an embedding in Haskell using *quasiquoting* [10, 11]. This means Haskell provides the functional level almost for free through shallow embedding. In contrast, the signal level is realised through deep embedding: signal relations expressed in terms of Hydra-specific syntax

```
type Coordinate = (Double, Double)          pendulum :: Double → Double
type Velocity = (Double, Double)                        → SR Body
type Body = (Coordinate, Velocity)          pendulum l phi0 = [$hydra|
                                              sigrel ((x, y), (vx, vy)) where
g :: Double                                     init phi      = $ phi0 $
g = 9.81                                        init der phi = 0
                                                init vx       = 0
freeFall :: Body → SR Body                      init vy       = 0
freeFall ((x0, y0), (vx0, vy0)) = [$hydra|      x             =    $ l $ * sin phi
  sigrel ((x, y), (vx, vy)) where               y             = −  $ l $ * cos phi
    init (x, y)       = ($x0$, $y0$)            (vx, vy)      = (der x, der y)
    init (vx, vy)     = ($vx0$, $vy0$)          der (der phi)
    (der x, der y)    = (vx, vy)                   + ($g / l$) * sin phi = 0
    (der vx, der vy)  = (0, −$g$)
|]                                          |]
```

Fig. 2. Models of the two modes of the pendulum.

are, through the quasiquoting machinery, turned into an internal representation that then is compiled into simulation code. This, along with the reasons for using quasiquoting, is discussed in more detail in an earlier paper [5]. However, that paper only treated *structurally static* systems.

Figure 2 shows how to model the two modes of the pendulum in Hydra. The type *Body* denotes the position and velocity of an object, where position and velocity both are 2-dimensional vectors represented by pairs of doubles. Each model is represented by a function that maps the *parameters* of the model to a relation on signals; i.e., an instance of the defining system of DAEs for specific values of the parameters. In the unbroken mode, the parameters are the length of the rod l and the initial angle of deviation *phi0*. In the broken mode, the signal relation is parametrised on the initial state of the body.

[$hydra| and |] are the open and close quasiquotes. Between them, we have signal-level definitions expressed in our custom syntax. 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.

Note the two kinds of variables: the functional level ones representing *time-invariant* parameters, and the signal-level ones, representing *time-varying* entities, the signals. Functional-level fragments, such as variable references, are spliced into the signal level by enclosing them between antiquotes, $. On the other hand time-varying entities are not allowed to escape to the functional level (meaning signal-variables are not in scope between antiquotes).

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. The operator *der* indicates differentiation with respect to time of the signal-valued expression to which it is applied.

$pendulumBE :: Double \rightarrow Double \rightarrow Double$
$\qquad \rightarrow SR\ (Body, E\ Body)$
$pendulumBE\ t\ l\ phi0 = [\$hydra|$
$\quad$ **sigrel** $(((x, y), (vx, vy)), \textbf{event}\ e)$ **where**
$\qquad \$\ pendulum\ l\ phi0\ \$\ \diamond ((x, y), (vx, vy))$
$\quad$ **event** $e = ((x, y), (vx, vy))$
$\qquad\qquad$ **when** $time = \$t\ \$$
$|]$

$breakingPendulum :: SR\ Body$
$breakingPendulum =$
$\quad switch$
$\qquad (pendulumBE\ 10\ 1\ (pi\ /\ 4))$
$\qquad freeFall$

(a) Pendulum extended with a breaking event   (b) Composition using switch

Fig. 3. The breaking pendulum

The non-causal nature of Hydra can be seen particularly clearly in the last equation of the unbroken mode that simply states a constraint on the angle of deviation and its second derivative, without making any assumption regarding which of the two time-varying entities is going to be used to solve for the other (both g and l are time-invariant functional-level variables).

To model a pendulum that breaks at some point, we need to create a composite model where the model that describes the dynamic behaviour of the unbroken pendulum is replaced, at the point of breaking, by the model describing a free falling body. These two submodels must be suitably joined to ensure the continuity of both the position and velocity of the body of the pendulum.

