FIRST-CLASS MODELS

On a Noncausal Language for Higher-order and
Structurally Dynamic Modelling and Simulation

GEORGE GIORGIDZE, BSc, MSc

Thesis submitted to the University of Nottingham
for the degree of Doctor of Philosophy

April 2012

Abstract

The field of physical modelling and simulation plays a vital role in advancing numerous scientific and engineering disciplines. To cope with the increasing size and complexity of physical models, a number of modelling and simulation languages have been developed. These languages can be divided into two broad categories: causal and noncausal. Causal languages express a system model in terms of directed equations. In contrast, a noncausal model is formulated in terms of undirected equations. The fact that the causality can be left implicit makes noncausal languages more declarative and noncausal models more reusable. These are considered to be crucial advantages in many physical domains.

Current, mainstream noncausal languages do not treat equational models as first-class values; that is, a model cannot be parametrised on other models or generated at simulation runtime. This results in very limited higher-order and structurally dynamic modelling capabilities, and limits the expressiveness and applicability of noncausal languages.

This thesis is about a novel approach to the design and implementation of noncausal languages with first-class models supporting higher-order and structurally dynamic modelling. In particular, the thesis presents a language that enables: (1) higher-order modelling capabilities by embedding noncausal models as first-class entities into a functional programming language and (2) efficient simulation of noncausal models that are generated at simulation runtime by runtime symbolic processing and just-in-time compilation. These language design and implementation approaches can be applied to other noncausal languages. This thesis provides a self-contained reference for such an undertaking by defining the language semantics formally and providing an in-depth description of the implementation. The language provides noncausal modelling and simulation capabilities that go beyond the state of the art, as backed up by a range of examples presented in the thesis, and represents a significant progress in the field of physical modelling and simulation.

"Unprovided with original learning, unformed in the habits of thinking, unskilled in the arts of composition, I resolved to write a book."

Edward Gibbon

For Mari and Mia Valentina

Acknowledgements

I would like to thank Henrik Nilsson for his help and guidance throughout my PhD studies, starting from our first paper and finishing with this thesis. My examiners, Brian Logan and Simon Thompson, provided an extensive and detailed feedback that helped to improve the thesis. I am grateful to Joey Capper and Neil Sculthorpe for reading and commenting on early drafts of this thesis. I am also grateful to Mari Chikvaidze and Kate Demirtas for proofreading several chapters of the thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The field of physical modelling and simulation plays a vital role in the design, implementation and analysis of systems in numerous areas of science and engineering. Examples include electronics, mechanics, thermodynamics, chemical reaction kinetics, population dynamics and neural networks [Cellier, 1991]. To cope with the increasing size and complexity of physical models, a number of modelling and simulation languages have been developed. The modelling and simulation languages can be divided into two broad categories: *causal* and *noncausal*.

A causal model is formulated in terms of *explicit* equations, for example, *ordinary differential equations* (ODEs) in explicit form; that is, the cause-effect relationship is explicitly specified by the modeller [Cellier and Kofman, 2006]. In other words, the equations are directed: only *unknown* variables can appear on the left hand side of the equals sign, and only *known* variables on the other side. Since the equations are directed, it is relatively straightforward to translate a causal model into low-level simulation code (e.g., into a sequence of assignment statements) and simulate it. Simulink is a prominent representative of causal modelling languages [Simulink, 2008].

A noncausal model is formulated in terms of *implicit* equations, for example, *differential algebraic equations* (DAEs) in implicit form. In other words, the equations are undirected: both known and unknown variables may appear on both sides of the equals sign [Cellier and Kofman, 2006]. The translation of noncausal models into simulation code involves additional symbolic processing and numerical simulation methods that are not required for causal modelling and simulation. Examples include symbolic transformations that try to causalise noncausal models and, if this is not possible, numerical solvers for (nonlinear) implicit equations. Modelica is a prominent, state-of-the-art representative of noncausal modelling languages [Modelica, 2010].

Noncausal modelling has a number of advantages over causal modelling. The most important ones are outlined in the following list.

- In many physical domains models are more naturally represented using noncausal equations, and in some physical domains models cannot be represented using only causal equations [Cellier, 1991, Brenan et al., 1996].

- Noncausal languages are declarative and approach modelling problems from a higher level of abstraction by focusing on *what* to model rather than *how* to model to enable simulation [Cellier, 1991, 1996, Nilsson et al., 2003].

- Noncausal models are more reusable, as equations can be used in a number of different ways depending on their context of usage (i.e., effectively causalised in a number of different ways) [Cellier, 1991, 1996, Cellier and Kofman, 2006].

Although causal modelling remains a dominant paradigm, interest in noncausal modelling has grown recently as evidenced by the wide adoption of the Modelica language both in industry and academia, and by release of noncausal modelling and simulation tools by prominent vendors such as Maple (MapleSim) and MathWorks (Simscape).

1.1 First-class Models

A language entity is *first-class* if it can be (1) passed as a parameter to functions, (2) returned as a result from functions, (3) constructed at runtime and (4) placed in data structures [Scott, 2009]. To my knowledge, this notion was first introduced by Christopher Strachey [Burstall, 2000] in the context of functions being first-class values in higher-order, functional programming languages. Current, mainstream noncausal languages do not treat models as first-class values [Nilsson et al., 2003]. This limits their expressiveness for *higher-order* and *structurally dynamic* modelling.

1.1.1 Higher-order Modelling

Higher-order modelling allows parametrisation of models on other models [Nilsson et al., 2003]. For instance, a car model can be parametrised on the list of tyres it is using, and an electrical transmission line model can be parametrised on the list of electrical components on the line. Mainstream, noncausal languages provide limited

support for this style of modelling. Tool specific and external scripting languages are often used to generate noncausal models for particular instances of higher-order models [Broman and Fritzson, 2008]. This is practical for some applications, but the aforementioned advantages of noncausal languages can be better realised with a coherent language supporting noncausal as well as higher-order modelling.

This thesis formally defines a language that supports higher-order modelling by treating noncausal models as first-class values in a purely functional programming language and describes its implementation. In this setting, a function from model (or from collections of models placed in a suitable data structure) to model can be seen as a higher-order model and an application of this function can be seen as an instantiation of the higher-order model.

The idea of treating noncausal models as first-class values in a functional programming language is introduced by Nilsson et al. [2003] in the context of a framework called Functional Hybrid Modelling (FHM) for designing and implementing noncausal modelling languages. However, the paper postpones the concrete language definition and implementation for future work. In addition, the FHM framework proposes to exploit the first-class nature of noncausal models for modelling *hybrid* systems (i.e., systems that exhibit both continuous and discrete behaviour); this is relevant in the following section.

Broman [2007] defined and implemented a noncausal language that supports parametrisation of models on other models and allows for a form of higher-order modelling. However, construction of noncausal models at simulation runtime and manipulation of collections of models placed in data structures were not considered.

1.1.2 Structurally Dynamic Modelling

Major changes in system behaviour are often modelled by changing the equations that describe the system [Mosterman, 1997]. A model where the equational description changes over time is called *structurally dynamic*. Each structural configuration of the model is known as a *mode* of operation. Cellier and Kofman [2006] refer to structurally dynamic systems as *variable-structure* systems. Structurally dynamic systems are an example of the more general notion of hybrid systems [Nilsson et al., 2003]. The term structurally dynamic emphasises only one discrete aspect, that is, the change of equations at discrete points in time.

Cyber-physical systems [Lee, 2008], where digital computers interact with continuous physical systems, can also be seen as instances of hybrid systems. In this context, structurally dynamic modelling is relevant; as modelling of a cyber-physical system where the digital part's influence causes major changes in the physical part may require changing the equations that describe the behaviour of the continuous part. Recently, the US National Science Foundation identified cyber-physical systems as one of its key research areas [NSF, 2008].

Currently, noncausal languages offer limited support for modelling structurally dynamic systems [Mosterman, 1997, 1999, Zauner et al., 2007, Zimmer, 2008]. There are a number of reasons for this. However, this thesis concentrates on one particular reason related to the design and implementation of modelling and simulation languages: the prevalent assumption that most or all processing to put a model into a form suitable for simulation will take place *prior* to simulation [Nilsson et al., 2007, Zimmer, 2007]. 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 a sequence of assignment statements) with little or no need for dynamic memory or code management. This results in good performance, but such language design and implementation approaches restrict the number of modes to be modest as, in general, separate code must be generated for each mode. This rules out supporting structurally dynamic systems where the number of modes is a priori unbounded. We refer to this kind of system as *unbounded structurally dynamic*. Systems with a priori bounded number of modes are referred as *bounded structurally dynamic*.

There are a number of efforts to design and implement modelling and simulation languages with improved support for structural dynamism. Examples include: HYBRSIM [Mosterman et al., 1998], MOSILAB [Nytsch-Geusen et al., 2005], Sol [Zimmer, 2008] and Acumen [Zhu et al., 2010]. However, thus far, implementations have either been interpreted (HYBRSIM and Sol) and thus sacrificing efficiency, or the languages have been restricted so as to limit the number of modes to make it feasible to compile code for all modes prior to simulation (MOSILAB and Acumen).

1.2 Contributions to the Field of Noncausal Modelling and Simulation

This dissertation presents a novel approach to the design and implementation of noncausal modelling and simulation languages with first-class models supporting higher-order and structurally dynamic modelling. The thesis formally defines a noncausal modelling language called Hydra and describes its implementation in detail. Hydra provides noncausal modelling and simulation capabilities that go beyond the state of the art and represents significant progress in the field of design and implementation of declarative modelling and simulation languages. The following list summarises the contributions to the field of noncausal modelling and simulation.

- The thesis shows how to enable higher-order modelling capabilities by embedding noncausal models as first-class entities into a purely functional programming language. To my knowledge, Hydra is the first noncausal language that faithfully treats equational models as first-class values (i.e., supports all four points outlined in the beginning of Section 1.1). See Section 2.6, Section 2.7, Chapter 3 and Chapter 4 for details.

- The thesis shows how to use runtime symbolic processing and *just-in-time* (JIT) compilation to enable efficient simulation of noncausal models that are generated at simulation runtime. To my knowledge, Hydra is the first noncausal language that enables support *both* for modelling and simulation of unbounded structurally dynamic systems and for compilation of simulation code for efficiency. See Chapter 6 for details.

- The thesis formally defines the Hydra language. To my knowledge, Hydra is the first noncausal language that features a formal specification capturing both continuous and discrete aspects of unbounded structural dynamism. See Chapter 5 for details.

In addition to presenting the language definition and implementation, the aforementioned claims are also backed up by a range of example physical systems that cannot be modelled and simulated in current, noncausal languages. The examples are carefully chosen to showcase those language features of Hydra that are lacking in other noncausal modelling languages.

15

The language design choices and implementation approaches presented here can be used to enhance existing noncausal modelling and simulation languages, as well as to design and implement new modelling languages. This thesis provides a self-contained reference for such an undertaking by defining the language semantics formally and providing an in-depth description of the implementation.

Many language features of Hydra follow closely those proposed by Nilsson et al. [2003] in the context of the FHM framework and can be seen as the first concrete language definition and implementation that is based on the FHM framework. However, as already mentioned, at this stage Hydra supports only one aspect of hybrid modelling, namely, structural dynamism. Other discrete aspects that do not lead to structural reconfigurations (e.g., *impulses* [Nilsson et al., 2003, Nilsson, 2003]) are not considered in this thesis, but, in principle, can be incorporated in the Hydra language.

This work can be seen as an application of successful ideas developed in functional programming languages research to declarative modelling and simulation languages. I hope that this work will aid to further cross-fertilisation and the exchange of ideas between these research communities.

1.3   Embedding

Hydra is a Haskell-embedded *domain-specific language* (DSL). Here, the domain is noncausal modelling and simulation using implicitly formulated DAEs. Haskell is a purely functional, higher-order, statically typed programming language [Peyton Jones, 2003], which is widely used for embedded DSL development [Stewart, 2009].

Embedding is a powerful and popular way to implement DSLs [Hudak, 1998]. Compared with implementing a language from scratch, extending a suitable general-purpose programming language, the *host language*, with notions addressing a particular application or problem domain tends to save a lot of design and implementation effort. The motivation behind using an embedding approach for Hydra is to concentrate the language design and implementation effort on noncausal modelling notions that are domain specific and absent in the host language, and to reuse the rest from the host language.

Having said that, the concept of first-class models, and the runtime symbolic processing and JIT compilation approaches implemented in Hydra, are not predicated on embedded implementation. These language design and implementation approaches can be used in other noncausal modelling languages, embedded or otherwise.

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. A shallow embedding is commonly realised as a higher-order combinator library. This is a light-weight approach for leveraging the facilities of the host language [Hudak, 1998]. In contrast, a deep embedding is about building embedded language terms as data in a suitable representation. These terms are given meaning by interpretation or compilation [Hudak, 1998]. This is a more heavy-weight approach, but also more flexible one. Indeed, it is often necessary to inspect the embedded language terms for optimisation or compilation. To benefit from the advantages of both shallow and deep embeddings, a combined approach called *mixed-level* embedding can be used [Giorgidze and Nilsson, 2010]. The aforementioned embedding approaches are not specific to Haskell. They can be realised in other higher-order programming languages (e.g., languages in ML and Lisp families).

As mentioned in Section 1.1, Hydra supports runtime generation and JIT compilation of noncausal models. Specifically, in response to *events* occurring at discrete points in time, the simulation is stopped and, depending on the simulation results thus far, new equations are generated for further simulation [Giorgidze and Nilsson, 2009]. In this thesis and in Giorgidze and Nilsson [2010] this kind of DSLs are referred to as *iteratively staged*, emphasising that the domain is characterised by repeated program generation, compilation and execution. An iteratively-staged language is a special kind of a *multi-staged* language [Taha, 2004] with the aforementioned characteristics.

Because performance is a primary concern in the domain, the numerical simulation code for each mode of the model has to be compiled. As this code is determined dynamically, this necessitates JIT compilation. For the numerical part of the language Hydra employs deep embedding techniques, along with the Low Level Virtual Machine (LLVM) compiler infrastructure [Lattner, 2002], a language-independent, portable, optimising, compiler backend with JIT support. In contrast, shallow embedding techniques are used for the parts of Hydra that are concerned with high-level, symbolic computations [Giorgidze and Nilsson, 2010].

An alternative might have been to use a multi-staged host language like MetaO-Caml [Taha, 2004]. The built-in runtime code generation capabilities of the host language then would have been used instead of relying on an external code generation framework such as LLVM. This approach has not been pursued, as tight control over the dynamically generated numerical code is essential in this application domain.

1.4 Contributions to the Field of DSL Embedding

Compilation of embedded DSLs is today a standard tool in the DSL-implementer's tool box. The seminal example is the work by Elliott et al. on compiling embedded languages, specifically the image synthesis and manipulation language Pan [Elliott et al., 2000]. Pan, like Hydra, 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.

Hydra is iteratively staged and the host language is part of the dynamic semantics of the embedded language through the shallow parts of the embedding (instead of acting merely as a meta language that is out of the picture once the generated program is ready for execution). We thus add further tools to the DSL tool box for embedding a class of languages that thus far has not been studied much from an embedding and staged programming perspective.

While embedded DSL development methodology is not the main focus of this work, I nevertheless think that the thesis should be of interest to embedded DSL implementers, as it presents an application of a new embedding technique. The following list summarises the contributions to the field of DSL embedding.

- The thesis presents a case study of mixed-level embedding of an iteratively staged DSL in a host language that does not provide built-in multi-stage programming capabilities. See Chapter 3 and Chapter 4 for details.

- The thesis describes how to use JIT compilation to implement an iteratively staged embedded DSL efficiently. See Chapter 6 for details.

1.5 Overview of Peer-reviewed Publications

The content of this thesis is partly based on the peer-reviewed publications that are listed in this section. I wrote the papers in collaboration with my coauthors. This thesis was written by myself and presents my own contributions. I have implemented

the Hydra language described in this dissertation and in the following papers. The software is available on my webpage[1] under the open source BSD3 license.

The following four papers describe various aspects of the design and implementation of Hydra, as well as a number of its applications.

- George Giorgidze and Henrik Nilsson. Embedding a Functional Hybrid Modelling language in Haskell. In *Revised selected papers of the 20th international symposium on Implementation and Application of Functional Languages, Hatfield, England*, volume 5836 of *Lecture Notes in Computer Science*. Springer, 2008.

- George Giorgidze and Henrik Nilsson. Higher-order non-causal modelling and simulation of structurally dynamic systems. In *Proceedings of the 7th International Modelica Conference, Como, Italy*. Linköping University Electronic Press, 2009.

- George Giorgidze and Henrik Nilsson. Mixed-level embedding and JIT compilation for an iteratively staged DSL. In *Revised selected papers of the 19th international workshop on Functional and (Constraint) Logic Programming, Madrid, Spain*, volume 6559 of *Lecture Notes in Computer Science*. Springer, 2010.

- Henrik Nilsson and George Giorgidze. Exploiting structural dynamism in Functional Hybrid Modelling for simulation of ideal diodes. In *Proceedings of the 7th EUROSIM Congress on Modelling and Simulation, Prague, Czech Republic*. Czech Technical University Publishing House, 2010.

The following two papers are about unbounded structurally dynamic, causal modelling and simulation using Yampa, a Haskell-embedded Functional Reactive Programming (FRP) language [Hudak et al., 2003]. The combinator that allows switching of equations during simulation runtime in Hydra draws its inspiration from switching combinators featured in Yampa [Nilsson et al., 2002, Courtney et al., 2003].

- George Giorgidze and Henrik Nilsson. Demo outline: Switched-on Yampa. In *Proceedings of the ACM SIGPLAN Haskell workshop, Freiburg, Germany*. ACM, 2007.

---

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

- George Giorgidze and Henrik Nilsson. Switched-on Yampa: declarative programming of modular synthesizers. In *Proceedings of the 10th international symposium on Practical Aspects of Declarative Languages, San Francisco, CA, USA*, volume 4902 of *Lecture Notes in Computer Science*. Springer, 2008.

Some of the embedding techniques described in this thesis are also used in the following two papers.

- George Giorgidze, Torsten Grust, Tom Schreiber, and Jeroen Weijers. Haskell boards the Ferry: Database-supported program execution for Haskell. In *Revised selected papers of the 22nd international symposium on Implementation and Application of Functional Languages, Alphen aan den Rijn, Netherlands*, volume 6647 of *Lecture Notes in Computer Science*. Springer, 2010. Peter Landin Prize for the best paper at IFL 2010.

- George Giorgidze, Torsten Grust, Nils Schweinsberg, and Jeroen Weijers. Bringing back monad comprehensions. In *Proceedings of the ACM SIGPLAN Haskell symposium, Tokyo, Japan*. ACM, 2011.

1.6 Prerequisites

Some parts of the thesis assume that the reader is familiar with Haskell, predicate logic, and BNF notation. Haskell is used for defining Hydra, as well as for implementing it. BNF notation is used for specifying the concrete syntax of Hydra. Predicate logic is used for explaining the language concepts and to give the ideal semantics of Hydra.

Readers unfamiliar with Haskell may refer to the language report by Peyton Jones [2003] or one of the following books: Thompson [1999], Hutton [2007], Hudak [1999] or O'Sullivan et al. [2008]. Having said that, readers familiar with other higher-order, typed functional programming languages, such as Standard ML [Milner et al., 1997], should also be able to follow the thesis in its entirety.

It is worthwhile to mention that only small subset of the Haskell features are needed to model and simulate physical systems in Hydra. The modeller should know:

- how to define a function by providing its name, arguments and result,

- how to apply a function to arguments,

- how to write a function type signature involving arbitrarily nested pairs of basic types,

- how to write functions that operate on lists (this is only needed for higher-order modelling with collections of models),

- and how to use functions as first class values.

Other features of Haskell, most notably laziness and type classes, are not needed to model in Hydra. The aforementioned two features are not used in the language implementation either. The implementation of Hydra makes use of the following two Haskell extensions available in Glasgow Haskell Compiler (GHC)[2]: quasiquoting [Mainland, 2007] and generalised algebraic data types (GADTs) [Peyton Jones et al., 2006].

1.7 Outline

The rest of the dissertation is organised as follows.

- Chapter 2 overviews the field of physical modelling, and the state-of-the-art causal and noncausal modelling languages.

- Chapter 3 introduces the central concepts of the Hydra language and its design.

- Chapter 4 explains how to model physical systems in Hydra by means of instructive examples. The examples were carefully chosen to showcase those language features that are absent in other noncausal modelling languages.

- Chapter 5 formally defines Hydra's concrete syntax, abstract syntax, type system and ideal semantics. The chapter formally defines the equational part of Hydra. The reader is referred to Peyton Jones [2003] for the semi-formal definition of the host functional language.

- Chapter 6 describes how Hydra is implemented.

- Chapter 7 overviews the related work.

- Chapter 8 concludes the thesis.

---

[2]`http://www.haskell.org/ghc`

Chapter 2

Background

Hydra is a domain-specific language. The domain of the language is equational modelling and simulation of physical systems. In order to make the thesis self contained, this chapter gives background information on the language domain.

The three essential steps involved in the process of modelling and simulation of a physical system are given in the following list.

- Mathematical modelling of the system behaviour

- Translation of the mathematical representation into a computer program

- Simulation of the system by compiling and executing the computer program

This chapter illustrates the aforementioned three steps by using simple and instructive examples. We start by conducting these steps manually. We then demonstrate how causal and noncausal modelling languages and tools can be used to automate this process, and discuss advantages and disadvantages of current, mainstream modelling languages.

In addition, by modelling and simulating the example physical systems, basic concepts of modelling and simulation are introduced. Where necessary, the presentation abstracts from the concrete examples and defines the basic concepts generally.