To this end, the *switch*-combinator, which forms signal relations by temporal composition, is used:

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

The composite behaviour is governed by the first signal relation until an *event* of type b occurs ($E\ b$ in the type signature above). At this point, the second argument to switch is applied to the value carried by the event to *compute* the signal relation that is going to govern the composite behaviour from then on. Event signals are *discrete-time* signals, signals that are only defined at (countably many) discrete points in time, as opposed to the continuous-time signals that (conceptually) are defined everywhere on a continuous interval of time. Each point of definition of an event signal is known as an *event occurrence*. Unlike continuous-time signals, the causality of event signals is always fixed.

Figure 3 shows how *switch* is used to construct a model of a breaking pendulum. The *pendulum* model is first extended into a signal relation *pendulumBE* that also provides the event signal that defines when the pendulum is to break: see figure 3(a). In our case, an event is simply generated at an *a priori* specified point in time, but the condition could be an arbitrary time-varying entity. The value of the event signal is the state (position and velocity) of the pendulum at that point, allowing the succeeding model to be initialised so as to ensure the continuity of the position and velocity as discussed above.

To bring the equations of *pendulum* into the definition of *pendulumBE*, *pendulum* is first 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, using the *signal relation application operator* $\diamond$. This instantiates the equations of *pendulum* in the context of *pendulumBE*. 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 $\diamond$. See [5] for further details.

Finally, a model of the breaking pendulum can be composed by switching form *pendulumBE* to *freeFall*: see figure 3(b). 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 pendulum body in free fall initialised so as to ensure the continuity of its position and velocity.

In our particular example, the pendulum is only going to break once. In other words, there is not much iteration going on, and it would in principle (with a suitable language design) be straightforward to generate code for both modes of operation prior to simulation. However, this is not the case in general. For example, given a parametrised signal relation:

$$sr1 :: Double \rightarrow SR\ ((Double, Double), E\ Double)$$

we can recursively define a signal relation *sr* that describes an overall behaviour by "stringing together" the behaviours described by *sr1*:

$$sr :: Double \rightarrow SR\ (Double, Double)$$
$$sr\ x = switch\ (sr1\ x)\ sr$$

In this case, because the number of instantiations of *sr1* in general cannot be determined statically (and because each instantiation can depend on the parameter in arbitrarily complex ways), there is no way to generate all code prior to simulation. However, the pendulum example is simple and suffice for illustrative purposes. Moreover, despite its simplicity, it is already an example with which present non-causal languages struggle, as mentioned above.

In practical terms, the *switch*-combinator is a somewhat primitive way of describing variable model structure. Our aim is to enrich Hydra with higher-level constructs as descried in the original FHM paper [13]. The basic aspects of the implementation should, however, not change much.

3   Embedding

In this section, we describe the Haskell embedding of Hydra in further detail. First, we introduce a Haskell data type that represents an embedded signal relation. This representation is untyped. We then introduce typed combinators that ensures that only well-typed signal relations can be constructed.

The following data type is the central, untyped representation of signal relations. There are two ways to form a signal relation: either from a set of defining equations, or by composing signal relations temporally:

data $SigRel =$
$\quad SigRel \qquad Pattern\ [\,Equation\,]$
$\quad |\ SigRelSwitch\ SigRel \quad (Expr \rightarrow SigRel)$

The constructor $SigRel$ forms a signal relation from equations. Such a relation is represented by a pattern and the list of defining equations. The pattern serves the dual purpose of describing the *interface* of the signal relation in terms of the types of values carried by the signals it relates and their time domains (continuous time or discrete time/events), and of introducing names for these signals for use in the equations. Patterns are just nested tuples of signal variable names along with indications of which ones are event signals: we omit the details. The list of equations constitute a system of Differential Algebraic Equations (DAEs)[5] that defines the signal relation by expressing constraints on the (signal) variables introduced by the pattern and any additional local variables.

The *switch*-combinator forms a signal relation by temporal composition of two signal relations. Internally, such a temporal composition is represented by a signal relation constructed by $SigRelSwitch$. The first argument is the signal relation that is initially active. The second argument is the function that, in the case of an event occurrence from the initially active signal relation, is used to compute a new signal relation from the value of that occurrence. This new signal relation then becomes the active one, replacing the initial signal relation.

Note the use of a mixture of shallow and deep techniques of embedding. The embedded function in a signal relation constructed by $SigRelSwitch$ corresponds to the shallow part of the embedding. The rest of the data types constitute a deep embedding, providing an explicit representation of language terms for further symbolic processing and ultimately compilation, as we will see in more detail below. The following data type represents equations. There are four different kinds:

data $Equation =$
$\quad EquationInit \quad Expr \quad Expr \qquad |\ EquationEq \qquad Expr \quad Expr\ |$
$\quad EquationEvent\ String\ Expr\ Expr\ |\ EquationSigRelApp\ SigRel\ Expr$

Initialisation equations, constructed by $EquationInit$, provide initial conditions. They are thus only in force when a signal relation instance first becomes active.

Equations constructed by $EquationEq$ are basic equations imposing the constraint that the valuations of the two expressions have to be equal for as long as the containing signal relation instance is active (e.g., equations like $der\ (der\ x) = 0$). Equations constructed by $EquationEvent$ define event signals; i.e., they represent equations like **event** $e = (x, y)$ **when** $time = 3$. These equations are

---

[5] Although not necessarily a *fixed* such system as these equations may refer to signal relations that contain switches.

directed. The string is the name of the defined event signal. The first expression gives the value of the event signal at event occurrences. The second expression defines these occurrences. An event occurs whenever the signal represented by this expression *crosses* 0. For the above example, the expression defining the event occurrences would thus be $time - 3$.

The fourth kind of equation is signal relation application, *EquationSigRelApp*, i.e. equations like $sr \diamond (x, y + 2)$. This brings all equations of a signal relation into scope by instantiating them for the expressions to which the relation is applied.

Finally, the representation of expressions is a standard first-order term representation making it easy to manipulate expressions symbolically (e.g. computing symbolic derivatives) and compiling expressions to simulation code:

data *Expr = ExprUnit | ExprReal Double | ExprVar String | ExprTime |*
ExprTuple Expr Expr [Expr] | ExprApp Function [Expr]
data *Function = FuncDer | FuncNeg | FuncAdd | FuncnSub | FuncMul | . . .*

We use quasiquoting, a recent Haskell extension implemented in Glasgow Haskell Compiler (GHC), to provide a convenient surface syntax for signal relations. We have implemented a quasiquoter that takes a string in the concrete syntax of Hydra and generates Haskell code that builds the signal relation in the mixed-level representation described above. GHC executes the quasiquoter for each string between the quasiquotes before type checking.

While the internal representation of a signal relation is untyped, Hydra itself is typed, and we thus have to make sure that only type-correct Hydra programs are accepted. As Hydra fragments are generated dynamically, during simulation, we cannot postpone the type checking to after program generation. Nor can we do it early, at quasiquoting time, at least not completely, as no type information from the context around quasiquoted program fragments are available (e.g., types of antiquoted Haskell expressions). In the current version of Hydra, only domain specific scoping rules (e.g., all constrained signal variables must be declared) are checked at the stage of quasiquoting. Fortunately, the type system of the present version of Hydra is fairly simple; in particular, Hydra is simply typed, so by using the standard technique of phantom types, the part of the type checking that requires type information outside the quasiquotes is delegated to the host language type checker [14].

A phantom type is a type whose type constructor has a parameter that is not used in its definition. We define phantom type wrappers for the untyped representation as follows:

data *SR a*   *= SR*   *SigRel*
data *PatternT a = PatternT Pattern*
data *ExprT a*   *= ExprT*   *Expr*
data *E a*

Phantom types can be used to restrict a function to building only type-correct domain-specific terms. For example, a typed combinator *sigrel* can be defined in the following way:

$$sigrel :: PatternT\ a \rightarrow [\,Equation\,] \rightarrow SR\ a$$
$$sigrel\ (PatternT\ p)\ eqs = SR\ (SigRel\ p\ eqs)$$

As can be seen, the type of the pattern that defines the interface of the signal relation is what determines its type.

Similarly, we define a typed combinator *switch*:

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

E is a type constructor with no constituent data constructors. It is used to type patterns that introduce event signals. The data for the event signals are constructed using event equations.