2.1   Equational Modelling

Figure 2.1 depicts a simple electrical circuit. The circuit is grounded and consists of four two-pin electrical components: a voltage source, a resistor, an inductor and a capacitor. The following system of equations is an equational model of the circuit.
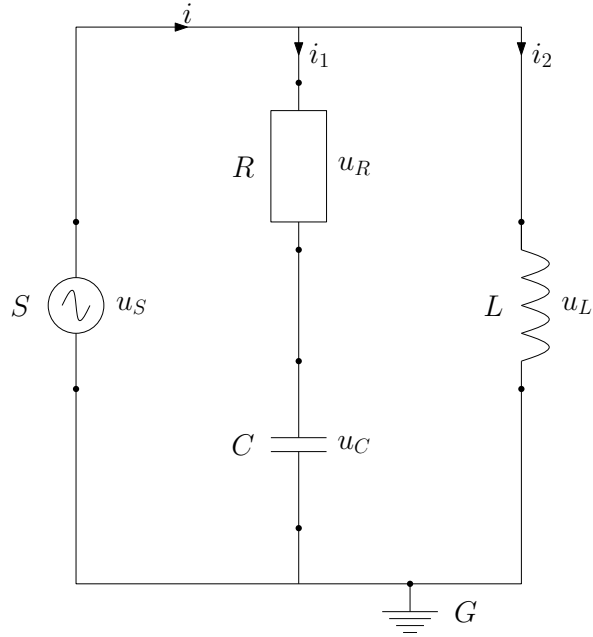
Figure 2.1: Simple electrical circuit.

$$u_S \;=\; sin(2\pi t) \tag{2.1a}$$

$$u_R \;=\; R \cdot i_1 \tag{2.1b}$$

$$i_1 \;=\; C \cdot \frac{du_C}{dt} \tag{2.1c}$$

$$u_L \;=\; L \cdot \frac{di_2}{dt} \tag{2.1d}$$

$$i_1 + i_2 \;=\; i \tag{2.1e}$$

$$u_R + u_C \;=\; u_S \tag{2.1f}$$

$$u_S \;=\; u_L \tag{2.1g}$$

The first four equations describe the component behaviours. The last three equations describe the circuit topology. The system of equations consists of undirected algebraic and differential equations[1]. This mathematical representation is a system of implicit *differential algebraic equations* (DAEs) [Cellier and Kofman, 2006]. More generally, a system of implicit DAEs can be written in the following form:

---

[1] Cellier [1991] provides wealth of information on how to derive equational models for physical systems.

$$f(\frac{d\vec{x}}{dt}, \vec{x}, \vec{y}, t) = 0 \qquad (2.2)$$

Here, $\vec{x}$ is a vector of *differential variables* (i.e., their derivatives with respect to time appear in the equations), $\vec{y}$ is a vector of *algebraic variables* (i.e., their derivatives with respect to time do not appear in the equations) and t is an independent scalar variable. In physical modelling t denotes *time*. Differential variables are also referred as *state variables*.

Numerical integration is a widely used approach for deriving approximate solutions of systems of DAEs. This is partly because, in general, exact symbolic methods do not suffice for solving systems of DAEs [Brenan et al., 1996]. There are a number of methods for numerical integration of an implicit DAE. For example, there are numerical solvers that directly operate on the implicit representation (e.g., the IDA solver from the SUNDIALS numerical suite [Hindmarsh et al., 2005]), however, in some cases it is possible to translate a DAE into a system of explicit *ordinary differential equations* (ODEs), which makes it possible to simulate the system using an ODE solver (e.g., the CVODE solver from the SUNDIALS numerical suite [Hindmarsh et al., 2005]). In the following we illustrate the latter approach, as ODE solvers are much simpler to implement. For an equational model that can be transformed to an ODE it is preferable to use an ODE solver for numerical integration, because ODE solvers are usually more efficient than DAE solvers.

2.2 Causalisation

In order to transform the implicit DAE describing the simple electrical circuit into an explicit one, we perform the following steps. Firstly, we identify *known* and *unknown* variables. Secondly, we decide which unknown variable should be solved in which equation. Thirdly, we sort the equations in such a way that no unknown variable is used before it is solved.

Time t and the differential variables u_c and i_2 are assumed to be known, the rest of the variables are unknowns, including the derivatives ($\frac{du_c}{dt}$ and $\frac{di_2}{dt}$). The equations that contain only one unknown are solved for it. After that, the solved variables are assumed to be known and rest of the variables are solved. In this case this technique suffices and we get the following explicit DAE:

$$u_S = sin(2\pi t) \tag{2.3a}$$

$$u_L = u_S \tag{2.3b}$$

$$u_R = u_S - u_C \tag{2.3c}$$

$$i_1 = \frac{u_R}{R} \tag{2.3d}$$

$$i = i_1 + i_2 \tag{2.3e}$$

$$\frac{du_C}{dt} = \frac{i_1}{C} \tag{2.3f}$$

$$\frac{di_2}{dt} = \frac{u_L}{L} \tag{2.3g}$$

This symbolic manipulation process is called *causalisation*[2]. Now the direction of equations is explicitly specified which was not the case for the implicit DAE.

Let us substitute the variables defined in the first five equations into the last two equations. This effectively eliminates the algebraic equations from the system.

$$\frac{du_C}{dt} = \frac{sin(2\pi t) - u_C}{R \cdot C} \tag{2.4a}$$

$$\frac{di_2}{dt} = \frac{sin(2\pi t)}{L} \tag{2.4b}$$

This representation is a system of explicit ODEs and can be passed to a numerical ODE solver. This representation is also called *state-space model*. More generally, a system of explicit ODEs can be written in the following form:

$$\frac{d\vec{x}}{dt} = f(\vec{x}, t) \tag{2.5}$$

Here, $\vec{x}$ is a vector of differential variables and t is time.

2.3 Numerical Integration

Let us give an illustration of the process of numerical integration through a concrete method. In the following the *forward Euler* method, which is the simplest numerical

---

[2]In general, the process of causalisation can be more involved than one described in this section. Cellier and Kofman [2006] give a good survey of partial and complete causalisation methods.

integration method for ODEs, is explained. The key idea is to replace the derivatives with the following approximation:

$$\frac{d\vec{x}}{dt} \approx \frac{\vec{x}(t+h) - \vec{x}(t)}{h} \qquad (2.6)$$

Here, h is a *sufficiently small* positive scalar which is referred to as the *step size* of the numerical integration.

Let us make use of Equation 2.5 and substitute the derivative.

$$\vec{x}(t+h) \approx \vec{x}(t) + h \cdot f(\vec{x}, t) \qquad (2.7)$$

Let us also fix the step size h and construct the following discrete sequences:

$$t_0 = 0, t_1 = t_0 + h, \ t_2 = t_1 + h, ..., t_n = t_{n-1} + h, ... \qquad (2.8)$$

$$\vec{x}_0 = \vec{x}(t_0), ..., \vec{x}_{n+1} = \vec{x}_n + h \cdot f(\vec{x}_n, t_n), \ ... \qquad (2.9)$$

Here, $\vec{x}_n$ is a numerical approximation of $\vec{x}(t_n)$.

More accurate and efficient numerical integration methods are available based on different approximations and integration algorithms. A comprehensive presentation of this and other more sophisticated methods can be found in the book by Cellier and Kofman [2006].

2.4   Simulation

Once an initial condition (i.e., a value of the differential vector at time zero) is given it is possible to numerically integrate the ODE. The Haskell code that is given in Figure 2.2 numerically integrates the ODE given in Equation 2.4 using the forward Euler method.

Given the numerical integration time step and the circuit parameters, this function computes the approximate solution and delivers the values of the differential vector at the discrete points of time given in Equation 2.9 as a list.

In the case of the simple electrical circuit model, the algebraic variables can be solved by adding more directed equations in the function that numerically integrates the system of equations. The Haskell code given in Figure 2.3 refines the integration function by adding the directed equation that solves the algebraic variable i_1. Fig-

$$integrateSimpleCircuit \; :: \; Double \rightarrow Double \rightarrow Double \rightarrow Double$$
$$\rightarrow [(Double, Double, Double)]$$
$$integrateSimpleCircuit \; dt \; r \; c \; l = go \; 0 \; 0 \; 0$$
$$\textbf{where}$$
$$go \; t \; uc \; i2 = \textbf{let} \; di2 = (sin \; (2 * \pi * t) \; / \; l) * dt$$
$$duc = ((sin \; (2 * \pi * t) - uc) \; / \; (r * c)) * dt$$
$$\textbf{in} \; (t, i2, uc) : go \; (t + dt) \; (uc + duc) \; (i2 + di2)$$

Figure 2.2: Function that numerically integrates the ODE given in Equation 2.4 using the forward Euler method.

$$integrateSimpleCircuit \; :: \; Double \rightarrow Double \rightarrow Double \rightarrow Double$$
$$\rightarrow [(Double, Double, Double, Double)]$$
$$integrateSimpleCircuit \; dt \; r \; c \; l = go \; 0 \; 0 \; 0$$
$$\textbf{where}$$
$$go \; t \; uc \; i2 = \textbf{let} \; di2 = (sin \; (2 * \pi * t) \; / \; l) * dt$$
$$duc = ((sin \; (2 * \pi * t) - uc) \; / \; (r * c)) * dt$$
$$i_1 \;\; = (sin \; (2 * \pi * t) - uc) \; / \; r$$
$$\textbf{in} \; (t, i2, uc, i_1) : go \; (t + dt) \; (uc + duc) \; (i2 + di2)$$

Figure 2.3: Function that adds one directed equation to the function given in Figure 2.2.

ure 2.4 shows a partial simulation result obtained by evaluating the function with the additional directed equation.

The simple electrical circuit example highlights the three essential steps involved in the process of modelling and simulation outlined in the introduction of this chapter.

As we have already seen, for some systems, it is feasible to conduct this process manually. Indeed translation of systems of equations into code in general purpose programming languages like Fortran, C, Java or Haskell is a common practice. However, manual translation becomes tedious and error prone with growing complexity. Imagine conducting the process presented in this section for a physical system described with hundreds of thousands of equations. Modelling languages and simulation tools can help with all three phases mentioned above as discussed in the following sections of this chapter.
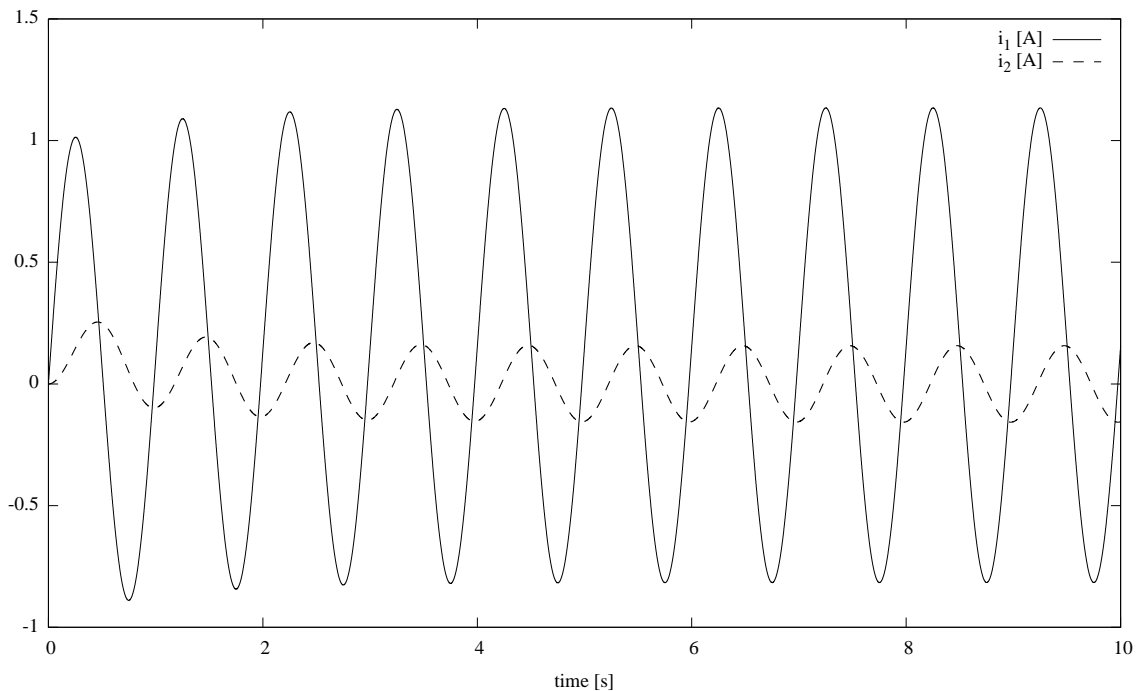
Figure 2.4: Plot showing how variables i_1 and i_2 change over time.

2.5 Causal Modelling

The block diagram depicted in Figure 2.5 is a model of the simple electrical circuit from Figure 2.1. Note that the diagram uses causal blocks (with inputs and outputs) for multiplication, summation and integration. The block diagram is a graphical representation of Equation 2.3. In order to make this correspondence clear, the block outputs for the variables U_S, i_1, i_2 and i are labelled with the corresponding variable name.

Block diagrams in causal languages correspond to systems of ODEs in explicit form. The construction of a block diagram is closely related to the process of causalisation. The causal model given in Figure 2.5 can be simulated by graphical block diagramming tools such as Simulink. Derivation of simulation code from a block diagram is done much in the same way as described earlier, but using more sophisticated numerical methods.

Structurally, the block diagram in Figure 2.5 is quite far removed from the circuit it models. Because of this, construction of block diagrams is generally regarded as a difficult and somewhat involved task [Nilsson et al., 2007]. Moreover, a slight change in

28
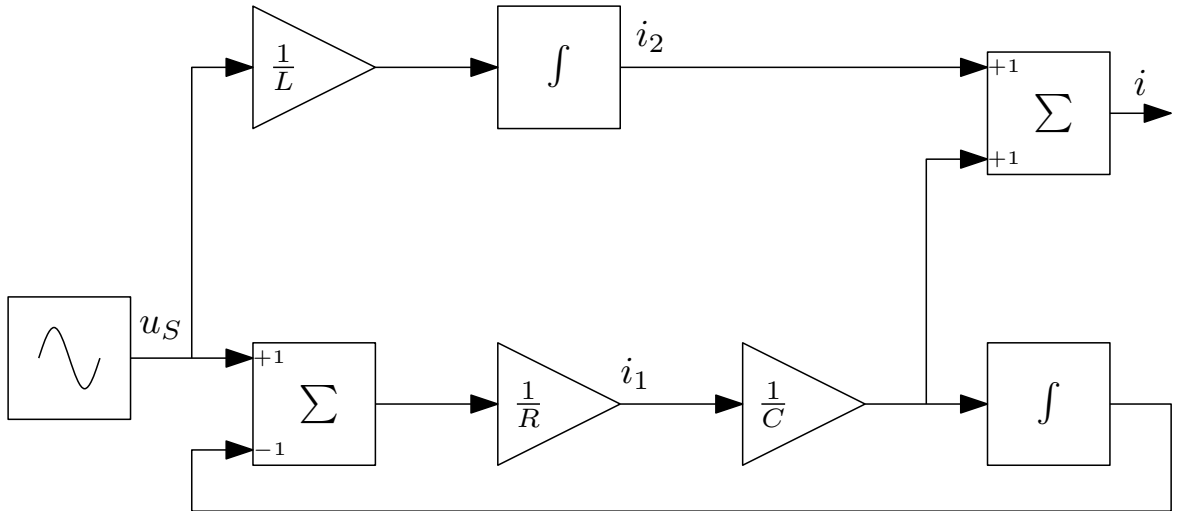
Figure 2.5: Block diagram modelling electrical circuit depicted in Figure 2.1.

a modelled system might require drastic changes in the corresponding block diagram. This is because causal models limit reuse [Cellier, 1996]. For example, a resistor behaviour is usually modelled using Ohm's law which can be written as $i = \frac{u}{R}$ or $u = R \cdot i$. Unfortunately, no single causal block can capture the resistor behaviour. If we need to compute the current from the voltage, we should use the block that corresponds to the first equation. If we need to compute the voltage from the current, we should use the block that corresponds to the second equation.

To demonstrate the aforementioned reuse problem, we modify the simple electrical circuit by adding one more resistor, as shown in Figure 2.6, and then causally model it as shown in Figure 2.7. Note that we were unable to reuse the resistor model from the original circuit diagram. Furthermore, a simple addition to the physical system caused changes to the causal model that are hardly obvious.

The block diagramming tool Simulink can be used to model bounded structurally dynamic systems: special blocks are used to *switch* between block diagrams as a response to discrete events. This makes Simulink very useful for modelling of bounded structurally dynamic systems. However, the number of modes (i.e., structural configurations) must be finite and all modes must be predetermined before simulation. Thus Simulink does not enable modelling and simulation of unbounded structurally dynamic systems. In addition, Simulink block diagrams are first order, thus Simulink does not support higher-order causal modelling.
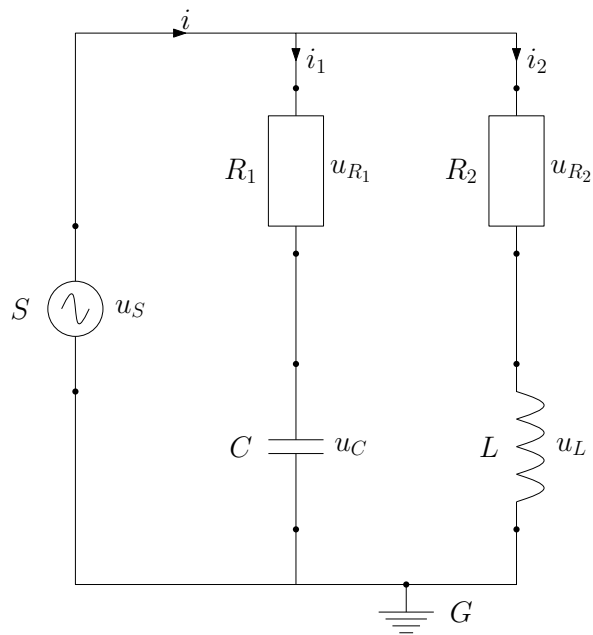
Figure 2.6:   Simple electrical circuit with two resistors.
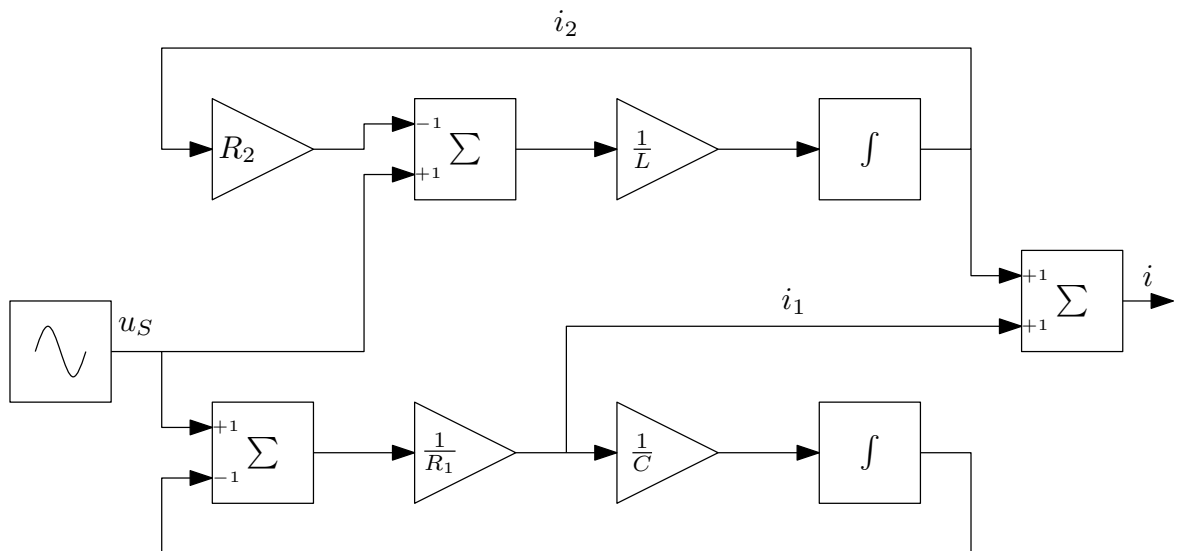
Figure 2.7:   Block diagram modelling electrical circuit depicted in Figure 2.6.

```
connector Pin
    flow Real i;
    Real v;
end Pin;
```

Figure 2.8:   Connector record defined in Modelica.

2.6   Noncausal Modelling Illustrated through Modelica

Modelica is a declarative language for noncausal modelling and simulation of physical systems. Modelica models are given using implicit DAEs. Modelica features a class system similar to what can be found in many object-oriented programming languages for structuring equations and for supporting model reuse.

This section presents a Modelica model of the simple electrical circuit depicted in Figure 2.1 to illustrate basic features of the language.

The Modelica code that is given in Figure 2.8 declares the *connector* record for representing electrical connectors. The connector record introduces the variable i and the variable v representing the current flowing into the connector and the voltage at the connector respectively. In Modelica, connector records do not introduce equations. The meaning of the flow annotation is explained later on when *connect equations* are introduced.

The Modelica code that is given in Figure 2.9 defines the model that captures common properties of electrical components with two connectors. The variables p and n represent the positive and negative pins of an electrical component. The variable u represents the voltage drop across the component. The variable i represents the current flowing into the positive pin. The *TwoPin* model defines the noncausal equations that these variables satisfy.

By *extending* the *TwoPin* model with component-specific equations Figure 2.10 defines the models representing a resistor, a capacitor, an inductor and a voltage source. Figure 2.10 also defines the model that represents the ground pin. Note the use of the concept of *inheritance* known from object-oriented programming languages for reusing the equations from the *TwoPin* model.

Variables qualified as **parameter** or as **constant** remain unchanged during simulation. The value of a constant is defined once and for all in the source code, while a

31

```
model TwoPin
   Pin  p, n;
   Real u, i;
equation
   u =  p.v − n.v;
   0 =  p.i + n.i;
   i  =  p.i;
end TwoPin;
```

Figure 2.9:   Modelica model for two-pin electrical components.

parameter can be set when an object of the class is instantiated. In this example all parameters are provided with default values allowing for instantiations with the default parameter values. All other variables represent dynamic, time-varying entities. Note that the expressions $der\,(u)$ and $der\,(i)$ denote time derivatives of the variables u and i respectively.

The Modelica model that is given in Figure 2.11 uses the circuit component models to define the simple electrical circuit model by "connecting" appropriate pins according to Figure 2.1.

Connect statements are analysed and appropriate *connection equations* are generated by the Modelica compiler as follows. Connected flow variables generate sum-to-zero equations. In this case, as the domain is electrical circuits, the sum-to-zero equations correspond to Kirchhoff's current law. For the *SimpleCircuit* model the Modelica compiler generates the following three sum-to-zero equations:

$$AC.n.i + C.n.i + L.n.i + G.p.i = 0;$$
$$R.n.i \;\; + C.p.i \qquad\qquad\quad = 0;$$
$$AC.p.i + R.p.i + L.p.i \qquad\quad = 0;$$

Connected potential variables generate equality constraints stating that all connected potential variables are equal at any point in time. For the *SimpleCircuit* model the Modelica compiler generates the following six equations:

```
model Resistor
  extends TwoPin;
  parameter Real R = 1;
equation
  R * i = u;
end Resistor;


model Capacitor
  extends TwoPin;
  parameter Real C = 1;
equation
  C * der (u) = i;
end Capacitor;


model Inductor
  extends TwoPin;
  parameter Real L = 1;
equation
  u = L * der (i);
end Inductor;


model VSourceAC
  extends TwoPin;
  parameter Real VA      = 1;
  parameter Real FreqHz = 1;
  constant Real PI        = 3.14159;
equation
  u = VA * sin (2 * PI * FreqHz * time);
end VSourceAC;


model Ground
  Pin p;
equation
  p.v = 0;
end Ground;
```

Figure 2.10:   Modelica models with component-specific equations.

```
model SimpleCircuit
  Resistor    R;
  Capacitor   C;
  Inductor    L;
  VSourceAC  AC;
  Ground     G;
equation
  connect (AC.p, R.p);
  connect (AC.p, L.p);
  connect (R.n,   C.p);
  connect (AC.n, C.n);
  connect (AC.n, L.n);
  connect (AC.n, G.p);
end SimpleCircuit;
```

Figure 2.11: Modelica model for the circuit given in Figure 2.1.

$$AC.n.v = C.n.v;$$
$$C.n.v = L.n.v;$$
$$L.n.v = G.p.v;$$
$$R.n.v = C.p.v;$$
$$AC.p.v = R.p.v;$$
$$R.p.v = L.p.v;$$

Connect statements can be used in any physical domain where flow variables (i.e., variables generating sum-to-zero equations at the connection points) and potential variables (i.e, variables generating equality constrains at the connection points) can be identified. The Modelica standard library includes examples of their usage in electrical, hydraulic, and mechanical domains.

Modelica compilers generate executable simulation code from hierarchical systems of equations structured using object-oriented programming constructs by utilising state-of-the-art symbolic and numerical methods.

As we have seen, noncausal languages allow us to model physical systems at a high level of abstraction. The structure of the models resemble the modelled systems. Consequently, it is easy to reuse or modify existing models. For example, it is now trivial to add one more resistor to the Modelica model as shown in Figure 2.12.

```
model SimpleCircuit
  Resistor    R1;
  Resistor    R2;
  Capacitor   C;
  Inductor    L;
  VSourceAC  AC;
  Ground      G;
equation
  connect (AC.p, R1.p);
  connect (AC.p, R2.p);
  connect (R1.n, C.p);
  connect (R2.n, L.p);
  connect (AC.n, C.n);
  connect (AC.n, L.n);
  connect (AC.n, G.p);
end SimpleCircuit;
```

Figure 2.12:   Modelica model for the circuit given in Figure 2.6.

2.7   Noncausal Modelling of Structurally Dynamic Systems

A structurally dynamic system is usually modelled using a combination of continuous equations and switching statements that specify discontinuous changes in the system. This section is about structurally dynamic modelling in noncausal languages. Current limitations are illustrated using a Modelica model of a simple structurally dynamic system. In particular, this section highlights the lack of expressiveness of the Modelica language when it comes to dynamic addition and removal of time-varying variables and continuous equations, and lack of runtime symbolic processing and code generation facilities in Modelica implementations.

Let us model a system whose structural configuration changes abruptly during simulation: a simple pendulum that can break at a specified point in time; see Figure 2.13. The pendulum is modelled as a body represented by a point mass m at the end of a rigid, mass-less rod, subject to gravity $m\vec{g}$. If the rod breaks, the body will fall freely.

The code that is given in Figure 2.14 is an attempt to model this system in Modelica that on the surface appears to solve the problem. Unfortunately the code fails to compile. The reason is that the latest version of the Modelica standard [Modelica,
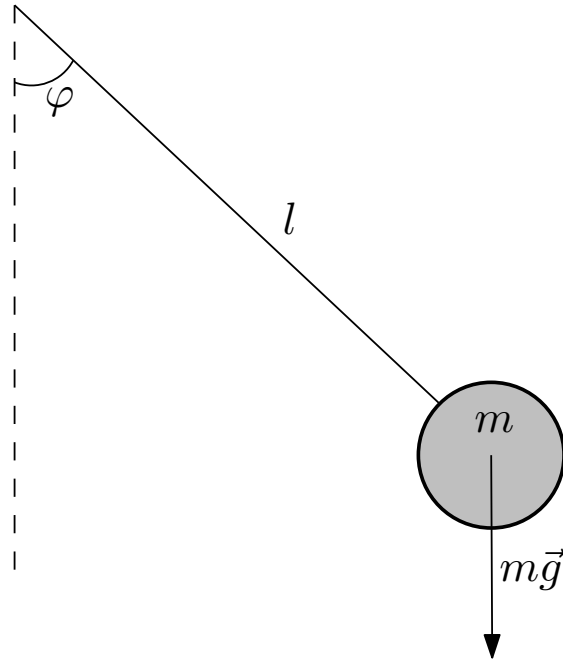
Figure 2.13: Pendulum subject to gravity.

2010] asserts that number of equations in both branches of an if statement must be equal when the conditional expression contains a time-varying variable. If considered separately, the equations in both branches do solve the publicly available variables successfully. In an attempt to fix the model, the modeller might try to add a dummy equation for the variable not needed in the second mode (i.e., the variable ϕ, which represents the angle of deviation of the pendulum before it is broken). This version compiles, but the generated code fails to simulate the system. This example was tried using the OpenModelica [Fritzson et al., 2006] and Dymola [Dymmola, 2008] compilers.

One of the difficulties of this example is that causality changes during the switch between the two modes. In the first mode the position is calculated from the differential variable ϕ, which is not the case after the switch. This makes the job of the simulation code generator a lot harder and, as it turns out, the Modelica tools are not able to handle it. Specifically, the tools commit to a certain causality before they generate the simulation code. This and related issues are covered in greater detail in [Modelica Tutorial, 2000]. The suggested Modelica solution is more involved and requires reformulation of the model by making it causal. The need for manual reformulation to conform to a certain causality eliminates the advantages of working in a noncausal modelling language.

36

```
model BreakingPendulum
  parameter Real l = 1, φ₀ = π / 4, t = 10;
  Real x, y, vₓ, vᵧ;
  protected Real φ (start = π / 2);
equation
  vₓ  = der x
  vᵧ  = der y
  if (time < t) then
    x =    l * sin φ
    y = − l * cos φ
    0 = der (der φ) + (g / l) * sin φ
  else
    der vₓ = 0
    der vᵧ = − g
  end if ;
end BreakingPendulum;
```

Figure 2.14:   Attempt to model a breaking pendulum in Modelica.

Currently, the Modelica language lacks the expressiveness to describe structural changes. The breaking pendulum example demonstrates the problems that arise when the number of variables in the system changes. In addition, the Modelica compilers carry out the symbolic processing and generate the simulation code all at once, prior to simulation, which introduces further limitations.

There are a number of efforts to improve support for structural dynamism in Modelica. The works by Nytsch-Geusen et al. [2005] and Zimmer [2008] are the most recent examples.

Nytsch-Geusen et al. [2005] designed and implemented a language extension to Modelica called MOSILAB. The language is extended with constructs allowing for description of statecharts specifying the discrete switches between Modelica objects. The statechart approach enables modelling of structurally dynamic systems. However, MOSILAB does not support unbounded structural dynamism because statecharts are required to be static and can not be extended at simulation runtime.

MOSILAB features a sophisticated compiled implementation producing efficient numerical simulation code for all modes of operation prior to simulation. This implementation approach works well for small number of modes. Simulation of bounded structurally dynamic systems with large number of modes is problematic.

Zimmer [2008] designed and implemented a Modelica-like language called Sol. The language introduces constructs allowing for description of noncausal models where equations and variables can be added and removed at runtime. Sol language solves many of the problems with the Modelica language outlined in this section. However, the increase in the language expressiveness comes at a cost of its efficiency. Currently, Sol only features an interpreted implementation. Compilation-based implementation approaches for Sol have not yet been explored. The ultimate goal of the work on Sol is to introduce its language features for structurally dynamic noncausal modelling in future versions of Modelica.

There are also a number of efforts to design and implement new structurally dynamic noncausal modelling languages. The works on HYBRSIM [Mosterman et al., 1998] and Acumen [Zhu et al., 2010] deserve a particular mention. Both languages take a very different language design approach from languages that are based on Modelica. However, when it comes to structurally dynamic modelling and simulation the expressive power of HYBRSIM and its interpreted implementation is comparable to that of Sol, while the expressive power of Acumen and its compiled implementation is comparable to that of MOSILAB.

To my knowledge none of the previous language design and implementation approaches support both unbounded structural dynamism and compilation for efficiency.

Chapter 3

FHM Concepts and Design of Hydra

This chapter introduces the three central concepts of the FHM framework that the Hydra language is based on, namely: signal, signal function and signal relation. These concepts facilitate development of and reasoning about Hydra models, and are used both in informal (see Chapter 4) and formal (see Chapter 5) presentations of the language. This chapter only covers conceptual definitions; how the concepts of Hydra are implemented is covered in Chapter 6. This chapter also discusses how Hydra's design facilitates the embedding of the aforementioned concepts into a functional programming language.

3.1   FHM and FRP

The idea of treating noncausal models as first-class values in a functional programming language is due to Nilsson et al. [2003]. The authors propose the FHM framework for designing and implementing noncausal modelling languages. The FHM framework borrows the notion of signal denoting time-varying values from the FRP languages and generalises the notion of signal function (featured in a number of variants of FRP, most notably Yampa [Nilsson et al., 2002, Hudak et al., 2003]) to signal relation.

Intuitively, a signal function can be understood as a block with inputs and outputs featured in causal modelling languages, while a signal relation can be understood as a noncausal model without explicitly specified inputs and outputs. In other words, FRP extends a functional programming language with causal modelling capabilities,

while FHM extends a functional programming language with noncausal modelling capabilities.

Signal functions are first-class entities in most variants of FRP and this property also carries over to signal relations in FHM. As we will see later in this chapter and also in Chapter 4, the first-class nature of signal relations is crucial for higher-order and structurally-dynamic modelling in Hydra. To my knowledge, Hydra is the first language that features the FHM's notion of signal relation as a first-class entity.

3.2   Signal

Conceptually, a *signal* is a time-varying value, that is, a function from time to value:

$$
\begin{aligned}
Time \quad &\approx \mathbb{R} \\
Signal\ \alpha &\approx Time \to \alpha
\end{aligned}
$$

Time is continuous and is represented as a real number. The type parameter α specifies the type of values carried by the signal; for example, a signal of type *Signal* $\mathbb{R}$ may represent a change to the total amount of current flowing in a certain electrical circuit over time, or a signal of type *Signal* $(\mathbb{R}, \mathbb{R})$ may represent a change in position of a certain object in a two dimensional space over time.

Hydra features signals of reals (i.e., *Signal* $\mathbb{R}$) and signals of arbitrarily nested pairs of reals. Signals of nested pairs are useful for grouping of related signals. As an example of a signal that carries nested pairs of reals, consider a signal of type *Signal* $((\mathbb{R}, \mathbb{R}), (\mathbb{R}, \mathbb{R}))$. This signal can represent current and voltage pairs at the positive and negative pins of a two-pin electrical component, for example.

In a concrete implementation, $\mathbb{R}$ would typically be represented by a suitable floating point type, such as *Double*. Indeed, *Double* is used in Hydra. However, $\mathbb{R}$ is used in most places of the presentation as we conceptually are dealing with real numbers.

The aforementioned treatment of signals as continues can be seen as an abstraction over the underling discretely sampled implementation. It is cumbersome and error prone to directly work with the discrete representation for entities that conceptually exhibit continuous dynamics. For example, explicit handling of fixed and variable sampling rates, composition of models using different sampling rates, and explicit accounting for numerical errors are problematic. This is not to say that being aware of the underling discretely sampled implementation is not important. In some cases

(e.g., when the numerical simulation fails or when the simulation performance is unacceptably slow) it is necessary to adjust the discrete simulation time step or change other simulation parameters. However, in most cases, assuming that the continuous model is correct, those adjustments can be made without changes to the model.

Although modelling with discrete streams for conceptually discrete systems (such as digital controllers) is out of the scope of this thesis, it would be interesting to explore a discrete variant of Hydra featuring noncausal equations on discrete streams.

3.3   Signal Function

Conceptually, a *signal function* is a function from signal to signal:

$$SF\ \alpha\ \beta \approx Signal\ \alpha \rightarrow Signal\ \beta$$

A signal function of type $SF\ \alpha\ \beta$ can be applied to an input signal of type $Signal\ \alpha$; it produces an output signal of type $Signal\ \beta$.

Because a pair of signals, say $(Signal\ \alpha, Signal\ \beta)$, is isomorphic to a signal of the pair of the signal types, in this case $Signal\ (\alpha, \beta)$, unary signal functions suffice for handling signal functions of any arity; for example, the binary signal function *add* that takes two signals and computes the sum of their values at each point in time can be given the following type and conceptual definition:

$$add :: SF\ (\mathbb{R}, \mathbb{R})\ \mathbb{R}$$
$$add\ s \approx \lambda t \rightarrow fst\ (s\ t) + snd\ (s\ t)$$

Hydra provides a number of primitive signal functions that lift common mathematical operations (e.g., $+$, $*$, *sin* and *cos*) to the signal level. Hydra also provides the *der* signal function of type $SF\ \mathbb{R}\ \mathbb{R}$. This signal function differentiates the given signal. Later we will see that the use of the *der* signal function in noncausal equations allows for the definition of differential equations.

It is worthwhile to note that except for the *der* signal function all primitive and user definable signal functions in Hydra are *stateless*; that is, their output only depends on their input at current point in time. The *der* signal function's output depends on its input signal's current rate of change.

3.4 Signal Relation

Conceptually, a *signal relation* is a relation on signals. Stating that some signals are in a particular relation to each other imposes *constraints* on those signals. Assuming these constraints can be satisfied, this allows some of the signals to be determined in terms of the others depending on which signals are known and unknown in a given context; that is, signal relations are noncausal, unlike signal functions where the inputs and outputs are given a priori.

An ordinary relation can be seen as a predicate that specifies whether some given values are related or not. The same is true for signal relations:

$$SR\ \alpha \approx Time \rightarrow Time \rightarrow Signal\ \alpha \rightarrow Prop$$

Given two points in time and a signal, a signal relation defines a proposition constraining the signal starting from the first point in time and ending with the second point in time. Here, *Prop* is a type for propositions defined in second-order logic. *Solving* a relation for a given period of time thus means finding a signal that satisfies the predicate. We say that in this period of time the signal relation instance is *active*.

Just like for signal functions, unary signal relations suffice for handling signal relations of any arity; for example, the following pseudo code conceptually defines a binary signal relation:

$$equal :: SR\ (\mathbb{R}, \mathbb{R})$$
$$equal\ t_1\ t_2\ s \approx \forall\, t \in \mathbb{R}.\ t_1 \leqslant t \wedge t \leqslant t_2 \Rightarrow fst\ (s\ t) \equiv snd\ (s\ t)$$

This signal relation asserts that the first and second components of the signal s are equal from t_1 to t_2.

Let us consider a slightly more elaborate example of a signal relation. The following conceptual definition gives a signal relation imposing constraints characteristic to electrical components with two connectors (see Figure 3.1).

$$twoPin :: SR\ (((\mathbb{R}, \mathbb{R}), (\mathbb{R}, \mathbb{R})), \mathbb{R})$$
$$twoPin\ t_1\ t_2\ s \approx$$
$$\quad \forall\, t \in \mathbb{R}.\ t_1 \leqslant t \wedge t \leqslant t_2 \Rightarrow twoPinProp\ (s\ t)$$
where
$$twoPinProp\ (((p_i, p_v), (n_i, n_v)), u) \approx \quad p_v - n_v = u$$
$$\wedge\ p_i + n_i = 0$$
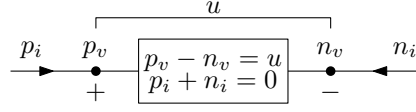
Figure 3.1: Electrical component with two connectors.

Here, p_i, p_v, n_i, n_v and u are the components of the tuple carried by the signal s. The tuple components p_i and p_v represent the current into the positive pin and the voltage at the positive pin, respectively. The tuple components n_i and n_v represent the current into the negative pin and the voltage at the negative pin, respectively. The tuple component u represents the voltage drop across the electrical component.

By *applying* signal relations to signals (in the sense of predicate application) it is possible to reuse the equational constraints. Signal relation application also allows for definition of hierarchically structured systems of equations. The following conceptual definition gives a signal relation imposing constrains characteristic to a resistor (with resistance r).

$$resistor :: \mathbb{R} \to SR\,(((\mathbb{R}, \mathbb{R}), (\mathbb{R}, \mathbb{R})), \mathbb{R})$$
$$resistor\ r\ t_1\ t_2\ s \approx$$
$$\forall\, t \in \mathbb{R}.\ t_1 \leqslant t \wedge t \leqslant t_2 \Rightarrow \exists\, u \in Signal\,\mathbb{R}.\quad twoPin\ t_1\ t_2\ (pairS\ s\ u)$$
$$\wedge\ resistorProp\ (s\ t)\ (u\ t)$$

where
$$pairS\ s\ u\ t' \qquad\qquad\qquad \approx (s\ t', u\ t')$$
$$resistorProp\ ((p_i, p_v), (n_i, n_v))\ u \approx r * p_i = u$$

Note how the signal relation *resistor* is defined in terms of the signal relation application of *twoPin* and one additional equation. As we will see in the next section and in Chapter 4, Hydra provides a convenient syntax for defining and applying signal relations.

3.5   Design of Hydra

Hydra is a two-level language. It features the *functional level* and the *signal level*. The functional level allows for the definition of ordinary functions operating on time-invariant values. The signal level allows for the definition of signal relations and signal functions on time-varying values.

Signal relations and signal functions are first-class entities at the functional level. In contrast, signals are not first-class entities at the functional level. However, crucially, instantaneous values of signals can be passed to the functional level, allowing for the generation of new signal relations that depend on signal values at discrete points in time.

A definition 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 signal-level equations are concerned. However, the opposite is not allowed; that is, a time-varying entity is confined to the signal level. The signal-level notions that exist at the functional level are signal relation and signal function. These notions are time-invariant. Concrete examples of signal-level and functional-level definitions as well as definitions where these two levels interact with each other are given in Chapter 4.

Hydra is implemented as a Haskell-embedded DSL using quasiquoting, a Haskell extension implemented in GHC, for providing a convenient surface syntax[1]. As a result, Haskell provides the functional level for free through shallow embedding. In contrast, the signal level is realised through deep embedding; that is, signal relations expressed in terms of Hydra-specific syntax are, through the quasiquoting machinery, turned into an internal representation, an abstract syntax tree (AST), that then is used for compilation into simulation code (see Chapter 6 for details). Note that, although Hydra is embedded in Haskell, the two-level language design outlined earlier in this section and the notion of first-class signal relations are not predicated on the embedding approach.

The Haskell-embedded implementation of Hydra adopts the following syntax for defining signal relations:

$$[rel \mid pattern \rightarrow equations \mid]$$

The symbol $[rel \mid$ is the opening quasiquote and the symbol $\mid]$ is the closing quasiquote. The pattern binds *signal variables* that scope over the equations that follow. An equation can be an equality constraint or a *signal relation application* (stated by using the operator $\diamond$). Signal relation application is how the constraints embodied by a signal relation are imposed on particular signals. In addition to the signal variables bound in the pattern, equations may also introduce local signal variables. Concrete examples of signal relations are given in Chapter 4.

---

[1]Quasiquoting is not unique to Haskell. It has been available in other languages (most notably in the Lisp family of languages).

The equations are required to be well typed. For example, consider the signal relation application $sr \diamond s$. Here, if sr has the type $SR\ \alpha$ then s must have the type $Signal\ \alpha$.

Hydra provides a conventional syntax for specifying equality constraints. For example, the equation $x * y = 0$ is an equality constraint. Here, 0 is a constant signal, $*$ is a primitive signal function, and x and y are signal variables.

In addition to user-defined signal relations, Hydra provides for user-defined signal functions. Hydra uses the following syntax for defining signal functions.

$$[fun \mid pattern \rightarrow expression \mid]$$

Just like for signal relations, quasiquoting is used for defining signal functions. The pattern binds signal variables that scope over the expression that follows. Signal functions can be applied to signals by juxtaposing them together:

$$sf\ s$$

Signal function applications are required to be well typed. In this example, if sf has the type $SF\ \alpha\ \beta$ then s must have the type $Signal\ \alpha$. The type of the resulting signal is $Signal\ \beta$.

The quasiquotes, in addition to serving as an embedded DSL implementation tool, can be seen as clear syntactic markers separating the signal level from the functional level. These markers are useful when reading Hydra code listings. The separation is also enforced at the type level of the host language by the SR and SF type constructors.

Because signals are not first-class entities at the functional level, it is not possible to construct a value of type $Signal\ \alpha$ directly at the functional level. Signals only exist indirectly through the signal level definitions of signal relations and signal functions.

Equality constrains can be used to describe flat systems of equations and the signal relation application operator (i.e., $\diamond$) provides for hierarchically structured systems of equations. Let us introduce a built-in (higher-order) signal relation that allows for description of structurally dynamic signal relations.

$$switch :: SR\ a \rightarrow SF\ a\ \mathbb{R} \rightarrow (a \rightarrow SR\ a) \rightarrow SR\ a$$

The *switch* combinator forms a signal relation by temporal composition. The combinator takes three arguments and returns the composite signal relation (of type $SR\ a$). The first argument (of type $SR\ a$) is a signal relation that is initially active.

The second argument is a signal function (of type $SF\ a\ \mathbb{R}$). Starting from the first point in time when the signal (of type $Signal\ \mathbb{R}$) that is computed by applying the signal function to the signal constrained by the composite signal relation is about to cross zero (i.e., when it is zero and its left derivative is nonzero), the composite behaviour is defined by the signal relation that is computed by applying the third argument (a function of type $a \rightarrow SR\ a$) to the instantaneous value of the constrained signal at that point in time. A formally defined meaning of the *switch* combinator is given in Chapter 5.

The *switch* combinator allows for definition of a signal relation whose equational description changes over time. In addition, the *switch* combinator allows for state transfer from the old mode and initialisation of the new mode using the function that computes the new mode from an instantaneous value of the constrained signal.

In the signal relation notation described earlier, the list of equations that follows the pattern is not necessarily a static one as the equations may contain a signal relation application of a structurally dynamic signal relation. We show how to use the *switch* combinator for modelling and simulation unbounded structurally dynamic systems in Chapter 4.

The two-level nature of Hydra also manifests itself in its implementation as a mixed-level embedding. The functional level and the signal-level notions that exist at the functional level are realised by the shallow part of the embedding. The notions that exist only at the signal level are realised by the deep part of the embedding. Specifically, we use shallow embedding techniques to represent signal relations (including higher-order signal relations), signal functions, signal relation applications, signal function applications, hierarchical compositions of signal relations and temporal compositions of signal relations; and we use deep embedding techniques to represent signal expressions and equality constraints on signal expressions. Note that in Section 6.1 we used more general terminology referring to the parts of Hydra realised through shallow and deep embeddings as high-level, symbolic and low-level, numerical, respectively.

The aforementioned combination of the two embedding techniques allowed us to maximise the reuse of the host language features and, as a result, to simplify the language implementation. As performance is a primary concern in the domain of physical modelling and simulation, the numerical simulation code for each mode of the model has to be compiled. As we will see in Chapter 6, a mode of operation is represented as equality constraints on signal expressions and zero-crossing signal expressions. The aforementioned need for compilation into efficient numerical simulation code neces-

sitates the use of deep embedding techniques for representing signal expressions and equality constraints on signal expressions. For the rest of the language we use the shallow embedding for maximum leverage of the host language.