A signal relation that is defined using the *switch* combinator is structurally dynamic. However, the type of the *switch* combinator statically guarantees that its type (i.e., its interface) remains unchanged. Thus, a structurally dynamic signal relation can be used in a signal relation application just like any other signal relation.

Well-typed equations are constructed using combinators in a similar way:

$$equationEq :: ExprT\ a \rightarrow ExprT\ a \rightarrow Equation$$

$$equationSigRelApp :: SR\ a \rightarrow ExprT\ a \rightarrow Equation$$

Typed combinators for the remaining parts of the language, including *Pattern* and *Expr*, are defined using the same technique.

Under the hood the representation is still untyped. However, if only the typed combinators are exposed for building of signal relations, it is guaranteed that only well-typed terms can be constructed. The quasiquoter of Hydra has only access to typed combinators for building signal relations.

Symbolic transformations (e.g., symbolic differentiation and flattening) on embedded language terms work with the untyped representation. These transformations need to be programmed with care as the Haskell type checker cannot verify that the transformations are type preserving.

Several type system extensions of Haskell (e.g., generalised algebraic data types, existential types, and type families) make alternative techniques for typing EDSLs possible. One alternative would be to directly construct signal relations in typed representation and implement the symbolic transformations on the typed representation. While this approach requires more work from the EDSL implementer, it provides additional correctness guarantees (e.g., the Haskell type checker could be used to verify that transformations are type preserving). We have not yet evaluated suitability of Haskell type system for such undertaking and opted for simpler, untyped representation.

4   Simulation

In this section we describe how an iteratively staged Hydra program is run. The process is illustrated in Figure 4 and is conceptually divided into four stages. In
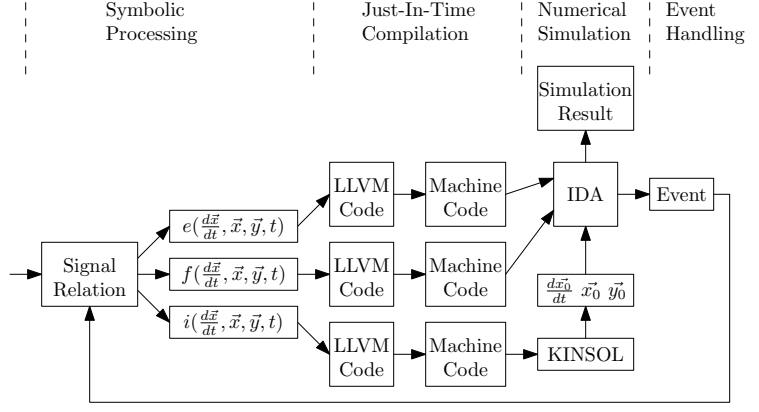
Fig. 4. Execution model of Hydra

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 JIT 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.

As a first step, all signal variables are renamed to give them distinct names. This helps avoiding name clashes during *flattening*, signal relation application unfolding, and thus simplifies this process. 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.

Finally, the following equations are generated at the end of the stage of symbolic processing: $i(\frac{d\vec{x}}{dt}, \vec{x}, \vec{y}, t) = 0, t = t_0$ (1), $f(\frac{d\vec{x}}{dt}, \vec{x}, \vec{y}, t) = 0$ (2), and $e(\frac{d\vec{x}}{dt}, \vec{x}, \vec{y}, t) = 0$ (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 (signals crossing 0).

As the functions i, f, and e are invoked from within inner loops of the solver, they have to be compiled into machine code for efficiency: any interpretive over-

head here would be considered intolerable by practitioners for most applications. However, as Hydra allows the equations to be changed in arbitrary ways *during* simulation, the equations have to be compiled whenever they change, as opposed to only prior to simulation. Our Hydra implementation employs 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 native machine code. Function pointers to the generated machine code are then passed to the numerical solver.

The numerical suite used in the current implementation of Hydra is called SUNDIALS[6]. The components we use are KINSOL, a nonlinear algebraic equation systems solver, and IDA, a 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. Event detection facilities are provided by IDA.

At the moment of an event occurrence (one of the signals monitored by e crossing 0), the numerical simulator terminates and presents the following information to an event handler: Name of the event variable for which an event occurrence has been detected, time t_e of the event occurrence and instantaneous values of the signal variables (i.e., values of $\frac{d\vec{x}}{dt}$, $\vec{x}$ and $\vec{y}$ at time t_e).

The event handler traverses the original unflattened signal relation and finds the event value expression (a signal-level expression) that is associated with the named event 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. 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 the instantaneous value of $((x, y), (vx, vy))$. The result of this application is a new signal relation. The part of the original unflattened signal relation is updated by replacing the old signal relation with the new one. The flat system of equations for the previous mode and the machine code that was generated for it by the LLVM JIT compiler are discarded. The simulation process for the updated model continues from the first stage and onwards.

In previous work [6], we conducted benchmarks to evaluate the performance of the proposed execution model. The initial results are encouraging. For a small system with handful of equations (e.g., the breaking pendulum) the total time spent on run-time symbolic processing and code generation is only a couple of hundredth of a second. To get an initial assessment of how well our approach scales, we also conducted a few large scale benchmarks (thousands of equations). These demonstrated that the overall performance of the execution model seems to scale well. In particular, time spent on run-time symbolic processing

---

[6] http://www.llnl.gov/casc/sundials/

and JIT compilation increased roughly linearly in the number of equations for these tests. The results also demonstrate that the time spent on JIT compilation dominates over the time spent on run-time symbolic processing. Above all, our benchmarks indicated that the time for symbolical processing and compilation remained modest in absolute terms, and thus should be relatively insignificant compared with the time for simulation in typical applications.

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 may be useful to identify exactly what has changed at each mode switch, thus enabling the reuse of *unchanged* equations and associated code from the previous mode. In particular, this could reduce the burden placed on the JIT compiler, which in our benchmarks accounted for most of the switching overheads. Using such techniques, it may even be feasible to consider our kind of approach for structurally dynamic (soft) *real-time* applications.

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 and simulate such systems non-causally.

5   Related Work

The deep embedding techniques used in the Hydra implementation for domain-specific optimisations and efficient code generation draws from the extensive work on compiling staged domain-specific embedded languages. Examples include Elliott et al. [3] and Mainland et al. [11]. However, these works are concerned with compiling programs all at once, meaning the host language is used only for meta-programming, not for running the actual programs.

The use of quasiquoting in the implementation of Hydra was inspired by Flask, a domain-specific embedded language for programming sensor networks [11]. However, we had to use a different approach to type checking. A Flask program is type checked by a domain-specific type checker *after* being generated, just before the subsequent compilation into the code to be deployed on the sensor network nodes. This happens at *host language run-time*. Because Hydra is iteratively staged, we cannot use this approach: we need to move type checking back to *host language compile-time*. The Hydra implementation thus translates embedded programs into typed combinators at the stage of quasiquoting, charging the host language type checker with checking the embedded terms. This ensures only well-typed programs are generated at run-time.

Lee et al. are developing a DSL embedded in Haskell for data-parallel array computations on a graphics processing unit (GPU) [9]. GPU programs are first-class entities. The embedded language is being designed for run-time code generation, compilation and execution, with results being fed back for use in further host language computations. Thus, this is another example of what we

term iterative staging. At the time of writing, the implementation is interpreted. However, a JIT compiler for a GPU architecture is currently being developed.

The FHM design was originally inspired by Functional Reactive Programming (FRP) [4], particularly Yampa [12]. A key difference is that FRP provides *functions* on signals whereas FHM generalises this to *relations* on signals. FRP can thus be seen as a framework for *causal* simulation, while FHM supports non-causal simulation. Signal functions are first class entities in most incarnations of FRP, and new ones can be computed and integrated into a running system dynamically. This means that these FRP versions, including Yampa, also are examples of iteratively staged languages. However, as all FRP versions supporting highly dynamic system structure so far have been interpreted, the program generation aspect is much less pronounced than what is the case for FHM.

In the area of non-causal modelling, Broman's work on the Modelling Kernel Language (MKL) has a number of similarities to FHM [1]. 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 like in FHM. However, support for structural dynamism has not yet been considered.