Hydra's two-level design can also be realised as a deep embedding or as a standalone implementation. The particular combination of the embedding techniques used in this work reflects the fact that advanced symbolic processing of hierarchical systems of equations, beyond producing a flat list of equations for each mode of operation (for which shallow embedding techniques suffice), is not the main focus of this work. Sol [Zimmer, 2008] and Acumen [Zhu et al., 2010] are examples of noncausal languages that feature implementations with more involved symbolic processing of hierarchical systems of equations in order to solve problems that are orthogonal to those treated in this thesis (see Chapter 7 for details).

Chapter 4

Modelling and Simulation in Hydra

This chapter presents the Hydra language informally, by means of instructive examples. The examples are carefully chosen to back up the contributions of the thesis by illustrating higher-order and structurally dynamic modelling and simulation in Hydra.

4.1   Models with Static Structure

Let us illustrate the Hydra language by modelling the circuit that is depicted in Figure 2.1. Let us first define the *twoPin* signal relation that captures the common behaviour of electrical components with two connectors (see Figure 3.1):

$$\textbf{type } Pin = (\mathbb{R}, \mathbb{R})$$

$$twoPin :: SR\ ((Pin, Pin), \mathbb{R})$$

$$twoPin = [\,rel \mid (((p_i, p_v), (n_i, n_v)), u) \rightarrow$$

$$p_v - n_v = u$$

$$p_i + n_i = 0$$

$$|]$$

The signal variables p_i and p_v, which are bound in the pattern, represent the current into the positive pin and the voltage at the positive pin, respectively. The signal variables n_i and n_v, which are also bound in the pattern, represent the current into the negative pin and the voltage at the negative pin, respectively. The signal variable u represents the voltage drop across the electrical component.

We can now use the *twoPin* signal relation to define a signal relation that models a resistor:

$$resistor :: \mathbb{R} \rightarrow SR \; (Pin, Pin)$$
$$resistor \; r = [\, rel \mid ((p_i, p_v), (n_i, n_v)) \rightarrow$$
$$\quad \textbf{local} \; u$$
$$\quad \$ \; twoPin \; \$ \diamond(((p_i, p_v), (n_i, n_v)), u)$$
$$\quad \$ \; r \; \$ * p_i = u$$
$$\,|]$$

Note that a parametrised signal relation is an ordinary function returning a signal relation. In the *resistor* signal relation, the signal variable u is declared as a local signal variable; that is, it is not exposed in the pattern of the signal relation. As a consequence, u can only be constrained in this signal relation, unlike the rest of the variables in the pattern, which can be constrained further.

As we have already mentioned, Hydra uses two kinds of variables: the functional-level ones representing time-invariant entities, 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; that is, signal-variables are not in scope between antiquotes and outside the quasiquotes. Note that, as discussed in Section 3.5, a signal relation is a time-invariant entity and thus can be spliced into the signal level.

The *resistor* signal relation uses antiquoting to splice in a copy of the *twoPin* signal relation; that is, its equations are reused in the context of the resistor model. Readers familiar with object-oriented, noncausal languages like Modelica, can view this as a definition of the resistor model by extending the *twoPin* model with an equation that characterises the specific concrete electrical component, in this case Ohm's law.

To clearly see how *twoPin* contributes to the definition of the *resistor* signal relation, let us consider what happens when the resistor model is *flattened* as part of flattening of a complete model, a transformation that is described in detail in Chapter 6. Intuitively, flattening can be understood as inlining of applied signal relations to reduce the signal relation into a flat list of equations (i.e., a flat DAE). In the process of flattening, the arguments of a signal relation application are substituted into the body of the applied signal relation, and the entire application is then replaced by the instantiated signal relation body, renaming local variables as necessary to avoid name clashes. In our case, the result of flattening the signal relation *resistor* 42 is:

$$[\,rel \mid ((p_i, p_v), (n_i, n_v)) \rightarrow$$

$$\quad \textbf{local } u$$

$$\quad p_v - n_v = u$$

$$\quad p_i + n_i \; = 0$$

$$\quad 42 * p_i \; = u$$

$$|\,]$$

Models for an inductor, a capacitor, a voltage source and a ground reference are
defined in Figure 4.1. Note that the inductor and the capacitor signal relations contain
init equations. An **init** equation is enforced only at the point in time when the signal
relation becomes active. In this example, the **init** equations are used to initialise the
differential variables involved in the inductor and the capacitor signal relations.

By default, Modelica implicitly initialises differential variables to zero. That is
why initialisation equations were not considered in the corresponding Modelica mod-
els given in Chapter 2. Hydra does not allow for implicit initialisation; that is, all
initialisation equations must be specified explicitly.

4.2   Noncausal Connections

In Giorgidze and Nilsson [2008] we describe syntactic sugar for specifying noncausal
connections. In this thesis we implement the same approach using higher-order mod-
elling combinators. In both cases we are able to describe noncausal connections with-
out a special semantic language construct. In this aspect, Hydra is simpler than other
noncausal modelling languages such as Modelica, MKL [Broman, 2007], and Chi [Beek
et al., 2008], as these languages feature special language constructs for specifying non-
causal connections. It is worthwhile mentioning that although the aforementioned
approaches to noncausal connections serve the same purpose, they are very different
from each other, both in their syntax and in their semantics. Detailed comparison of
approaches to noncausal connections still lies ahead, including devising of a minimal
set of higher-order combinators expressive enough to capture all possible noncausal
interconnections.

Because signal relations are first-class entities, it is possible to implement higher-
order combinators that facilitate connection of noncausal models. To model the simple
electrical circuit as an interconnection of the already modelled components let us

$iInductor :: \mathbb{R} \to \mathbb{R} \to SR\ (Pin, Pin)$
$inductor\ p_{i_0}\ l = [\,rel \mid ((p_i, p_v), (n_i, n_v)) \to$
  local u
  init $p_i = p_{i_0}$
  $\$\ twoPin\ \$\ \diamond(((p_i, p_v), (n_i, n_v)), u)$
  $\$\ l\ \$ * der\ p_i = u$
$\,|\,]$

$iCapacitor :: \mathbb{R} \to \mathbb{R} \to SR\ (Pin, Pin)$
$iCapacitor\ u_0\ c = [\,rel \mid ((p_i, p_v), (n_i, n_v)) \to$
  local u
  init $u = u_0$
  $\$\ twoPin\ \$\ \diamond(((p_i, p_v), (n_i, n_v)), u)$
  $\$\ c\ \$ * der\ u = p_i$
$\,|\,]$

$vSourceAC :: \mathbb{R} \to \mathbb{R} \to SR\ (Pin, Pin)$
$vSourceAC\ v\ f = [\,rel \mid ((p_i, p_v), (n_i, n_v)) \to$
  local u
  $\$\ twoPin\ \$\ \diamond(((p_i, p_v), (n_i, n_v)), u)$
  $u = \$v\ \$ * sin\ (2 * \$\pi\ \$ * \$f\ \$ * time)$
$\,|\,]$

$ground :: SR\ (Pin)$
$ground = [\,rel \mid (p_i, p_v)\ \textbf{where}$
  $p_v = 0$
$\,|\,]$

Figure 4.1: Hydra models for inductor, capacitor, voltage source and ground reference.
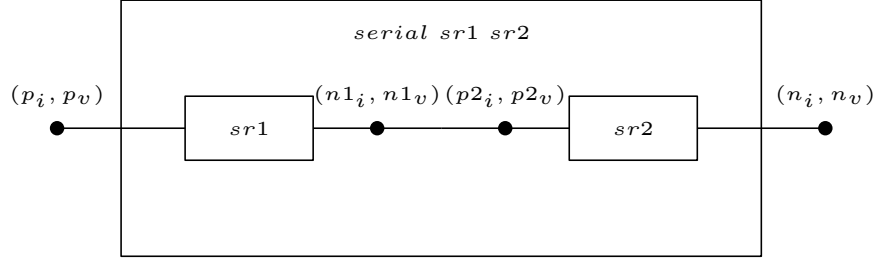
Figure 4.2:   Serial connection of two electrical components.

define three higher-order signal relations facilitating noncausal connection of two-pin electrical components.

Firstly, we define a higher-order signal relation that takes two signal relations modelling two-pin electrical components and returns the signal relation that models the serial connection of the two electrical components. The graphical representation of the signal relation is given in Figure 4.2.

$$serial :: SR\,(Pin, Pin) \to SR\,(Pin, Pin) \to SR\,(Pin, Pin)$$

$$serial\ sr1\ sr2 = [\,rel \mid ((p_i, p_v), (n_i, n_v)) \to$$

$$\textbf{local } p1_i; \textbf{local } p1_v; \textbf{local } n1_i; \textbf{local } n1_v;$$

$$\$\ sr1\ \$\ \diamond\,((p1_i, p1_v), (n1_i, n1_v))$$

$$\textbf{local } p2_i; \textbf{local } p2_v; \textbf{local } n2_i; \textbf{local } n2_v;$$

$$\$\ sr2\ \$\ \diamond\,((p2_i, p2_v), (n2_i, n2_v))$$

$$(-p_i) + p1_i = 0$$

$$p_v = p1_v$$

$$n1_i + p2_i = 0$$

$$n1_v = p2_v$$

$$n2_i + (-n_i) = 0$$

$$n2_v = n_v$$

$$\,|\,]$$

Secondly, we define a higher-order signal relation that takes two signal relations modelling two-pin electrical components and returns the signal relation that models the parallel connection of the two electrical components. The graphical representation of the signal relation is given in Figure 4.3.
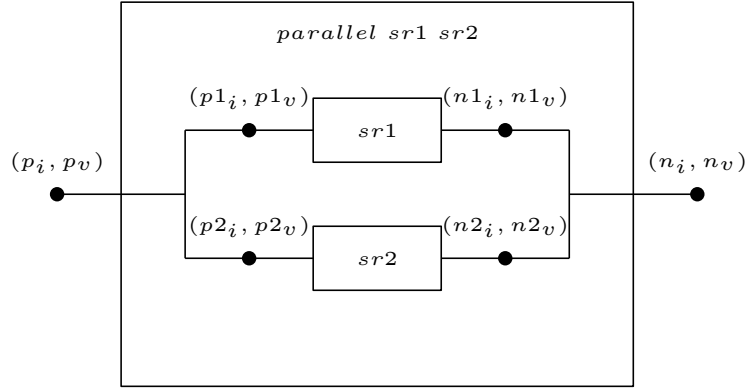
Figure 4.3: Parallel connection of two electrical components.

$parallel :: SR\ (Pin, Pin) \to SR\ (Pin, Pin) \to SR\ (Pin, Pin)$

$parallel\ sr1\ sr2 = \lceil rel \mid ((p_i, p_v), (n_i, n_v)) \to$

   local $p1_i$; **local** $p1_v$; **local** $n1_i$; **local** $n1_v$;

   $\$\ sr1\ \$\ \diamond ((p1_i, p1_v), (n1_i, n1_v))$

   local $p2_i$; **local** $p2_v$; **local** $n2_i$; **local** $n2_v$;

   $\$\ sr2\ \$\ \diamond ((p2_i, p2_v), (n2_i, n2_v))$

   $(-p_i) + p1_i + p2_i = 0$

   $p_v = p1_v$

   $p1_v = p2_v$

   $(-n_i) + n1_i + n2_i = 0$

   $n_v = n1_v$

   $n1_v = n2_v$

$\rfloor$

Finally, we define a higher-order signal relation that takes two signal relations modelling two-pin electrical components and returns the signal relation that models the grounded circuit involving the two electrical components. The graphical representation of the signal relation is given in Figure 4.4.
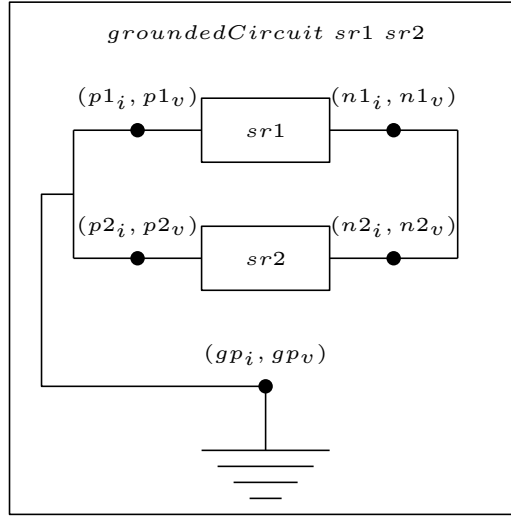
Figure 4.4: Grounded circuit involving two electrical components.

$groundedCircuit :: SR\ (Pin, Pin) \rightarrow SR\ (Pin, Pin) \rightarrow SR\ ()$

$groundedCircuit\ sr1\ sr2 = \lceil rel \mid () \rightarrow$

    local $p1_i$; **local** $p1_v$; **local** $n1_i$; **local** $n1_v$;

    \$ $sr1$ \$ $\diamond ((p1_i, p1_v), (n1_i, n1_v))$

    local $p2_i$; **local** $p2_v$; **local** $n2_i$; **local** $n2_v$;

    \$ $sr2$ \$ $\diamond ((p2_i, p2_v), (n2_i, n2_v))$

    local gp_i; **local** gp_v;

    \$ $ground$ \$ $\diamond (gp_i, gp_v)$

    $p1_i + p2_i = 0$

    $p1_v + p2_v = 0$

    $n1_i + n2_i + gp_i = 0$

    $n1_v = n2_v$

    $n2_v = gp_v$

  $\rfloor$

Now we can assemble the models of the electrical components into the simple electrical circuit as follows.

$$simpleCircuit :: SR~()$$
$$simpleCircuit =$$
$$groundedCircuit~(vSourceAC~1~1)$$
$$(parallel~(serial~(resistor~1)~(iCapacitor~0~1))$$
$$(iInductor~0~1))$$

Here the state variables are initially set to zero and all other parameters are set to one. Note that the above code is a direct textual representation of how the components are connected in the circuit. Unlike the Modelica model that specifies the noncausal connections in terms of connections of time-varying variables, Hydra allows for definition of higher-order combinators that are capable of specifying noncausal connections by connecting noncausal models directly.

It is trivial in Hydra to reuse the circuit components and model the modified circuit that is depicted in Figure 2.6:

$$simpleCircuit2 :: SR~()$$
$$simpleCircuit2 =$$
$$groundedCircuit~(vSourceAC~1~1)$$
$$(parallel~(serial~(resistor~1)~(iCapacitor~0~1))$$
$$(serial~(resistor~1)~(iInductor~~0~1)))$$

4.3   Higher-order Modelling with Collections of Models

We have already seen several higher-order models; for example, the *serial*, *parallel* and *groundedCircuit* signal relations. This section considers more higher-order modelling examples, but this time concentrating on signal relations that are parametrised on collections of signal relations. In addition, this section puts an emphasis on how the host, higher-order functional language can provide expressive facilities for higher-order, noncausal modelling.

Let us define a higher-order signal relation that takes as an argument a list of signal relations modelling two-pin electrical components and returns the signal relation that models serial connection of the given electrical components. This would be useful to model electronic transmission lines, for example.
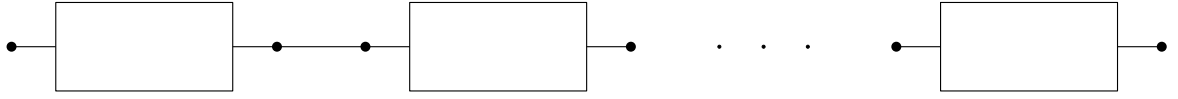
Figure 4.5: Serial connection of electrical components.

$$serialise :: [\,SR\,(Pin, Pin)\,] \rightarrow SR\,(Pin, Pin)$$
$$serialise = foldr\ serial\ wire$$

The definition of the *serialise* signal relation begs for detailed explanation. The *foldr* function is defined in the standard Haskell prelude and has the following type signature and definition.

$$foldr :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [\,a\,] \rightarrow b$$
$$foldr\ \_\ z\ [\,] = \qquad z$$
$$foldr\ f\ z\ (x : xs) = f\ x\ (foldr\ f\ z\ xs)$$

The function takes as arguments a binary operator, a starting value that is typically the right-identity of the binary operator, and a list. The *foldr* function folds the list using the binary operator, from right to left:

$$foldr\ serial\ wire\ [\,sr1, sr2, ..., srn\,] =$$
$$\quad sr1\ `serial`\ (sr2\ `serial`\ ...(srn\ `serial`\ wire)...)$$

Here the higher-order signal relation *serial* is in the role of a binary operator and the *wire* signal relation is in the role of a starting value which is a right identity of the binary operator. Figure 4.5 graphically demonstrates the result of this application of the *foldr* function.

The *wire* signal relation models an electrical wire and is defined as follows.

$$wire :: SR\,(Pin, Pin)$$
$$wire = [\,rel\ |\ ((p_i, p_v), (n_i, n_v)) \rightarrow$$
$$\quad \$\ twoPin\ \$ \diamond((p_i, p_v), (n_i, n_v), u)$$
$$\quad u = 0$$
$$|\,]$$

Just like other two-pin electrical components, the *wire* signal relation is modelled by extending the *twoPin* signal relation with a suitable equation.

The *wire* signal relation is both left and right identity of the *serial* higher-order signal relation as stated by the following equation and illustrated in Figure 4.6.

56

Figure 4.6: The *wire* signal relation as a left and right identity of the *serial* higher-order signal relation.

$$wire\ `serial`\ sr = sr\ `serial`\ wire = sr$$

The *serial* signal relation is associative:

$$sr1\ `serial`\ (sr2\ `serial`\ sr3) = (sr1\ `serial`\ sr2)\ `serial`\ sr3$$

Here by the equality of the signal relations we mean that the signal relations introduce equivalent constraints (i.e., one constraint implies the other and vice versa), and not necessarily the same equations. Because the *wire* signal relation is both left and right identity of the *serial* binary function and the *serial* signal relation is associative, in the definition of the *serialise* signal relation we could also use the left fold instead of the right fold.

Somewhat similarly to the *serialise* signal relation the higher-order signal relation *parallelise* that takes as an argument a list of signal relations modelling two-pin electrical components and returns the signal relation that models parallel connection of the given electrical components can be defined as follows.

$$parallelise :: [SR\ (Pin, Pin)] \rightarrow SR\ (Pin, Pin)$$
$$parallelise = foldr\ parallel\ noWire$$

The *noWire* signal relation is defined as follows.

$$noWire :: SR\ (Pin, Pin)$$
$$noWire = [rel\ |\ ((p_i, p_v), (n_i, n_v)) \rightarrow$$
$$\$\ twoPin\ \$ \diamond((p_i, p_v), (n_i, n_v), u)$$
$$p_i = 0$$
$$|]$$

The *noWire* signal relation is both left and right identity of the *parallel* higher-order signal relation as stated by the following equation and illustrated in Figure 4.7.

$$noWire\ `parallel`\ sr = sr\ `parallel`\ noWire = sr$$
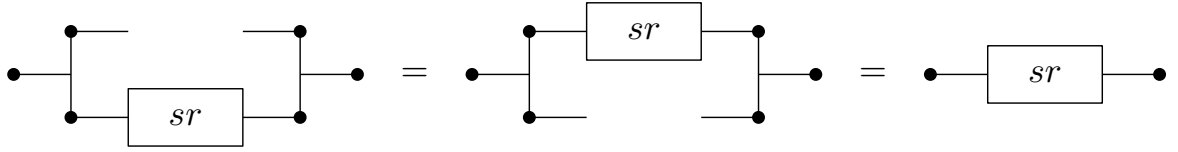
Figure 4.7: The *noWire* signal relation as a left and right identity of the *parallel* higher-order signal relation.

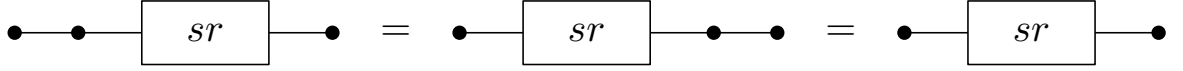The *parallel* signal relation is associative:

$$sr1 \text{ `parallel` } (sr2 \text{ `parallel` } sr3) = (sr1 \text{ `parallel` } sr2) \text{ `parallel` } sr3$$

Because the *noWire* signal relation is both left and right identity of the *parallel* binary function and the *parallel* signal relation is associative, in the definition of the *parallelise* signal relation we could also use the left fold instead of the right fold.

4.4 Structurally Dynamic Modelling

For a concrete example of structurally dynamic modelling in Hydra, let us model the breaking-pendulum system described in Section 2.7. The system system has two modes of operation. The differences between the two modes are sufficiently large that, for example, Modelica does not support noncausal modelling of this system, as discussed in Section 2.7.

The code that is given in Figure 4.8 shows how to model the two modes of the pendulum in Hydra. The type *Body* denotes the state of the pendulum body; that is, its position and velocity, where position and velocity both are 2-dimensional vectors represented by pairs of reals. Each model is represented by a function that maps the parameters of the model to a relation on signals. 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. Once again, note that the equations that are marked by the keyword **init** are initialisation equations used to specify initial conditions.

To model a pendulum that breaks at some point, we need to create a composite signal relation where the signal relation that models the dynamic behaviour of the unbroken pendulum is replaced, at the point when it breaks, by the signal relation modelling 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.

```
type Pos  = (ℝ, ℝ)
type Vel  = (ℝ, ℝ)
type Body = (Pos, Vel)

g :: ℝ
g = 9.81

freeFall :: Body → SR Body
freeFall ((x0, y0), (vx0, vy0)) = [ rel | ((x, y), (vx, vy)) →
    init (x, y)      = ($x0$, $y0$)
    init (vx, vy)    = ($vx0$, $vy0$)
    (der x, der y)   = (vx, vy)
    (der vx, der vy) = (0, −$g$)
 |]

pendulum :: ℝ → ℝ → SR Body
pendulum l φ₀ = [ rel | ((x, y), (vx, vy)) →
    local φ

    init φ       =    $ φ₀ $
    init der φ =    0
    init vx    =    0
    init vy    =    0

    x          =        $ l $ * sin φ
    y          =    −  $ l $ * cos φ
    (vx, vy)   =    (der x, der y)
    der (der φ) + ($g / l$) * sin φ = 0
 |]
```

Figure 4.8:   Signal relations modelling the two modes of the pendulum.
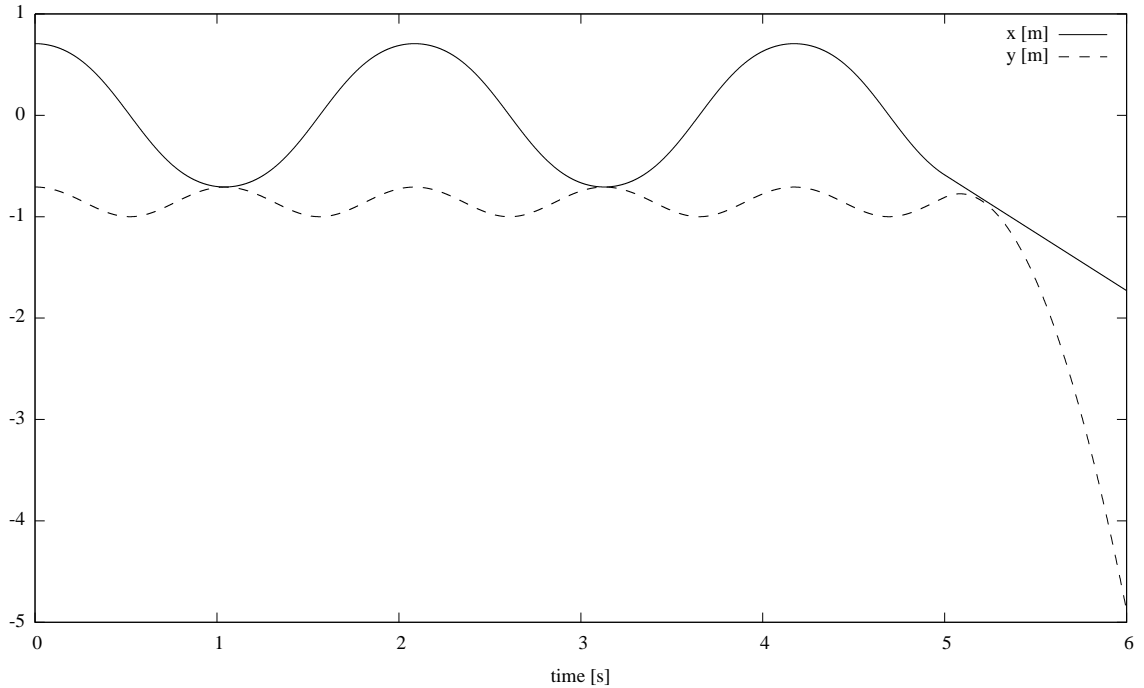
Figure 4.9: Plot showing how x and y coordinates of the body on the breaking pendulum change over time.