Non-causal languages that do support more general forms of structural dynamism than the current mainstream ones include MOSILAB[7], a Modelica extension, and Sol [16], a Modelica-like language. MOSILAB has a compiled implementation, but insists all structural configurations are predetermined to enable compilation once and for all, prior to simulation. Sol is less restrictive, but currently only has an interpreted implementation. Both MOSILAB and Sol could thus benefit from the implementation techniques described in this paper.

6   Conclusions

In this paper we presented a novel implementation approach for non-causal modelling and simulation languages supporting structural dynamism. Our approach was to embed an iteratively staged DSL in Haskell, using a mixed-level embedding to capitalise maximally on the host language while simultaneously enabling an efficient implementation through JIT compilation. The iterative staging allows us to model systems that current, main-stream, non-causal languages cannot handle without resorting to causal modelling. As far as we are aware, this is the first compiled implementation of a non-causal modelling language that supports highly structurally dynamic systems, demonstrating the practical feasibility of such a language as compilation of simulation code is considered essential for most practical applications for reasons of performance.

The use of the EDSL approach was instrumental to achieve the above. By reusing the features of the host language and its tool chain, we could focus our efforts on the problems that are specific to non-causal modelling and simulation. We also note that LLVM has worked really well for our purposes. Compilation

---

[7] http://www.mosilab.de/

of iteratively staged embedded languages does not seem to have attracted much attention thus far. We hope the implementation techniques we have developed will be useful to others who are faced with implementing such languages.

References

1. D. Broman and P. Fritzson. Higher-order acausal models. In *Proceedings of the 2nd International Workshop on Equation-Based Object-Oriented Languages and Tools*, 2008.
2. F. E. Cellier. Object-oriented modelling: Means for dealing with system complexity. In *Proceedings of the 15th Benelux Meeting on Systems and Control*, 1996.
3. C. Elliott, S. Finne, and O. de Moor. Compiling embedded languages. In *Semantics, Applications, and Implementation of Program Generation*, 2000.
4. C. Elliott and P. Hudak. Functional reactive animation. In *Proceedings of International Conference on Functional Programming*, 1997.
5. G. Giorgidze and H. Nilsson. Embedding a functional hybrid modelling language in Haskell. In *Proceedings of the 20th International Symposium on the Implementation and Application of Functional Languages*, 2008.
6. G. Giorgidze and H. Nilsson. Higher-order non-causal modelling and simulation of structurally dynamic systems. In *Proceedings of the 7th International Modelica Conference*, 2009.
7. M. Hanus, H. Kuchen, and J. J. Moreno-Navarro. Curry: A truly functional logic language. In *Proceedings Workshop on Visions for the Future of Logic Programming*, 1995.
8. P. Hudak. Modular domain specific languages and tools. In *Proceedings of Fifth International Conference on Software Reuse*, 1998.
9. S. Lee, M. Chakravarty, V. Grover, and G. Keller. GPU kernels as data-parallel array computations in Haskell. In *Workshop on Exploiting Parallelism using GPUs and other Hardware-Assisted Methods*, 2009.
10. G. Mainland. Why it's nice to be quoted: quasiquoting for Haskell. In *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, 2007.
11. G. Mainland, G. Morrisett, and M. Welsh. Flask: Staged functional programming for sensor networks. In *Proceedings of the Thirteenth ACM SIGPLAN International Conference on Functional Programming*, 2008.
12. H. Nilsson, A. Courtney, and J. Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop*, 2002.
13. H. Nilsson, J. Peterson, and P. Hudak. Functional hybrid modeling. In *Proceedings of 5th International Workshop on Practical Aspects of Declarative Languages*, 2003.
14. M. Rhiger. A foundation for embedded languages. *ACM Trans. Program. Lang. Syst.*, 2003.
15. W. Taha. A gentle introduction to multi-stage programming. In *Domain-Specific Program Generation*, 2004.
16. D. Zimmer. Introducing Sol: A general methodology for equation-based modeling of variable-structure systems. In *Proceedings of the 6th International Modelica Conference*, 2008.