To this end, the *switch* combinator is used:

> *breakingPendulum* :: $\mathbb{R} \rightarrow \mathbb{R} \rightarrow \mathbb{R} \rightarrow$ *SR Body*
> *breakingPendulum t l phi0* =
>   *switch* (*pendulum l phi0*) [*fun* | $\lambda\_ \rightarrow$ *time* $-$ \$t\$ |] ($\lambda b \rightarrow$ *freeFall b*)

In this signal relation, the switch happens at an *a priori* specified point in time, but a switching condition can be derived from an arbitrary time-varying entity. Note how the succeeding signal relation (i.e., *freeFall*) is initialised so as to ensure the continuity of the position and velocity as discussed above. The simulation results obtained by the *simulate* function can be seen in Figure 4.9

In the breaking pendulum example the *switch* combinator was used to dynamically add and remove signal variables and noncausal equations from the model. The *switch* combinator can also be used when the number of equations and variables remain unchanged during the simulation. The book by Cellier and Kofman [2006] gives one such example: the half-wave rectifier circuit with an ideal diode and an in-line inductor that is depicted in Figure 4.10.
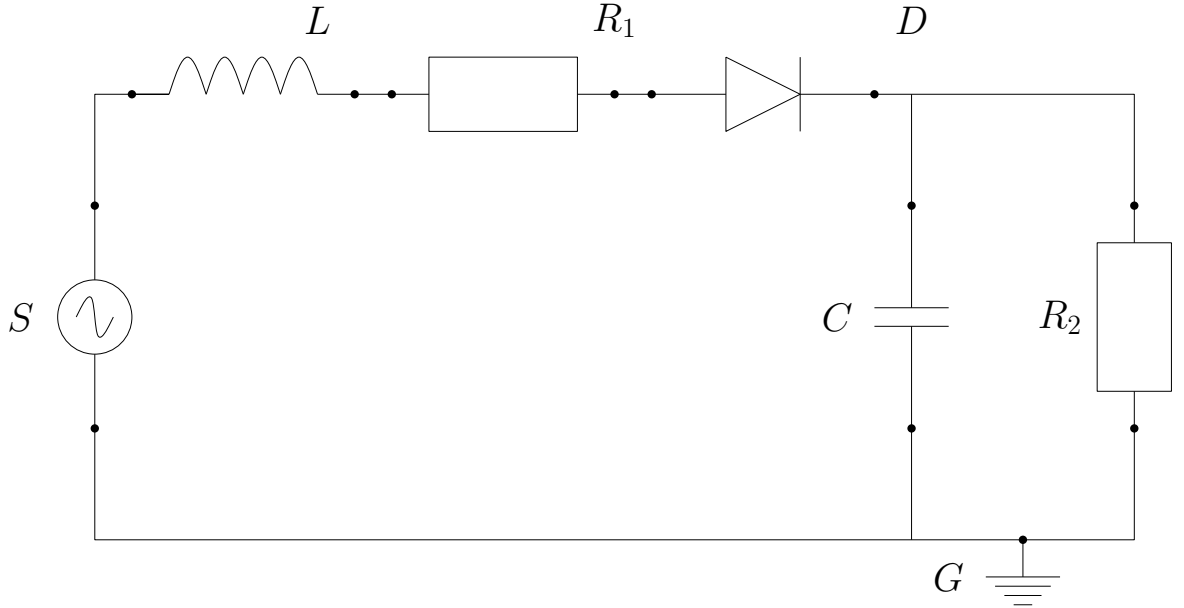
Figure 4.10: Half-wave rectifier circuit with an ideal diode and an in-line inductor.

The half-wave rectifier circuit can be modelled easily in languages like Modelica. However, any attempt to simulate this model assuming fixed causality, as current mainstream noncausal language implementations tend to, will fail as the causalised model will lead to a division by zero when the switch is open: there simply is no one fixed causality model that is valid both when the switch is open and closed.

One common solution to the division-by-zero problem is to avoid the ideal model and opt for a leaky diode model instead. This works, but often leads to very stiff equations. Thus, if an ideal model would suffice for the purpose at hand, that would be preferable [Cellier and Kofman, 2006].

Let us model the half-wave rectifier circuit in Hydra. The following two signal relations model initially opened (*ioDiode*) and initially closed (*icDiode*) ideal diodes. Note the use of the host language feature of mutual recursion in the following definitions allowing for signal relations to switch into each other.

$ioDiode :: SR\ (Pin, Pin)$
$ioDiode = switch\ nowire\ [fun\ |\ ((\_, p_v), (\_, n_v)) \to p_v - n_v\ |]\ (\lambda\_ \to icDiode)$
$icDiode :: SR\ (Pin, Pin)$
$icDiode = switch\ wire\quad [fun\ |\ ((p_i, \_), (\_, \_))\quad \to p_i\qquad |]\ (\lambda\_ \to ioDiode)$

61
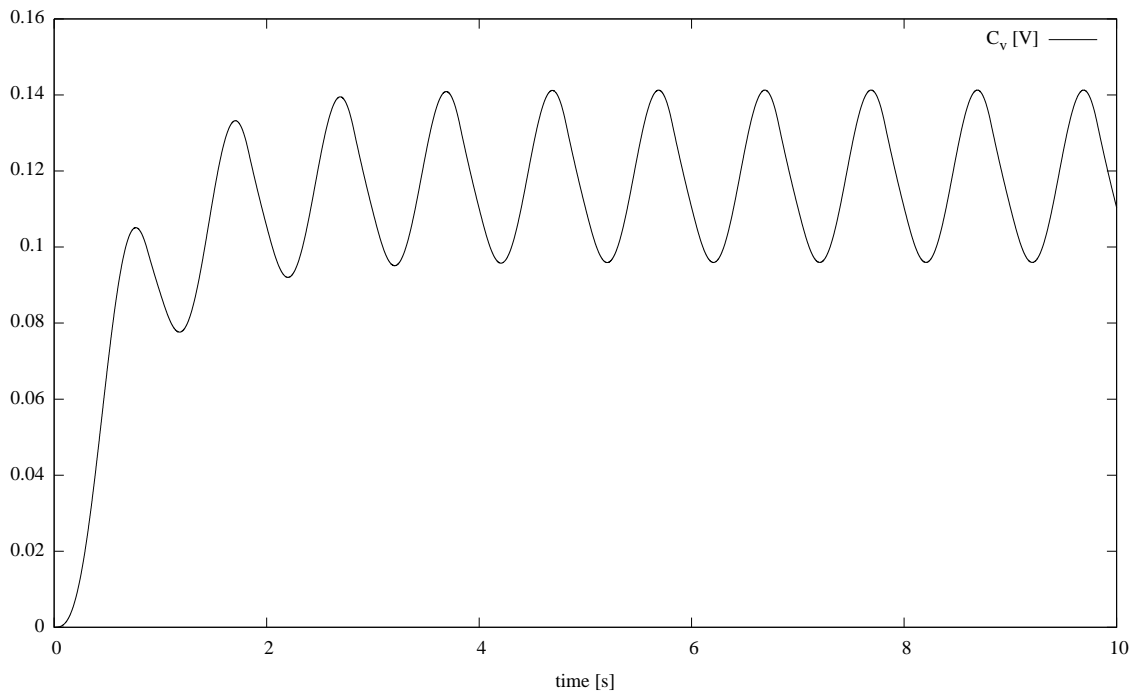
Figure 4.11: Voltage across the capacitor in the half-wave rectifier circuit with in-line inductor.

The switches are controlled by the polarity of the voltage and the current through the component. Now we can assemble the half-wave rectified circuit into a single signal relation using the higher-order connection combinators defined earlier in this chapter.

$halfWaveRectifier :: SR ()$
$halfWaveRectifier =$
    $groundedCircuit (vSourceAC\ 1\ 1)$
                    $(serialise\ [iInductor\ 0\ 1$
                            $, resistor\ 1$
                            $, icDiode$
                            $, parallel\ (iCapacitor\ 0\ 1)\ (resistor\ 1)$
                            $]$
                    $)$

Partial simulation results of the *halfWaveRectifier* signal relation obtained by using the *simulate* function are presented in Figure 4.11 and in Figure 4.12

Simulation of the full-wave rectifier circuit given in Figure 4.13 is more challenging than simulating the half-wave rectifier [Nilsson and Giorgidze, 2010]. A key difficulty

62
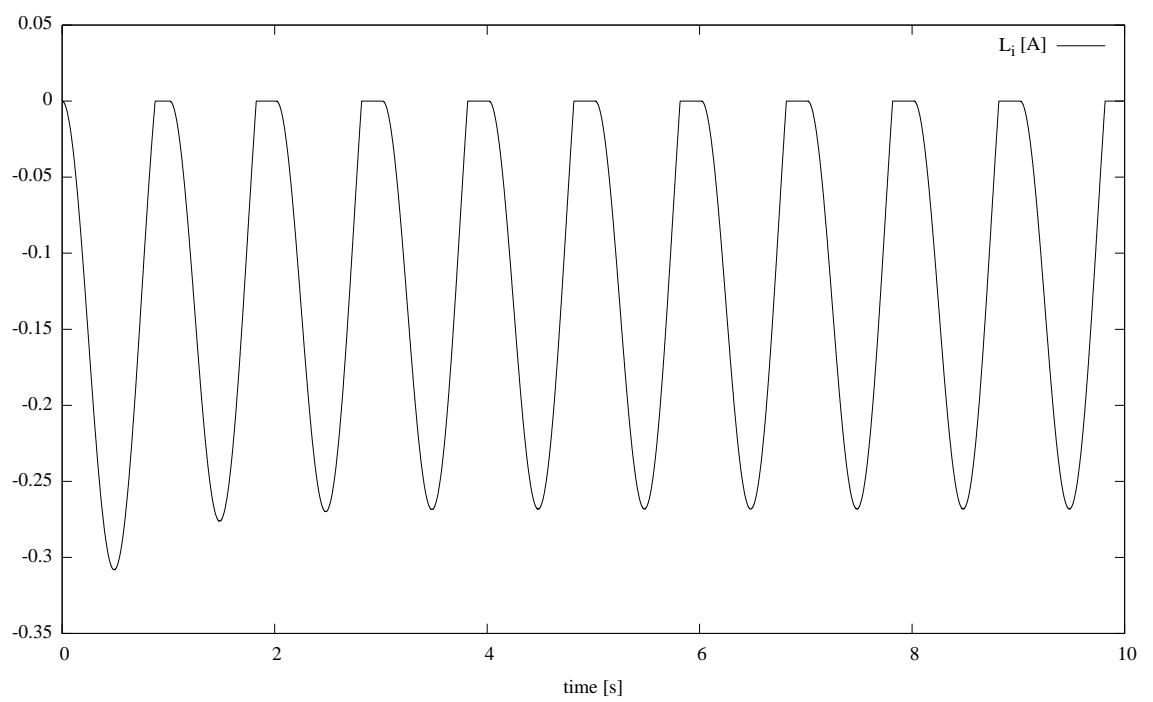
Figure 4.12: Current through the inductor in the half-wave rectifier circuit with in-line inductor.
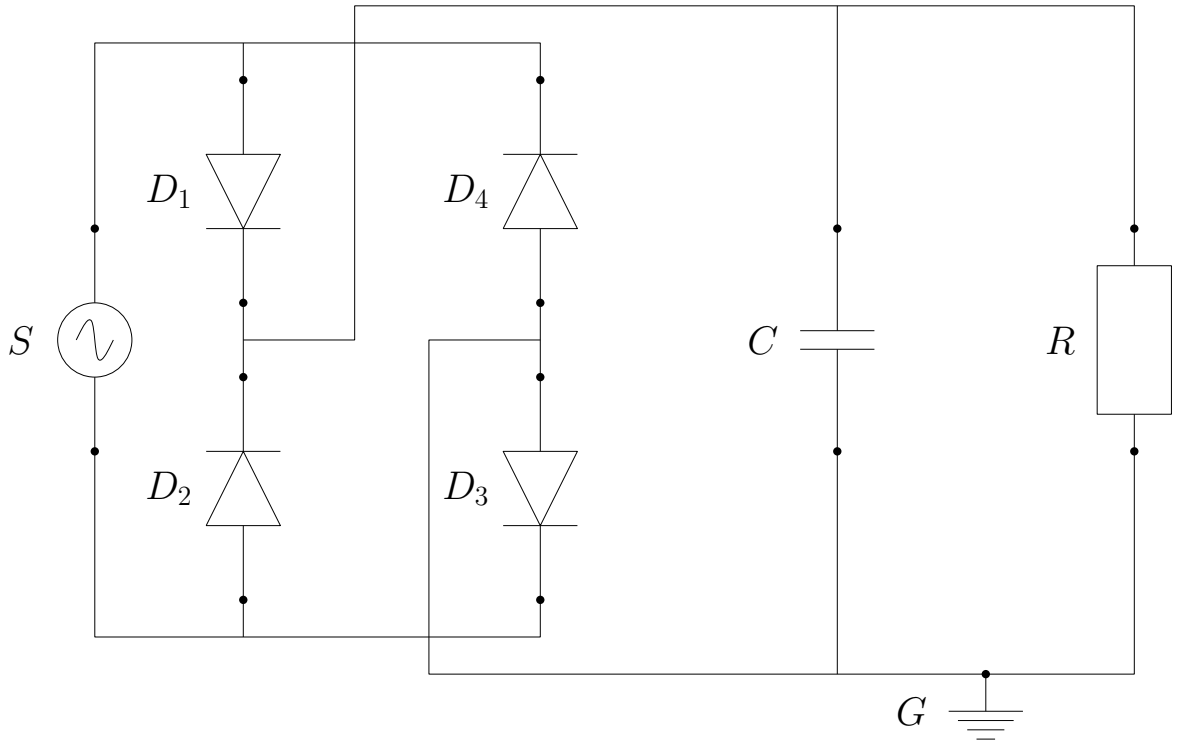
Figure 4.13: Full-wave rectifier circuit with ideal diodes.

is that the circuit breaks down into two isolated halves when all diodes are open. The lack of a ground reference for the left part means the system becomes under-determined and it cannot be simulated.

However, a more detailed analysis reveals that this is as it should be as the model is incomplete: for the model to make sense, there is further tacit modelling knowledge that needs to be stated explicitly in the form of additional equations. If the diodes are truly ideal, this means that they are also identical, which in turn implies that the voltage drops over them are always going to be pairwise equal, even when they are open. The model for the full circuit can be described along the lines we saw in the previous section, except that two extra equations, stating the pairwise equality of the voltages across the diodes, are needed. That is:

$$(dp_{v_1} - dn_{v_1}) = (dp_{v_3} - dn_{v_3}) \tag{4.1}$$
$$(dp_{v_2} - dn_{v_2}) = (dp_{v_4} - dn_{v_4}) \tag{4.2}$$

However, adding Equation (4.1) and Equation (4.2) results in additional complications for simulation as the system now seemingly becomes over-determined when some diodes are closed. It turns out, though, that the system is only trivially over-determined; that is, the extra equations are equivalent to other equations in the system. This is easy to see: when a diode is closed, there is an equation provided by the model of the diode itself that states that the voltage across it is 0. If, for example, $D1$ and $D3$ are closed, we have:

$$(dp_{v_1} - dn_{v_1}) = 0 \qquad (4.3)$$
$$(dp_{v_3} - dn_{v_3}) = 0 \qquad (4.4)$$

But, additionally, $(dp_{v_1} - dn_{v_1})$ and $(dp_{v_3} - dn_{v_3})$ are related by Equation 4.1 that is provided by model of the overall circuit.

In this case, a simple symbolic simplification pass involving substitution of algebraic variables and constant folding suffices to eliminate the redundant equations in the modes where the diodes are pairwise closed. Using Equation 4.3 and Equation 4.4, Equation 4.1 can be simplified to the trivially satisfied equation $0 = 0$ that then can be eliminated. After this the model can be simulated without further issues. Note that dynamic generation of equations followed by symbolic processing, as provided by Hydra, is crucial to this approach to simulating ideal diodes. Studies to precisely characterise in which circumstances can over determined systems of equations simplified by eliminating the redundant equations still lie ahead.

The implementation of Hydra provides an extensible and configurable symbolic processor. The symbolic processor that simplifies trivially over-determined equations is provided with the implementation of Hydra and can be activated through the experiment description passed to the *simulate* function (see Section 4.6 and Section 6.2 for details).

It should be pointed out that changes caused by an instance of a switch only concern equations originating from that switch instance. All other equations remain as they were. Thus, even though, in the case of ideal diodes, a circuit with n diodes has up to 2^n distinct structural configurations or modes, it is always entirely clear which mode to move to after a switch; there is no need to search among the up to 2^n possibilities for a consistent successor mode.

As a result, we have obtained a model of an ideal full-wave rectifier that is constructed in a modular way from individual, reusable components. The proper behaviour emerges from simply assembling the components, with just some minor additional guidance from the modeller in the form of a couple of extra equations. There is no need for any heavyweight, auxiliary mechanisms, such as an explicit finite state machine, to control how the model moves between structural configurations.

4.5 Unbounded Structurally Dynamic Modelling

The breaking pendulum, half-wave rectifier and full-wave rectifier examples feature a priori bounded number modes of operation. In principle (with a suitable language design and implementation) it is feasible to generate code for these modes of operation prior to simulation. However, despite their simplicity, these are examples with which mainstream noncausal languages such as Modelica struggle, as mentioned earlier.

In general, it is not possible to compile Hydra models prior to simulation. For example, given a parametrised signal relation $sr' :: \mathbb{R} \to SR\ \mathbb{R}$ and a signal function $sf :: SF\ \mathbb{R}\ \mathbb{R}$ one can recursively define a signal relation sr that describes an overall behaviour by "stringing together" the behaviours described by sr':

$$sr :: \mathbb{R} \to SR\ \mathbb{R}$$
$$sr\ x = switch\ (sr'\ x)\ sf\ sr$$

In this case, because the number of instantiations of sr' 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.

Perhaps the example involving the sr signal relation is a bit abstract. In the following we emphasise the same point by using a variation on the familiar bouncing-ball example. Assuming elastic collision with the floor, the bouncing ball system can be modelled in Hydra as follows.

$$bouncingBall :: Body \to SR\ Body$$
$$bouncingBall\ b = switch\ (freeFall\ b)$$
$$[fun\ |\ ((\_, y), \_) \to y\ |]$$
$$(\lambda(p, (vx, vy)) \to bouncingBall\ (p, (vx, -vy))$$

This example involves stringing of the $bouncingBall$ signal relation. But even here, in principle, it is possible to generate the code prior to simulation, because the active equations always remain the same; that is, only the initial condition is changing.

The following code models a variation of the bouncing ball example where the ball breaks at every collision with the floor.

$$bouncingBall' :: Body \rightarrow SR\ Body$$
$$bouncingBall'\ b = switch\ (freeFall\ b)$$
$$[fun\ |\ ((\_, y), \_) \rightarrow y\ |\ ]$$
$$(\lambda(p, v) \rightarrow divide\ (p, v))$$
$$divide :: Body \rightarrow SR\ Body$$
$$divide\ ((x0, y0), (vx0, vy0)) = [\,rel\ |\ ((x, y), (vx, vy)) \rightarrow$$
$$\$\ bouncingBall'\ ((x0, y0), (\quad vx0\ /\ 2, -\ vy0\ /\ 2))\ \$\ \diamond ((x, y), (vx, vy))$$
$$\textbf{local}\ x'\ y'\ vx'\ vy'$$
$$\$\ bouncingBall'\ ((x0, y0), (-vx0\ /\ 2, -\ vy0\ /\ 2))\ \$\ \diamond ((x', y'), (vx', vy'))$$
$$|\ ]$$

The model assumes that the kinetic energy is not lost and the balls divide the initial kinetic energy by bouncing in opposite directions. This is an example of an unbounded structurally dynamic system where the number of modes cannot be determined prior to simulation.

4.6   Simulation

We conclude this chapter with a brief description of how to simulate Hydra models, including the ones that are described in this chapter. The Haskell-embedded implementation of Hydra features the following function:

$$simulate :: SR\ () \rightarrow Experiment \rightarrow IO\ ()$$

The *simulate* function takes two arguments. The first argument is the signal relation that needs to be simulated. The second argument describes the *experiment*, essentially a description of what needs to happen during the simulation. Using the second argument the modeller can set the simulation starting and ending times, desired time step, symbolic processor, numerical solver, or how to visualise the trajectories of the constrained signals. The definition of the *Experiment* data type and the default experiment description are given in Chapter 6.

For example, the simple circuit model can be simulated using the default experiment description as follows.

simulate simpleCircuit defaultExperiment

With the *defaultExperiment* parameter the *simpleCircuit* signal relation is simulated for 10 seconds of simulation time starting from the time point of zero. The time step is set to 0.001 and the trajectories of the constrained signals are printed to the standard output in the gnuplot[1] compatible format. The default numerical solver is SUNDIALS [Hindmarsh et al., 2005], but users are allowed to provide their own symbolic processors and numerical solvers. This and other implementation aspects are described in detail in Chapter 6.

---

[1] http://www.gnuplot.info/

Chapter 5

Definition of Hydra

This is a technical chapter giving a formal definition of the Hydra language. Note that this chapter defines Hydra's signal-level sublanguage. The functional-level sublanguage is provided by Haskell. The definition of Haskell is given in the book by Peyton Jones [2003].

The language definition is given in four steps. Firstly, we define Hydra's lexical structure and concrete syntax by using regular expression and BNF notations, respectively. Secondly, we give Hydra's untyped abstract syntax as a Haskell algebraic data type (ADT) definition. Thirdly, we define Hydra's typed abstract syntax as a Haskell generalised algebraic data type (GADT) [Peyton Jones et al., 2006] definition and give a translation from the untyped abstract syntax to the typed abstract syntax. The typed representation fully embodies Hydra's type system and can be seen as a definition of Hydra's type system in terms of the Haskell type system. In other words, Hydra's type system is embedded into Haskell's type system. Finally, we give ideal semantics of Hydra by giving meaning to the typed abstract syntax in terms of second-order logic.

Note that the first three steps are concerned with the quasiquoted part of Hydra (i.e., quasiquoted definitions of signal functions and signal relations). The fourth step, which gives the ideal semantics, treats the full signal level of Hydra including the *switch* combinator. This is because structurally dynamic signal relations are defined using the *switch* combinator and not the quasiquotes.

5.1 Concrete Syntax

The syntactic structure of Hydra is given in Figure 5.1, which uses BNF notation. Non-terminals are enclosed between ⟨ and ⟩. The symbols ::= (production), | (union) and ϵ (empty rule) belong to BNF notation. All other symbols are terminals[1].

Identifiers ⟨*Ident*⟩ are unquoted strings beginning with a letter, followed by any combination of letters, digits, and the characters _ and ', reserved words excluded. The reserved words used in Hydra are **init** and **local**.

Integer literals ⟨*Int*⟩ are nonempty sequences of digits. Double-precision float literals ⟨*Double*⟩ have the structure indicated by the regular expression ⟨*digit*⟩+ '.'⟨*digit*⟩+ ('e'-?⟨*digit*⟩+)?; that is, two sequences of digits separated by a decimal point, optionally followed by an unsigned or negative exponent.

HsExpr literals represent antiquoted Haskell expressions and are recognised by the regular expression '$'(⟨*anychar*⟩ − '$') ∗ '$'

The symbols used in Hydra are given in Figure 5.2. In Hydra, single-line comments begin with −− and multiple-line comments are enclosed with {− and −}.

5.2 Abstract Syntax

Hydra's abstract syntax is given in Figure 5.3. The ADT definition is derived from the concrete syntax defined in previous section. The representation is untyped; that is, it allows for terms that are syntactically correct but not necessarily type correct.

The data type *Ident* is used to represent identifiers, specifically, signal variable and built-in signal function identifiers. The data type *HsExpr* is used to represent antiquoted Haskell expressions.

The data type *SigRel* is used to represent signal relations. The data type has a single constructor. Given a pattern and a list of equations it constructs the corresponding signal relation.

The data type *SigFun* is used to represent signal functions. The data type has a single constructor. Given a pattern and a signal expression it constructs the corresponding signal function.

The data type *Pattern* is used to represent patterns that bind signal variables. There are four ways to construct a pattern. The constructor *PatWild* constructs the

---

[1]Note that in Chapter 3, Chapter 4 and Chapter 6 the terminal symbols −> and <> are typeset as → and ⋄ respectively.

$\langle SigRel \rangle$ ::= $\langle Pattern \rangle \; -> \; \{ \; \langle ListEquation \rangle \; \}$

$\langle SigFun \rangle$ ::= $\langle Pattern \rangle \; -> \; \{ \; \langle Expr \rangle \; \}$

$\langle Pattern \rangle$ ::=   _
              | $\langle Ident \rangle$
              | ()
              | ($\langle Pattern \rangle$, $\langle Pattern \rangle$)

$\langle Equation \rangle$ ::= $\langle Expr \rangle = \langle Expr \rangle$
              | init $\langle Expr \rangle = \langle Expr \rangle$
              | local $\langle Ident \rangle$
              | $\langle HsExpr \rangle <> \langle Expr \rangle$

$\langle Expr1 \rangle$ ::= $\langle Expr1 \rangle + \langle Expr2 \rangle$
              | $\langle Expr1 \rangle - \langle Expr2 \rangle$
              | $\langle Expr2 \rangle$

$\langle Expr2 \rangle$ ::= $\langle Expr2 \rangle$ / $\langle Expr3 \rangle$
              | $\langle Expr2 \rangle$ * $\langle Expr3 \rangle$
              | $\langle Expr3 \rangle$

$\langle Expr3 \rangle$ ::= $\langle Expr3 \rangle$ ^ $\langle Expr4 \rangle$
              | $- \langle Expr4 \rangle$
              | $\langle Expr4 \rangle$

$\langle Expr4 \rangle$ ::= $\langle Expr4 \rangle \; \langle Expr5 \rangle$
              | $\langle Expr5 \rangle$

$\langle Expr5 \rangle$ ::= $\langle Ident \rangle$
              | $\langle HsExpr \rangle$
              | $\langle Integer \rangle$
              | $\langle Double \rangle$
              | ()
              | ($\langle Expr \rangle$, $\langle Expr \rangle$)
              | ($\langle Expr \rangle$)

$\langle Expr \rangle$ ::= $\langle Expr1 \rangle$

$\langle ListEquation \rangle$ ::= ϵ
              | $\langle Equation \rangle$
              | $\langle Equation \rangle$; $\langle ListEquation \rangle$

Figure 5.1: Syntactic structure of Hydra.

```
                  ->  {   }
                  _   ()  (
                  ,   )   =
                  <>  +   -
                  /   *   ^
                  ;
```

Figure 5.2: Symbols used in Hydra.

data *Ident*   = *Ident   String*
data *HsExpr* = *HsExpr String*

data *SigRel* = *SigRel Pattern* [*Equation*]
data *SigFun* = *SigFun Pattern Expr*

data *Pattern* = *PatWild*
            | *PatVar Ident*
            | *PatUnit*
            | *PatPair Pattern Pattern*

data *Equation* = *EquEqual   Expr   Expr*
               | *EquInit   Expr   Expr*
               | *EquLocal   Ident*
               | *EquSigRelApp HsExpr Expr*

data *Expr* = *ExprAdd   Expr Expr*
          | *ExprSub   Expr Expr*
          | *ExprDiv   Expr Expr*
          | *ExprMul   Expr Expr*
          | *ExprPow   Expr Expr*
          | *ExprNeg   Expr*
          | *ExprApp   Expr Expr*
          | *ExprVar   Ident*
          | *ExprAnti   HsExpr*
          | *ExprInteger Integer*
          | *ExprDouble Double*
          | *ExprUnit*
          | *ExprPair   Expr Expr*

Figure 5.3:   Abstract syntax of the quasi-quoted fragment of Hydra.

wild card pattern. Given an identifier the constructor *PatVar* constructs a pattern that
binds the corresponding single signal variable. The constructor *PatUnit* constructs the
pattern that only matches unit signals. The constructor *PatPair* constructs a pattern
that matches the components of a signal carrying pairs or, due to the the isomorphism
between a signal of products and a product of signals, a pair of signals.

The data type *Equation* is used to represent noncausal equations and local signal
variable declarations. The constructor *EquEqual* constructs an equation that asserts
equality of two signal expressions. The constructor *EquInit* constructs an initialisation
equation that asserts equality of two signal expressions. The constructor *EquLocal*
constructs a local variable declaration. The constructor *EquSigRelApp* constructs a
signal relation application that applies the signal relation that is computed by the
antiquoted Haskell expression to the given signal expression.

The data type *Expr* is used to represent signal expressions. Common mathematical
operations, identifiers, antiquoted Haskell expressions, integer and real constants, unit
signals, and pairs of signals can be used to construct signal expressions (see Figure 5.3
for details).

5.3   Desugaring

Before we turn our attention to the translation of the untyped abstract syntax into
typed abstract syntax, we describe a translation that desugars all equations that assert
equality of signal pairs into equations asserting equality of scalar signals. This trans-
lation allows for a simpler typed representation as we show in the following section.
The translation is given in Figure 5.4 as a Haskell function working with the untyped
abstract syntax of Hydra.

5.4   Typed Abstract Syntax

The typed abstract syntax that embodies the type system of Hydra is given in Fig-
ure 5.5 as a GADT definition. The types *Signal* α and *PrimSF* α β are genuine
GADTs, while the data types *SR* α, *SF* α β and *Equation* are ADTs that use the
GADT notation for consistency. Note that the typed abstract syntax uses a technique
called *higher-order abstract syntax (HOAS)* [Pfenning and Elliot, 1988]. Specifically,
through the use of function-valued fields we use Haskell's variable binding mechanism
to represent signal variable bindings in Hydra.

$$\llbracket \cdot \rrbracket \qquad\qquad\qquad :: SigRel \to SigRel$$
$$\llbracket SigRel\ pat\ eqs \rrbracket = SigRel\ pat\ (concat\ [\,\llbracket eq \rrbracket_{eq}\ |\ eq \leftarrow eqs\,])$$

$$\llbracket \cdot \rrbracket_{eq} \qquad\qquad\qquad\qquad :: Equation \to [\,Equation\,]$$
$$\llbracket EquEqual\ (Pair\ e_1\ e_2)\ (Pair\ e3\ e4) \rrbracket_{eq} =$$
$$\quad \llbracket EquEqual\ e_1\ e3 \rrbracket_{eq} \mathbin{+\!\!+} \llbracket EquEqual\ e_2\ e4 \rrbracket_{eq}$$
$$\llbracket EquInit\quad (Pair\ e_1\ e_2)\ (Pair\ e3\ e4) \rrbracket_{eq} =$$
$$\quad \llbracket EquInit\quad e_1\ e3 \rrbracket_{eq} \mathbin{+\!\!+} \llbracket EquInit\quad e_2\ e4 \rrbracket_{eq}$$
$$\llbracket eq \rrbracket_{eq} \qquad\qquad\qquad\qquad = [\,eq\,]$$

Figure 5.4: Desugaring translation of Hydra.

5.5 From Untyped to Typed Abstract Syntax

The translation rules that transform a model in the untyped representation into the corresponding model in the typed representation are given in Figure 5.6 and Figure 5.7. These rules translate an untyped term into Haskell code that builds the corresponding typed term. The pattern matching semantics in the left-hand side of the translation rules are that of Haskell.

You may have noticed that there are no translation rules that generate the *Switch* constructor. The functional-level combinator *switch*, which was introduced in Chapter 4, generates the *Switch* constructor of the typed abstract syntax. Specifically, the *switch* combinator is defined as follows.

$$switch :: SR\ a \to SF\ a\ \mathbb{R} \to (a \to SR\ a) \to SR\ a$$
$$switch = Switch$$

The typed abstract syntax embodies Hydra's type system features that were only informally introduced in earlier sections of the thesis. Let us outline several key features. The type of a signal relation is determined by its pattern. A type of a structurally dynamic signal relation remains unchanged despite the structural changes. Equality constraints, and signal relation and signal function applications must be well typed.

Note that the type system says nothing about the solvability of signal relations. It is possible to define a type correct signal relation that does not have a solution or has more than one solution. It is the responsibility of the modeller to define a signal

data *SR* α **where**
   SR   :: (*Signal* α → [*Equation*]) → *SR* α
   Switch :: *SR* α → *SF* α ℝ → (α → *SR* α) → *SR* α

data *SF* α β **where**
   SF :: (*Signal* α → *Signal* β) → *SF* α β

data *Equation* **where**
   Local :: (*Signal* ℝ → [*Equation*]) → *Equation*
   Equal :: *Signal* ℝ → *Signal* ℝ → *Equation*
   Init   :: *Signal* ℝ → *Signal* ℝ → *Equation*
   App   :: *SR* α → *Signal* α → *Equation*

data *Signal* α **where**
   Unit   :: *Signal* ()
   Time   :: *Signal* ℝ
   Const   :: ℝ → *Signal* ℝ
   Pair   :: *Signal* α → *Signal* β → *Signal* (α, β)
   PrimApp :: *PrimSF* α β → *Signal* α → *Signal* β
   Signal   :: (ℝ → α) → *Signal* α

data *PrimSF* α β **where**
   Der   :: *PrimSF* ℝ ℝ
   Exp   :: *PrimSF* ℝ ℝ
   Sqrt   :: *PrimSF* ℝ ℝ
   Log   :: *PrimSF* ℝ ℝ
   Sin   :: *PrimSF* ℝ ℝ
   Tan   :: *PrimSF* ℝ ℝ
   Cos   :: *PrimSF* ℝ ℝ
   Asin :: *PrimSF* ℝ ℝ
   Atan :: *PrimSF* ℝ ℝ
   Acos :: *PrimSF* ℝ ℝ
   Sinh :: *PrimSF* ℝ ℝ
   Tanh :: *PrimSF* ℝ ℝ
   Cosh :: *PrimSF* ℝ ℝ
   Asinh :: *PrimSF* ℝ ℝ
   Atanh :: *PrimSF* ℝ ℝ
   Acosh :: *PrimSF* ℝ ℝ
   Add   :: *PrimSF* (ℝ, ℝ) ℝ
   Mul   :: *PrimSF* (ℝ, ℝ) ℝ
   Div   :: *PrimSF* (ℝ, ℝ) ℝ
   Pow :: *PrimSF* (ℝ, ℝ) ℝ

Figure 5.5: Typed intermediate representation of Hydra.

$$\llbracket SigRel\ pattern\ equations \rrbracket_{sr} = SR\ (\lambda \llbracket pattern \rrbracket_{pat} \rightarrow \llbracket equations \rrbracket_{eqs})$$
$$\llbracket SigFun\ pattern\ expression \rrbracket_{sf} = SF\ (\lambda \llbracket pattern \rrbracket_{pat} \rightarrow \llbracket expression \rrbracket_{exp})$$

$$\llbracket PatWild \rrbracket_{pat} = \_$$
$$\llbracket PatVar\ (Ident\ s) \rrbracket_{pat} = \llbracket s \rrbracket_{hs}$$
$$\llbracket PatUnit \rrbracket_{pat} = Unit$$
$$\llbracket PatPair\ pat_1\ pat_2 \rrbracket_{pat} = Pair\ \llbracket pat_1 \rrbracket_{pat}\ \llbracket pat_2 \rrbracket_{pat}$$

$$\llbracket [\ ] \rrbracket_{eqs} = [\ ]$$
$$\llbracket (EquSigRelApp\ (HsExpr\ s)\ e) : eqs \rrbracket_{eqs} = (App\quad \llbracket s \rrbracket_{hs}\quad \llbracket e \rrbracket_{exp})\ : \llbracket eqs \rrbracket_{eqs}$$
$$\llbracket (EquEqual\ e_1\ e_2) : eqs \rrbracket_{eqs} = (Equal\ \llbracket e_1 \rrbracket_{exp}\ \llbracket e_2 \rrbracket_{exp}) : \llbracket eqs \rrbracket_{eqs}$$
$$\llbracket (EquInit\quad e_1\ e_2) : eqs \rrbracket_{eqs} = (Init\quad \llbracket e_1 \rrbracket_{exp}\ \llbracket e_2 \rrbracket_{exp}) : \llbracket eqs \rrbracket_{eqs}$$
$$\llbracket (EquLocal\ (Ident\ s)) : eqs \rrbracket_{eqs} = [\ Local\ (\lambda \llbracket s \rrbracket_{hs} \rightarrow \llbracket eqs \rrbracket_{eqs})]$$

Figure 5.6: Translation of untyped signal functions and signal relations into typed signal functions and signal relations. The translation rule $\llbracket \cdot \rrbracket_{hs}$ takes a string in the concrete syntax of Haskell and generates the corresponding Haskell expression. The translation rules $\llbracket \cdot \rrbracket_{exp}$ and $\llbracket \cdot \rrbracket_{ident}$ are given in Figure 5.7.

relation that has an unique solution. Recent work in the context of the FHM framework has made progress in the direction of more expressive type systems incorporating the solvability aspect of noncausal models [Nilsson, 2008, Capper and Nilsson, 2010]. Incorporation of the aforementioned work in Hydra is a subject of future research.

5.6 Ideal Semantics

A formal language definition has a number of advantages over an informal presentation. A formal semantics does not leave room for ambiguity and allows different implementers to implement the same language. In addition, a formally defined semantics paves the way for proving useful statements about the language.

For conventional programming languages, the aim of the dynamic semantics is typically to characterise the meaning of programs expressed in the language exactly; for example, as a final result, a trace, or a set of possible results or traces. The situation for languages for physical modelling is quite different: the model has a precise mathematical meaning, but when trying to find out what this meaning is through

$$\begin{aligned}
\llbracket ExprAnti \ (HsExpr \ s_1) \rrbracket_{exp} &= Const \ \llbracket s_1 \rrbracket_{hs} \\
\llbracket ExprVar \ (Ident \ s_1) \rrbracket_{exp} &= \llbracket s_1 \rrbracket_{ident} \\
\llbracket ExprAdd \ e_1 \ e_2 \rrbracket_{exp} &= PrimApp \ Add \ (Pair \ \llbracket e_1 \rrbracket_{exp} \ \llbracket e_1 \rrbracket_{exp}) \\
\llbracket ExprSub \ e_1 \ e_2 \rrbracket_{exp} &= PrimApp \ Sub \ (Pair \ \llbracket e_1 \rrbracket_{exp} \ \llbracket e_1 \rrbracket_{exp}) \\
\llbracket ExprDiv \ e_1 \ e_2 \rrbracket_{exp} &= PrimApp \ Div \ (Pair \ \llbracket e_1 \rrbracket_{exp} \ \llbracket e_1 \rrbracket_{exp}) \\
\llbracket ExprMul \ e_1 \ e_2 \rrbracket_{exp} &= PrimApp \ Mul \ (Pair \ \llbracket e_1 \rrbracket_{exp} \ \llbracket e_1 \rrbracket_{exp}) \\
\llbracket ExprPow \ e_1 \ e_2 \rrbracket_{exp} &= PrimApp \ Pow \ (Pair \ \llbracket e_1 \rrbracket_{exp} \ \llbracket e_1 \rrbracket_{exp}) \\
\llbracket ExprNeg \ e_1 \rrbracket_{exp} &= PrimApp \ Neg \ \llbracket e_1 \rrbracket_{exp} \\
\llbracket ExprApp \ (ExprAnti \ (HsExpr \ s)) \ e \rrbracket_{exp} &= (\textbf{case} \ \llbracket s \rrbracket_{hs} \ \textbf{of} \ SF \ f \rightarrow f) \ \llbracket e \rrbracket_{exp} \\
\llbracket ExprApp \ e_1 \ e_2 \rrbracket_{exp} &= \llbracket e_1 \rrbracket_{exp} \ \llbracket e_2 \rrbracket_{exp} \\
\llbracket ExprInteger \ i_1 \rrbracket_{exp} &= Const \ (fromIntegral \ i_1) \\
\llbracket ExprDouble \ d_1 \rrbracket_{exp} &= Const \ d_1 \\
\llbracket ExprUnit \rrbracket_{exp} &= Unit \\
\llbracket ExprPair \ e_1 \ e_2 \rrbracket_{exp} &= Pair \ \llbracket e_1 \rrbracket_{exp} \ \llbracket e_2 \rrbracket_{exp}
\end{aligned}$$

$$\begin{aligned}
\llbracket Ident \ \texttt{"time"} \rrbracket_{ident} &= Time \\
\llbracket Ident \ \texttt{"der"} \rrbracket_{ident} &= PrimApp \ Der \\
\llbracket Ident \ \texttt{"exp"} \rrbracket_{ident} &= PrimApp \ Exp \\
\llbracket Ident \ \texttt{"sqrt"} \rrbracket_{ident} &= PrimApp \ Sqrt \\
\llbracket Ident \ \texttt{"log"} \rrbracket_{ident} &= PrimApp \ Log \\
\llbracket Ident \ \texttt{"sin"} \rrbracket_{ident} &= PrimApp \ Sin \\
\llbracket Ident \ \texttt{"tan"} \rrbracket_{ident} &= PrimApp \ Tan \\
\llbracket Ident \ \texttt{"cos"} \rrbracket_{ident} &= PrimApp \ Cos \\
\llbracket Ident \ \texttt{"asin"} \rrbracket_{ident} &= PrimApp \ Asin \\
\llbracket Ident \ \texttt{"atan"} \rrbracket_{ident} &= PrimApp \ Atan \\
\llbracket Ident \ \texttt{"acos"} \rrbracket_{ident} &= PrimApp \ Acos \\
\llbracket Ident \ \texttt{"sinh"} \rrbracket_{ident} &= PrimApp \ Sinh \\
\llbracket Ident \ \texttt{"tanh"} \rrbracket_{ident} &= PrimApp \ Tanh \\
\llbracket Ident \ \texttt{"cosh"} \rrbracket_{ident} &= PrimApp \ Cosh \\
\llbracket Ident \ \texttt{"asinh"} \rrbracket_{ident} &= PrimApp \ Asinh \\
\llbracket Ident \ \texttt{"atanh"} \rrbracket_{ident} &= PrimApp \ Atanh \\
\llbracket Ident \ \texttt{"acosh"} \rrbracket_{ident} &= PrimApp \ Acosh \\
\llbracket Ident \ s \rrbracket_{ident} &= \llbracket s \rrbracket_{hs}
\end{aligned}$$

Figure 5.7: Translation of untyped signal expressions into typed signal expressions.

numerical simulation, we can only ever hope to find an approximation to this meaning up to some desired precision.

For example, consider the system of equations modelling the simple electrical circuit given in Chapter 2. In the process of deriving the simulation code we introduced a number of approximations. The continuous real numbers were approximated using the double-precision machine floating-point numbers and the system of equations was approximated using the Haskell code implementing the forward Euler method.

Implementations of noncausal modelling languages typically allow modellers to choose floating-point representations (e.g., single or double precision), symbolic processing methods and numerical simulation methods to be used during the simulation. This amounts to allowing modellers to choose a combination of approximations prior to simulation.

The fact that the implementations are only expected to approximate noncausal models needs to be taken into account when defining a formal semantics for a noncausal language. For example, definition of operational semantics [Plotkin, 2004] is problematic as it is hard to account for the myriad of approximation combinations that were outlined earlier. One option is to parameterise the operational semantics on approximations. This is feasible, but leaves the bulk of operational details unspecified defeating the purpose of an operational semantics.

For the reasons outlined above, and because the concept of first-class models, which allows for higher-order and structurally dynamic modelling, is not predicated on particular approximations used during simulation, we opted to use *ideal* semantics obtained by translating noncausal models into second-order logic predicates for formally defining the Hydra language. By referring to the semantics as ideal, we emphasise that concrete implementations are only expected to approximate the semantics.

The primary goal of the semantics that is given in this section is to precisely and concisely communicate Hydra's definition to modelling language designers and implementers, in order to facilitate incorporation of Hydra's key features in other noncausal modelling languages.

Although not considered in this thesis, the ideal semantics of Hydra can also be used to verify concrete implementations of Hydra with certain approximations. In addition, the ideal semantics can be used to check whether concrete simulation results correspond to the source-level noncausal model, again under certain approximation; for example, by using the absolute error tolerance of the numerical simulation. These two applications of the ideal semantics are subjects of future work.

The ideal semantics of Hydra are given in Figure 5.8 and in Figure 5.9. Note that the translation targets are the same as the conceptual definitions of signals, signal functions, and signal relations given in Chapter 4. Specifically, signal relations are mapped to functions from starting time, stopping time and signal to second-order logic propositions, signal functions are mapped to functions from signal to signal, and signals are mapped to function from time to value. Time is represented by the real number.

A signal relation translation may involve existentially quantified function symbols (i.e., signals). This is what makes the target predicates second-order logic predicates (i.e., not expressible in first-order logic). In other words, solving of a signal relation can be understood as proving of *existence* of signals that satisfy the given constraints (see Figure 5.8 for details). Thus the simulator can be seen as a constraint solver that tries to solve the translations of Hydra models.

It is interesting to note that for a model of an unbounded structurally dynamic system the semantics gives rise to the predicate in *infinitary* second-order logic [Karp, 1964]. The typed abstract syntax representation of such model would be infinite because of the switches that generate unbounded number of modes, and consequently the ideal semantics would map such infinite representation to the predicate with infinite chain of disjunctions. This is not to say that it is not possible to solve such constraints. Quite the opposite, in Section 4.5 we modelled and simulated one such system. If in a finite period of time the model switches finite number of times it is possible to simulate it, assuming of course that individual modes of operations can be solved in the given period. Interestingly, in the context of ideal semantics of FRP languages, similar requirements have been proposed by Wan and Hudak [2000] and Sculthorpe and Nilsson [2011] for characterising when an unbounded structurally dynamic causal model can be simulated.

To my knowledge, Hydra is the first language that features a formal specification capturing both continuous and discrete aspects of a noncausal language that supports unbounded structural dynamism. Although detailed studies of the ideal semantics and its properties still lie ahead, we think that the semantics given in this section provides a good starting point for such an undertaking.

$$\llbracket SR\ f \rrbracket_{sr} \qquad\quad = \lambda t_1\ t_2\ s \to \llbracket f \rrbracket_{f1}\ t_1\ t_2\ s$$

$$\llbracket Switch\ sr\ sf\ f \rrbracket_{sr} = \lambda t_1\ t_2\ s \to$$
$$\quad (\llbracket sr \rrbracket_{sr}\ t_1\ t_2\ s) \wedge (\forall\,t \in \mathbb{R}.\ t_1 < t \leqslant t_2 \Rightarrow \neg\ (\llbracket sf \rrbracket_{zc}\ s\ t))$$
$$\quad \vee$$
$$\quad (\exists\,t_e \in \mathbb{R}.\ (t_1 < t_e \leqslant t_2)$$
$$\qquad\qquad\qquad \wedge$$
$$\qquad\qquad\qquad (\llbracket sr \rrbracket_{sr}\ t_1\ t_e\ s)$$
$$\qquad\qquad\qquad \wedge$$
$$\qquad\qquad\qquad (\llbracket sf \rrbracket_{zc}\ s\ t_e)$$
$$\qquad\qquad\qquad \wedge$$
$$\qquad\qquad\qquad (\forall\,t \in \mathbb{R}.\ t_1 < t < t_e \Rightarrow \neg\ (\llbracket sf \rrbracket_{zc}\ s\ t))$$
$$\qquad\qquad\qquad \wedge$$
$$\qquad\qquad\qquad (\llbracket f \rrbracket_{f2}\ t_e\ t_2\ s))$$

$$\llbracket SF\ sf \rrbracket_{sf} = sf$$

$$\llbracket -,-,-,[\ ] \rrbracket_{eqs} \qquad\qquad\quad = \top$$
$$\llbracket i\,,t_1,t_2,(Local\ f) \qquad : eqs \rrbracket_{eqs} = (\exists\,s_i \in \mathbb{R} \to \mathbb{R}.\ \llbracket f \rrbracket_{f3}\ i\ t_1\ t_2\ s_i\ eqs)$$
$$\llbracket i\,,t_1,t_2,(App\ sr\ s) \qquad : eqs \rrbracket_{eqs} = (\llbracket sr \rrbracket_{sr}\ t_1\ t_2\ \llbracket s \rrbracket_{sig}) \wedge \llbracket i,t_1,t_2,eqs \rrbracket_{eqs}$$
$$\llbracket i\,,t_1,t_2,(Equal\ s_1\ s_2) : eqs \rrbracket_{eqs} =$$
$$\quad (\forall\,t \in \mathbb{R}.\ (t \geqslant t_1 \wedge t \leqslant t_2) \Rightarrow \llbracket s_1 \rrbracket_{sig}\ t \equiv \llbracket s_2 \rrbracket_{sig}\ t) \wedge \llbracket i,t_1,t_2,eqs \rrbracket_{eqs}$$
$$\llbracket i\,,t_1,t_2,(Init\ s_1\ s_2) \quad : eqs \rrbracket_{eqs} =$$
$$\quad (\forall\,t \in \mathbb{R}.\ (t \equiv t_1) \qquad\qquad \Rightarrow \llbracket s_1 \rrbracket_{sig}\ t \equiv \llbracket s_2 \rrbracket_{sig}\ t) \wedge \llbracket i,t_1,t_2,eqs \rrbracket_{eqs}$$

$$\llbracket sf \rrbracket_{zc} = \lambda s\ t \to \llbracket(\llbracket sf \rrbracket_{sf}\ (Signal\ s))\rrbracket_{sig}\ t \equiv 0 \wedge \tfrac{d_-}{dt}\llbracket(\llbracket sf \rrbracket_{sf}\ (Signal\ s))\rrbracket_{sig}\ t \not\equiv 0$$

$$\llbracket f \rrbracket_{f1} = \lambda\quad t_1\ t_2\ s \qquad \to \llbracket(0,t_1,t_2,f\ (Signal\ s))\rrbracket_{eqs}$$
$$\llbracket f \rrbracket_{f2} = \lambda\quad t_1\ t_2\ s \qquad \to \llbracket f\ (s\ t_1) \rrbracket_{sr}$$
$$\llbracket f \rrbracket_{f3} = \lambda i\ t_1\ t_2\ s\ eqs \to \llbracket i+1,t_1,t_2,f\ (Signal\ s) +\!\!+ eqs \rrbracket_{eqs}$$

Figure 5.8: Translation of signal relations, signal functions and equations. Note that, $\frac{d_-}{dt}$ denotes left derivative.

$$
\begin{aligned}
[\![Unit]\!]_{sig} &= \lambda_- \rightarrow () \\
[\![Time]\!]_{sig} &= \lambda t \rightarrow t \\
[\![Const\ d]\!]_{sig} &= \lambda_- \rightarrow d \\
[\![Pair\ s_1\ s_2]\!]_{sig} &= \lambda t \rightarrow ([\![s_1]\!]_{sig}\ t, [\![s_2]\!]_{sig}\ t) \\
[\![PrimApp\ Der\ s]\!]_{sig} &= \lambda t \rightarrow \tfrac{d}{dt}[\![s]\!]_{sig}\ t \\
[\![PrimApp\ Exp\ s]\!]_{sig} &= \lambda t \rightarrow exp \quad ([\![s]\!]_{sig}\ t) \\
[\![PrimApp\ Sqrt\ s]\!]_{sig} &= \lambda t \rightarrow sqrt \quad ([\![s]\!]_{sig}\ t) \\
[\![PrimApp\ Log\ s]\!]_{sig} &= \lambda t \rightarrow log \quad ([\![s]\!]_{sig}\ t) \\
[\![PrimApp\ Sin\ s]\!]_{sig} &= \lambda t \rightarrow sin \quad ([\![s]\!]_{sig}\ t) \\
[\![PrimApp\ Tan\ s]\!]_{sig} &= \lambda t \rightarrow tan \quad ([\![s]\!]_{sig}\ t) \\
[\![PrimApp\ Cos\ s]\!]_{sig} &= \lambda t \rightarrow cos \quad ([\![s]\!]_{sig}\ t) \\
[\![PrimApp\ Asin\ s]\!]_{sig} &= \lambda t \rightarrow asin \quad ([\![s]\!]_{sig}\ t) \\
[\![PrimApp\ Atan\ s]\!]_{sig} &= \lambda t \rightarrow atan \quad ([\![s]\!]_{sig}\ t) \\
[\![PrimApp\ Acos\ s]\!]_{sig} &= \lambda t \rightarrow acos \quad ([\![s]\!]_{sig}\ t) \\
[\![PrimApp\ Sinh\ s]\!]_{sig} &= \lambda t \rightarrow sinh \quad ([\![s]\!]_{sig}\ t) \\
[\![PrimApp\ Tanh\ s]\!]_{sig} &= \lambda t \rightarrow tanh \quad ([\![s]\!]_{sig}\ t) \\
[\![PrimApp\ Cosh\ s]\!]_{sig} &= \lambda t \rightarrow cosh \quad ([\![s]\!]_{sig}\ t) \\
[\![PrimApp\ Asinh\ s]\!]_{sig} &= \lambda t \rightarrow asinh\ ([\![s]\!]_{sig}\ t) \\
[\![PrimApp\ Atanh\ s]\!]_{sig} &= \lambda t \rightarrow atanh\ ([\![s]\!]_{sig}\ t) \\
[\![PrimApp\ Acosh\ s]\!]_{sig} &= \lambda t \rightarrow acosh\ ([\![s]\!]_{sig}\ t) \\
[\![PrimApp\ Add\ (Pair\ s_1\ s_2)]\!]_{sig} &= \lambda t \rightarrow ([\![s_1]\!]_{sig}\ t) + ([\![s_2]\!]_{sig}\ t) \\
[\![PrimApp\ Mul\ (Pair\ s_1\ s_2)]\!]_{sig} &= \lambda t \rightarrow ([\![s_1]\!]_{sig}\ t) * ([\![s_2]\!]_{sig}\ t) \\
[\![PrimApp\ Div\ (Pair\ s_1\ s_2)]\!]_{sig} &= \lambda t \rightarrow ([\![s_1]\!]_{sig}\ t)\ /\ ([\![s_2]\!]_{sig}\ t) \\
[\![PrimApp\ Pow\ (Pair\ s_1\ s_2)]\!]_{sig} &= \lambda t \rightarrow ([\![s_1]\!]_{sig}\ t) \uparrow ([\![s_2]\!]_{sig}\ t) \\
[\![Signal\ s]\!]_{sig} &= s
\end{aligned}
$$

Figure 5.9: Translation of signals.

Chapter 6

Implementation of Hydra

This chapter describes how Hydra is embedded in Haskell and how embedded non-causal models are simulated. Performance of the simulator is evaluated by focussing on the implementation aspects that are absent from mainstream noncausal modelling language implementations (i.e., runtime symbolic processing and JIT compilation).

6.1 Embedding

Hydra is implemented as a Haskell embedded DSL. In this section we give a detailed description of how Hydra is embedded in the host language Haskell.

We use quasiquoting, a recent Haskell extension implemented in GHC [Mainland, 2007], to provide a convenient surface (i.e., concrete) syntax for Hydra. The implementation uses quasiquoting to generate the typed representation of Hydra models from strings in the concrete syntax. An opening quasiquote specifies a function (a so-called quasiquoter) that performs the aforementioned transformation. In GHC, a quasiquoter generates Haskell code using Template Haskell [Sheard and Peyton Jones, 2002], a compile-time meta-programming facility implemented in GHC.

GHC executes quasiquoters at Haskell compile time, before type checking. As the typed abstract syntax of Hydra fully embodies the type system of Hydra, we effectively delegate the task of type checking to the host language type checker. This approach reduces the language specification and implementation effort by reusing the host language type system and the host language type checker. However, the disadvantage of this approach is the fact that type-related error messages are not phrased in domain-specific terms, but rather in terms of the Haskell encoding of the domain-specific types.

As we have seen in Chapter 5, the type system of Hydra provides the following guarantees: a type of a structurally dynamic signal relation remains unchanged despite the structural changes; arithmetic expressions, equations, signal relation applications and signal function applications are well typed. A type checked Hydra program may still fail at runtime for kinds of errors that are not checked by the type system. Specifically, a runtime error will be raised in the following three circumstances: the symbolic processor discovers that the number of equations and the number of variables for the active mode are not equal, the numerical solver can not solve the system of equations, and the numerical solver triggers a numerical exception such as division by zero.

The implementation of Hydra provides two quasiquoters: the *rel* quasiquoter for generating typed signal relations, and the *fun* quasiquoter for generating typed signal functions. The implementation of the quasiquoters is broken down into three stages: parsing, desugaring, and translation into the typed abstract syntax.

Firstly, the string in the concrete syntax of Hydra is parsed and the corresponding untyped representation is generated as an abstract syntax tree (AST). The BNF Converter (BNFC), a compiler front-end generator from a labelled BNF grammar [Pellauer et al., 2004], is used to generate the parser and the AST data type. The labelled BNF grammar of Hydra is given in Figure 6.1. The generated AST data type and the syntax that the generated parser implements are exactly the same as given in the language definition in Chapter 5. In addition, we use BNFC to generate Hydra's layout resolver allowing for a list of equations in *rel* quasiquotes to be constructed without curly braces and semicolons. The layout rules are the same as for Haskell.

Secondly, the untyped representation is desugared exactly as it is presented in the language definition (see Chapter 5).

Finally, the desugared untyped representation is translated into the typed representation. This step implements the corresponding translation rules given in Chapter 5. Note that the translation rules generate Haskell code. This is implemented by using the Template Haskell facility of GHC.

We illustrate the quasiquoting process by using a signal relation that models a parametrised van der Pol oscillator. The oscillator model is given in Figure 6.2. After the parsing stage the quasiquoted signal relation is turned into the AST that is given in Figure 6.3. After the desugaring stage we get the AST that is given in Figure 6.4. After translation into the typed representation we get the typed AST that is given in Figure 6.5.

```
entrypoints SigRel, SigFun;

SigRel. SigRel ::= Pattern "->" "{" [Equation] "}" ;

SigFun. SigFun ::= Pattern "->" "{" Expr        "}" ;

PatWild. Pattern ::= "_" ;
PatVar.  Pattern ::= Ident ;
PatUnit. Pattern ::= "()" ;
PatPair. Pattern ::= "(" Pattern "," Pattern ")" ;

EquEqual.      Equation ::=        Expr "=" Expr ;
EquInit.       Equation ::= "init" Expr "=" Expr ;
EquLocal.      Equation ::= "local" Ident;
EquSigRelApp.  Equation ::= HsExpr "<>" Expr ;

ExprAdd.     Expr1 ::= Expr1 "+"  Expr2 ;
ExprSub.     Expr1 ::= Expr1 "-"  Expr2 ;
ExprDiv.     Expr2 ::= Expr2 "/"  Expr3 ;
ExprMul.     Expr2 ::= Expr2 "*"  Expr3 ;
ExprPow.     Expr3 ::= Expr3 "^"  Expr4 ;
ExprNeg.     Expr3 ::= "-" Expr4 ;
ExprApp.     Expr4 ::= Expr4 Expr5 ;
ExprVar.     Expr5 ::= Ident ;
ExprAnti.    Expr5 ::= HsExpr ;
ExprInteger. Expr5 ::= Integer ;
ExprDouble.  Expr5 ::= Double ;
ExprUnit.    Expr5 ::= "()" ;
ExprPair.    Expr5 ::= "(" Expr "," Expr ")" ;
_.           Expr  ::= Expr1 ;
_.           Expr1 ::= Expr2 ;
_.           Expr2 ::= Expr3 ;
_.           Expr3 ::= Expr4 ;
_.           Expr4 ::= Expr5 ;
_.           Expr5 ::= "(" Expr ")" ;

separator Equation ";" ;

comment "--" ;
comment "{-" "-}" ;

token HsExpr ('$' (char - '$')* '$') ;

layout "->" ;
```

Figure 6.1: Labelled BNF grammar of Hydra. This labelled BNF grammar is used to generate Hydra's parser, untyped abstract syntax and layout resolver.

$vanDerPol :: \mathbb{R} \rightarrow SR\ ()$
$vanDerPol\ \mu = [\,rel\ |\ () \rightarrow$
$\quad$ **local** $x\ y$
$\quad$ **init** $(x, y) = (1, 1)$
$\quad der\ x = y$
$\quad der\ y = -\,x + \$\mu\ \$\,*(1 - x * x) * y$
$]\,]$

Figure 6.2:   Signal relation modelling parametrised van der Pol oscillator.

$SigRel\ PatUnit$
$\quad [\,EquLocal\ (Ident\ \texttt{"x"})\ [Ident\ \texttt{"y"}]$
$\quad , EquInit\quad (ExprPair\ (ExprVar\ (Ident\ \texttt{"x"}))\ (ExprVar\ (Ident\ \texttt{"y"})))$
$\qquad\qquad\quad (ExprPair\ (ExprInteger\ 1)\ (ExprInteger\ 1))$
$\quad , EquEqual\ (ExprApp\ (ExprVar\ (Ident\ \texttt{"der"}))\ (ExprVar\ (Ident\ \texttt{"x"})))$
$\qquad\qquad\quad (ExprVar\ (Ident\ \texttt{"y"}))$
$\quad , EquEqual\ (ExprApp\ (ExprVar\ (Ident\ \texttt{"der"}))\ (ExprVar\ (Ident\ \texttt{"y"})))$
$\qquad\qquad\quad (ExprAdd\ (ExprNeg\ (ExprVar\ (Ident\ \texttt{"x"})))$
$\qquad\qquad\qquad\qquad\quad (ExprMul\ (ExprMul\ (ExprAnti\ (HsExpr\ \texttt{"\$mu\$"}))$
$\qquad\qquad\qquad\qquad\qquad\qquad\quad (ExprSub\ (ExprInteger\ 1)$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad (ExprMul$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (ExprVar\ (Ident\ \texttt{"x"}))$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (ExprVar\ (Ident\ \texttt{"x"})))))$
$\qquad\qquad\qquad\qquad\quad (ExprVar\ (Ident\ \texttt{"x"}))))$
$\quad]$

Figure 6.3:   Untyped abstract syntax tree representing the *vanDerPol* signal relation.

SigRel PatUnit
  [*EquLocal* (*Ident* "x") [ ]
  , *EquLocal* (*Ident* "y") [ ]
  , *EquInit* (*ExprVar* (*Ident* "x")) (*ExprInteger* 1)
  , *EquInit* (*ExprVar* (*Ident* "y")) (*ExprInteger* 1)
  , *EquEqual* (*ExprApp* (*ExprVar* (*Ident* "der")) (*ExprVar* (*Ident* "x")))
     (*ExprVar* (*Ident* "y"))
  , *EquEqual* (*ExprApp* (*ExprVar* (*Ident* "der")) (*ExprVar* (*Ident* "y")))
     (*ExprAdd* (*ExprNeg* (*ExprVar* (*Ident* "x")))
        (*ExprMul* (*ExprMul* (*ExprAnti* (*HsExpr* "mu"))
           (*ExprSub* (*ExprInteger* 1)
            (*ExprMul*
             (*ExprVar* (*Ident* "x"))
             (*ExprVar* (*Ident* "x")))))
        (*ExprVar* (*Ident* "x"))))
  ]

Figure 6.4: Desugared, untyped abstract syntax tree representing the *vanDerPol* signal relation.

SR (λ() →
  [*Local* (λ*x* →
   [*Local* (λ*y* →
    [*Init x* (*Const* 1.0)
    , *Init y* (*Const* 1.0)
    , *Equal* (*PrimApp Der x*) *y*
    , *Equal* (*PrimApp Der y*)
      (*PrimApp*
       Add
       (*Pair* (*PrimApp Neg x*)
        (*PrimApp*
         Mul
         (*Pair* (*PrimApp*
           Mul
          (*Pair* (*Const* μ)
           (*PrimApp*
            Sub
           (*Pair* (*Const* 1.0)
            (*PrimApp Mul* (*Pair x x*)))))))
        y)))
   ])])])

Figure 6.5: Typed abstract syntax tree representing the *vanDerPol* signal relation.

Let us briefly overview the typed abstract syntax used in the implementation of Hydra. This is to highlight a minor difference from the typed abstract syntax presented in the language definition and to draw the reader's attention to the mixed-level embedding techniques used in the implementation.

The typed abstract syntax allows for two ways to form a signal relation: either from equations that constrain a given signal, or by temporal composition of two signal relations:

> **data** SR a **where**
> SR $:: (Signal\ a \to [\,Equation\,]) \to SR\ a$
> $Switch :: SR\ a \to SF\ a\ \mathbb{R} \to (a \to SR\ a) \to SR\ a$

The constructor SR forms a signal relation from a function that takes a signal and returns a list of equations constraining the given signal. This list of equations constitutes a system of DAEs that defines the signal relation by expressing constraints on the signal. The system of equations is not necessarily a static one as the equations may refer to signal relations that contain switches.

The *switch* combinator, which was introduced in Chapter 4, forms a signal relation by temporal composition of two signal relations. Internally, in the implementation of Hydra, such a temporal composition is represented by a signal relation formed by the *Switch* constructor:

> $switch :: SR\ a \to SF\ a\ \mathbb{R} \to (a \to SR\ a) \to SR\ a$
> $switch = Switch$

Recall that (see Section 3.2), in the implementation of Hydra, the type $\mathbb{R}$ is a type synonym of *Double*, which is a standard double-precision floating-point type of Haskell.

There are four kinds of equations:

> **data** $Equation$ **where**
> $Local$ $:: (Signal\ \mathbb{R} \to [\,Equation\,]) \to Equation$
> $Equal :: Signal\ \mathbb{R} \to Signal\ \mathbb{R} \to Equation$
> $Init$ $:: Signal\ \mathbb{R} \to Signal\ \mathbb{R} \to Equation$
> App $:: SR\ a \to Signal\ a \to Equation$

The *Local* constructor forms equations that merely introduce local signals. As it is evident from the language definition (see Section 5.6), such signals can be constrained

only by the equations that are returned by the function that is the first argument of the *Local* constructor. In contrast, equation generating functions in the *SR* constructor are allowed to be passed a signal that is constrained elsewhere. As we will see later in this chapter, this distinction is enforced by the language implementation.

Initialisation equations, formed by the *Init* constructor, state initial conditions. They are only in force when a signal relation instance first becomes active.

Equations formed by the *Equal* constructor are basic equations imposing the constraint that the valuations of the two signals have to be equal for as long as the signal relation instance that contains the equation is active.

The fourth kind of equation is signal relation application, *App*; that is, an equation such as $sr \diamond (x, y + 2)$. The application constrains the given signals by the equations defined by the signal relation.

The following code defines the typed representation of signals used in the implementation of Hydra:

> **data** *Signal a* **where**
>   *Unit*   :: *Signal* ()
>   *Time*   :: *Signal* $\mathbb{R}$
>   *Const*   :: $\mathbb{R} \rightarrow Signal \ \mathbb{R}$
>   *Pair*   :: *Signal a* $\rightarrow$ *Signal b* $\rightarrow$ *Signal* (a, b)
>   *PrimApp* :: *PrimSF a b* $\rightarrow$ *Signal a* $\rightarrow$ *Signal b*
>   *Var*   :: *Integer* $\rightarrow$ *Signal* $\mathbb{R}$

As you can see, this data type definition replaces the *Signal* constructor featured in the language definition (see Chapter 5) with the *Var* constructor. In the implementation, instead of representing signals that need to be solved as functions from time to values we represent them as signal variables whose approximated values will be determined by the numerical solver.

The *Var* constructor is not used at the stage of quasiquoting. Instead, the constructor is used later at the stage of runtime symbolic processing to instantiate each local signal variable to a distinct signal variable by using the constructor's *Integer* field. This is similar to the usage of the *Signal* constructor in the language definition in Chapter 5. There, the *Signal* constructor is not used in the first three steps of the definition. It is only used in the ideal semantics.

The implementation of Hydra supports the same set of primitive functions as defined in the language definition. Hence, in the implementation we use the same *PrimSF* data type as given in the language definition.

The implementation of Hydra uses a mixture of shallow and deep techniques of embedding. The function-valued fields in the *SR*, *Switch*, *Local* and *App* constructors correspond to the shallow part of the embedding. The rest of the data constructors, namely, *Equal*, *Init*, and all constructors of the *Signal* data type correspond to the deep part of the embedding, providing an explicit representation of language terms for further symbolic processing and ultimately compilation. As we will see in more detail later in this chapter, the continuous behaviour of each mode of operation can be described as a flat list of equations where each equation is constructed, either, by the *Init* constructor or by the *Equal* constructor. It is this representation that allows for generation of efficient simulation code. This combination of the two embedding techniques allowed us to leverage shallow embedding for high-level aspects of the embedded language, such as equation generation and temporal composition, and deep embedding for low-level aspects of the embedded language, such as generation of simulation code for efficiency.

6.2 Simulation

In this section we describe how iteratively staged Hydra models are simulated. The process is conceptually divided into three stages as illustrated in Figure 6.6. 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 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 case of an event occurrence, the process is repeated from the first stage by starting the new iteration.

Before we describe the three stages of the simulation in detail, let us briefly overview a function that performs these three stages. The simulator performs the aforementioned three stages iteratively at each structural change. A function that performs simulation has the following type signature:

$$simulate :: SR\ () \rightarrow Experiment \rightarrow IO\ ()$$
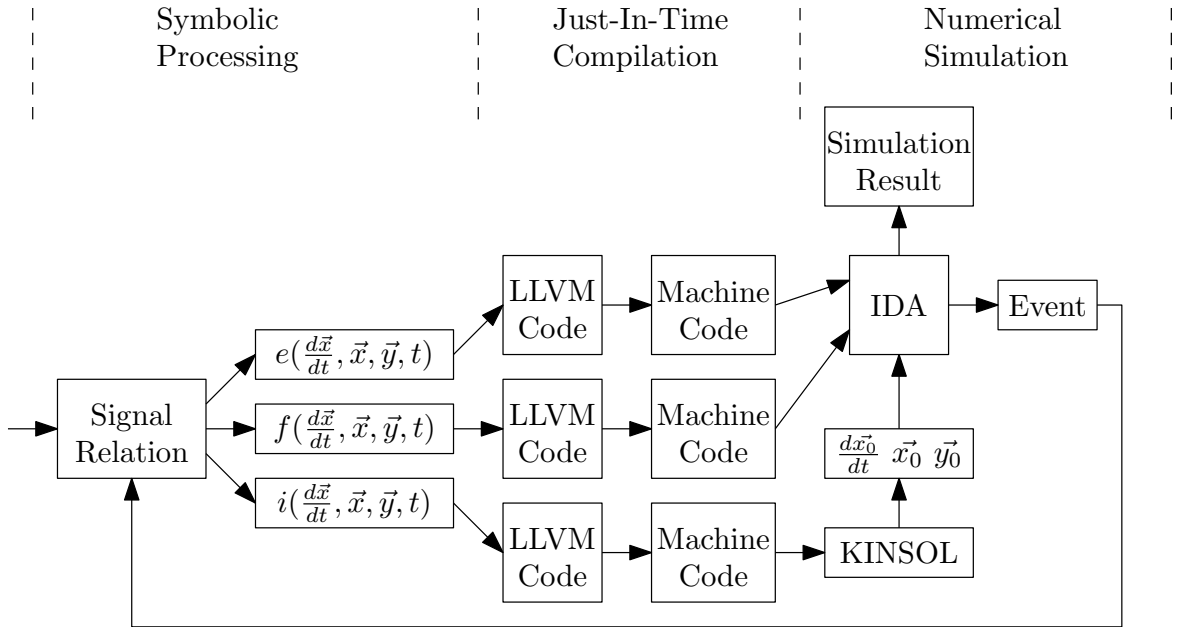
Figure 6.6: Execution model of Hydra.

The function takes a signal relation and an experiment description and simulates the system. The *Experiment* data type is defined in Figure 6.7. The *timeStart* field specifies the simulation starting time. The *timeStop* field specifies the simulation stopping time. The *timeStep* field specifies the simulation time step. The *symbolicProcessor* field specifies the simulator's runtime symbolic processor. The *numericalSolver* field specifies the simulator's numerical solver. The *trajectoryVisualiser* field specifies how to visualise the simulation results (i.e., change of signal values over time). The data type definitions for *ActiveModel*, *NumericalSover* and *TrajectoryVisualiser* are given later in this chapter.

The implementation of Hydra provides the default experiment configuration that is given in Figure 6.8. Note that the last three fields of the experiment description record are expected to be modified by expert users willing to provide their own runtime symbolic processors, numerical solvers, or trajectory visualisers. The behaviour of the *defaultSymbolicProcessor*, *defaultNumericalSolver* and *defaultTrajectoryVisualiser* are described in detail later in this chapter.

```
data Experiment = Experiment {
    timeStart           :: ℝ
  , timeStop            :: ℝ
  , timeStep            :: ℝ
  , symbolicProcessor   :: ActiveModel → ActiveModel
  , numericalSolver     :: NumericalSolver
  , trajectoryVisualiser :: TrajectoryVisualiser
  }
```

Figure 6.7: Data type for experiment description.

```
defaultExperiment :: Experiment
defaultExperiment = Experiment {
    timeStart            = 0
  , timeStop             = 10
  , timeStep             = 0.001
  , symbolicProcessor    = defaultSymbolicProcessor
  , numericalSolver      = defaultNumericalSolver
  , trajectoryVisualiser = defaultTrajectoryVisualiser
  }
```

Figure 6.8: Default experiment description.

6.3 Symbolic Processing

In this section we describe the first stage performed by the simulator: symbolic processing. A symbolic processor is a function from an active model to an active model. The active model data type that is used in the implementation of Hydra is given in Figure 6.9. The active model record has five fields.

The *model* field stores the currently active top-level signal relation. At the start of the simulation, the *simulate* function binds this field to the result of applying its signal relation argument of type *SR* () to the *Unit* signal. In other words, the *model* field contains currently active system of hierarchical equations that contains equality constrains, signal relation applications and temporal compositions.

The *equations* field is for a flat list of equations that describe an active mode of operation. By flat we mean that the list of equations only contain *Init* and *Equal* equations. A detailed description of the flattening process is given later in this section. At the start of the simulation, the *simulate* function places an empty list in this field.

The *events* field is for a list of zero-crossing signal expressions defining the event occurrences. Recall the type signature of the *switch* combinator given in Section 6.1. A signal function that detects events returns a real valued zero-crossing signal.

At the start of the simulation, the *simulate* function places an empty list in the *events* field. After the first iteration of symbolic processing, the simulator places the list of zero-crossing signals defined in the first mode of operation in the *events* field of the active model. Once the numerical solver, during the simulation of the first mode, detects that one or more signals have crossed zero the simulator deletes those signals from the *events* field that have not crossed zero. The updated *events* field is then used by the event handler to identify which events have occurred and to perform corresponding structural reconfigurations computing the next mode of operation. Before the simulation continues, the *events* field is populated with the zero-crossing signals that are active in the new mode of operation.

The *time* field is for current time. Initially the simulator places the starting time given in the experiment description in this field. The *time* field is modified at each structural change with the time of an event occurrence.

The *instants* field is for storing instantaneous values of signals. The simulator stores instantaneous values of active signals at each structural change. The instantaneous real values are stored as an array of pairs of reals. The first field is for storing the

```
data ActiveModel = ActiveModel {
    model     :: [Equation]
  , equations :: [Equation]
  , events    :: [Signal ℝ]
  , time      :: ℝ
  , instants  :: Array Integer (ℝ, ℝ)
  }
```

Figure 6.9: Data type for active models.

instantaneous signal values, while the second field is for storing the instantaneous values of signal differentials.

The task of the symbolic processor is to handle events by modifying the *model* field of the active model, to generate a flat list of events that may occur in the active mode of operation by updating the *events* field of the active model, and to generate the flat list of equations describing the active mode of operation by updating the *equations* field of the active model. The implementation of Hydra provides the default symbolic processor that is defined as follows.

$$defaultSymbolicProcessor :: ActiveModel \rightarrow ActiveModel$$
$$defaultSymbolicProcessor = flattenEquations \circ flattenEvents \circ handleEvents$$

The default symbolic processor is defined as a composition of three symbolic processing steps. The first step handles occurred events by modifying the *model* field of the active model. The event handler is defined in Figure 6.10. The second step generates a list of signal expressions representing the list of possible events in the active mode of operation as defined in Figure 6.11. Note that this step involves evaluation of the instantaneous signal values by using the $[\![\cdot]\!]_{sig}$ function. The $[\![\cdot]\!]_{sig}$ function is defined in Figure 6.12. The third step flattens the hierarchical system of equations placed in the *model* field of the active model into the *equations* field of the active model. The flat list only contains *Init* and *Equal* equations. The *Equal* equations define the DAE that describes the active mode of operation. The *Init* equations describe the initial conditions for the DAE. The flattening transformation is given in Figure 6.13.

Each of the three steps of the default symbolic processor has a compact definition, especially, the *flattenEquations* function. To my knowledge, this is the shortest formal and executable definition of the flattening process for a noncausal modelling language.

$$handleEvents \quad :: \ ActiveModel \rightarrow ActiveModel$$
$$handleEvents \ st = st \ \{ model = [\![(st, events \ st, model \ st)]\!]_{eqs} \}$$

$$[\![\cdot]\!]_{eqs} :: (ActiveModel, [Signal \ \mathbb{R}], [Equation]) \rightarrow [Equation]$$

$$[\![(\_, \_, [\,])]\!]_{eqs} = [\,]$$

$$[\![(st, evs, (Equal \ \_ \ \_) : eqs)]\!]_{eqs} = eq : [\![(st, evs, eqs)]\!]_{eqs}$$

$$[\![(st, evs, (Init \quad \_ \ \_) : eqs)]\!]_{eqs} = [\![(st, evs, eqs)]\!]_{eqs}$$

$$[\![(st, evs, (Local \ f) : eqs)]\!]_{eqs} =$$
$$\quad Local \ (\lambda s \rightarrow [\![(st, evs, f \ s)]\!]_{eqs}) : [\![(st, evs, eqs)]\!]_{eqs}$$

$$[\![(st, evs, (App \ (SR \ sr) \ s_1 \ f) : eqs)]\!]_{eqs} =$$
$$\quad App \ (SR \ (\lambda s_2 \rightarrow [\![(st, evs, f \ s_2)]\!]_{eqs})) \ s_1 : [\![(st, evs, eqs)]\!]_{eqs}$$

$$[\![(st, evs, (App \ (Switch \ sr \ (SF \ sf) \ f) \ s) : eqs)]\!]_{eqs} =$$
$$\quad \textbf{if} \ elem \ (sf \ s) \ evs$$
$$\quad\quad \textbf{then} \quad App \ (f \ [\![(time \ st, instants \ st, s)]\!]_{sig}) \ s : [\![(st, evs, eqs)]\!]_{eqs}$$
$$\quad\quad \textbf{else} \quad App \ (Switch \ (SR \ (\lambda \_ \rightarrow [\![(st, evs, [App \ sr \ s])]\!]_{eqs})) \ (SF \ sf) \ f) \ s$$
$$\quad\quad\quad : [\![(st, evs, eqs)]\!]_{eqs}$$

Figure 6.10:   Function that handles events.

$$flattenEvents :: ActiveModel \rightarrow ActiveModel$$
$$flattenEvents \ st = st \ \{ events = [\![(0, st \ \{ events = [\,] \}, model \ st)]\!]_{eqs} \}$$

$$[\![\cdot]\!]_{eqs} :: (Integer, ActiveModel, [Equation]) \rightarrow ActiveModel$$

$$[\![(\_, st, [\,])]\!]_{eqs} = st$$

$$[\![(i, st, (Local \ f) : eqs)]\!]_{eqs} = [\![(i + 1, st, f \ (Var \ i) + \!\!+ \ eqs)]\!]_{eqs}$$

$$[\![(i, st, (Equal \ \_ \ \_) : eqs)]\!]_{eqs} = [\![(i, st, eqs)]\!]_{eqs}$$

$$[\![(i, st, (Init \quad \_ \ \_) : eqs)]\!]_{eqs} = [\![(i, st, eqs)]\!]_{eqs}$$

$$[\![(i, st, (App \ (SR \ sr) \ s) : eqs)]\!]_{eqs} = [\![(i, st, sr \ s + \!\!+ \ eqs)]\!]_{eqs}$$

$$[\![(i, st, (App \ (Switch \ sr \ (SF \ sf) \ \_) \ s) : eqs)]\!]_{eqs} =$$
$$\quad [\![(i, st \ \{ events = (sf \ s) : (events \ st) \}, (App \ sr \ s) : eqs)]\!]_{eqs}$$

Figure 6.11:   Function that generates the flat list of events that may occur in the active mode of operation.

$$[\![\, \cdot \,]\!]_{sig} :: (\mathbb{R}, \textit{Array Integer } (\mathbb{R}, \mathbb{R}), \textit{Signal } a) \rightarrow a$$

$$
\begin{aligned}
&[\![(\_, \_, \textit{Unit})]\!]_{sig} &&= () \\
&[\![(t, \_, \textit{Time})]\!]_{sig} &&= t \\
&[\![(\_, \_, \textit{Const } c)]\!]_{sig} &&= c \\
&[\![(\_, v, \textit{Var } i)]\!]_{sig} &&= \textit{fst } (v \, ! \, i) \\
&[\![(t, v, \textit{Pair } e_1 \, e_2)]\!]_{sig} &&= ([\![(t, v, e_1)]\!]_{sig}, [\![(t, v, e_2)]\!]_{sig}) \\
&[\![(\_, v, \textit{PrimApp Der } (\textit{Var } i))]\!]_{sig} &&= \textit{snd } (v \, ! \, i) \\
&[\![(t, v, \textit{PrimApp } sf \; e)]\!]_{sig} &&= [\![sf]\!]_{sf} \; [\![(t, v, e)]\!]_{sig}
\end{aligned}
$$

$$[\![\, \cdot \,]\!]_{sf} :: \textit{PrimSF } a \; b \rightarrow (a \rightarrow b)$$

$$
\begin{aligned}
&[\![\textit{Exp}]\!]_{sf} &&= \textit{exp} \\
&[\![\textit{Sqrt}]\!]_{sf} &&= \textit{sqrt} \\
&[\![\textit{Log}]\!]_{sf} &&= \textit{log} \\
&[\![\textit{Sin}]\!]_{sf} &&= \textit{sin} \\
&[\![\textit{Tan}]\!]_{sf} &&= \textit{tan} \\
&[\![\textit{Cos}]\!]_{sf} &&= \textit{cos} \\
&[\![\textit{Asin}]\!]_{sf} &&= \textit{asin} \\
&[\![\textit{Atan}]\!]_{sf} &&= \textit{atan} \\
&[\![\textit{Acos}]\!]_{sf} &&= \textit{acos} \\
&[\![\textit{Sinh}]\!]_{sf} &&= \textit{sinh} \\
&[\![\textit{Tanh}]\!]_{sf} &&= \textit{tanh} \\
&[\![\textit{Cosh}]\!]_{sf} &&= \textit{cosh} \\
&[\![\textit{Asinh}]\!]_{sf} &&= \textit{asinh} \\
&[\![\textit{Atanh}]\!]_{sf} &&= \textit{atanh} \\
&[\![\textit{Acosh}]\!]_{sf} &&= \textit{acosh} \\
&[\![\textit{Add}]\!]_{sf} &&= \textit{uncurry } (+) \\
&[\![\textit{Mul}]\!]_{sf} &&= \textit{uncurry } (*) \\
&[\![\textit{Div}]\!]_{sf} &&= \textit{uncurry } (/) \\
&[\![\textit{Pow}]\!]_{sf} &&= \textit{uncurry } (**)
\end{aligned}
$$

Figure 6.12: Functions that evaluate instantaneous signal values.

$$flattenEquations :: ActiveModel \rightarrow ActiveModel$$
$$flattenEquations\ st = st\ \{\ equations = [\![(0, model\ st)]\!]_{eqs}\ \}$$

$$
\begin{array}{ll}
[\![\ \cdot\ ]\!]_{eqs} & :: (Integer, [\,Equation\,]) \rightarrow [\,Equation\,] \\
[\![(\_, [\,])]\!]_{eqs} & = [\,] \\
[\![(i, (App\ (SR\ sr)\ s) : eqs)]\!]_{eqs} & = [\![(i, sr\ s \mathbin{+\!\!+} eqs)]\!]_{eqs} \\
[\![(i, (App\ (Switch\ sr\ \_\ \_)\ s) : eqs)]\!]_{eqs} & = [\![(i, (App\ sr\ s) : eqs)]\!]_{eqs} \\
[\![(i, (Local\ f) : eqs)]\!]_{eqs} & = [\![(i+1, f\ (Var\ i) \mathbin{+\!\!+} eqs)]\!]_{eqs} \\
[\![(i, (Equal\ s_1\ s_2) : eqs)]\!]_{eqs} & = (Equal\ s_1\ s_2) : [\![(i, eqs)]\!]_{eqs} \\
[\![(i, (Init\ s_1\ s_2)\quad : eqs)]\!]_{eqs} & = (Init\ s_1\ s_2)\quad : [\![(i, eqs)]\!]_{eqs}
\end{array}
$$

Figure 6.13: Functions that flatten hierarchical systems of equations.

This is partly due to the simple abstract syntax and utilisation of shallow embedding techniques, specifically, embedded functions in the *SR* and *Switch* constructors.

The default symbolic processor that is described in this section can be extended by modellers. This extensibility is especially useful for providing further symbolic processing steps that operate on flat systems of equations. For example, the default symbolic processor does not implement an index reduction transformation [Cellier and Kofman, 2006]. Index reduction transformations minimise algebraic dependencies between equations involved in flat systems of DAEs. This allows numerical solvers to more efficiently simulate DAEs [Brenan et al., 1996]. An overview of index reduction algorithms is given in the book by Cellier and Kofman [2006]. One of those algorithms can be used to extend the default symbolic processor by introducing an index reduction step after the *flattenEquations* step.

As an example of a symbolic processor extension the implementation of Hydra provides a processor that handles higher-order derivatives and derivatives of complex signal expressions (i.e., not just signal variables). Equations that involve higher-order derivatives are translated into equivalent set of equations involving only first-order derivatives. Derivatives of complex signal expressions are simplified using symbolic differentiation.

6.4 Just-in-time Compilation

Mathematically the end result of the stage of symbolic processing is the following list of equations:

$$i(\frac{d\vec{x}}{dt}, \vec{x}, \vec{y}, t) = \vec{r_i} \qquad (6.1)$$

$$f(\frac{d\vec{x}}{dt}, \vec{x}, \vec{y}, t) = \vec{r_f} \qquad (6.2)$$

$$e(\frac{d\vec{x}}{dt}, \vec{x}, \vec{y}, t) = \vec{r_e} \qquad (6.3)$$

Here, $\vec{x}$ is a vector of differential variables, $\vec{y}$ is a vector of algebraic variables, t is time, $\vec{r_i}$ is a residual vector of initialisation equations, $\vec{r_f}$ is a residual vector of differential algebraic equations, and $\vec{r_e}$ is a vector of zero-crossing signal values. The aforementioned vectors are signals; that is, time-varying vectors.

Equation 6.1 corresponds to the *Init* equations that are placed in the *equations* field of the active model and determines the initial conditions for Equation 6.2; that is, the values of $\frac{d\vec{x}}{dt}, \vec{x}$ and $\vec{y}$ at the starting time of the active mode of operation. Equation 6.2 corresponds to the *Equal* equations that are placed in the *equations* field of the active model, and thus is the main DAE of the system that is integrated over time starting from the initial conditions. Equation 6.3 corresponds to the zero-crossing signals placed in the *events* field of the active model and specifies event conditions.

The task of a DAE solver is to find time varying valuations of $\vec{x}$ and $\vec{y}$ such that the residual vectors are zero. In addition, a DAE solver is required to detect points in time when the vector $\vec{r_e}$ changes and report it as an event occurrence.

Because it is not always possible to turn implicit equations into causal explicit ones, that is, to completely causalise them [Brenan et al., 1996], we make no assumption that any such causalisation happened, leaving causalisation when possible as future work. The generated equations are thus implicitly formulated ones. Consequently, a system of implicit equations needs to be solved at the start of the simulation of each mode of operation and at every integration step. For example, a numerical solution of the implicitly formulated DAE given in Equation 6.2 involves evaluation 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 nonlinearity of the DAE and so on.

As the functions i, f and e are evaluated from within inner loops of the solver, they have to be compiled into machine code for efficiency. Any interpretive overhead 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. As an optimisation, the code compiled for equations might be cached for future, possible reuse (see Chapter 8). The implementation of Hydra 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 function pointers for i, f and e have the following Haskell type:

> **data** *Void*
> **type** *Residual = FunPtr (* $\mathbb{R}$
> $\rightarrow Ptr\ \mathbb{R}$
> $\rightarrow Ptr\ \mathbb{R}$
> $\rightarrow Ptr\ \mathbb{R}$
> $\rightarrow IO\ Void)$

The first function argument is time. The second argument is a vector of instantaneous values of real valued signal. The third argument is a vector of instantaneous values of differentials of real-valued signals. The forth argument is a vector of residual results, or, in the case of the event specification, a vector of zero-crossing signal values. The residual functions read the first three arguments and write the residual values in the fourth argument. As these functions are passed to numerical solvers it is critical to allow for fast positional access of vector elements and in-place vector updates. Hence the use of C-style arrays.

Figure 6.14 gives the unoptimised LLVM code that is generated for the parametrised van der Pol oscillator. The corresponding optimised LLVM is given in Figure 6.15.

```
define void @hydra_residual_main(double, double*, double*, double*) {
entry:
  %4  = getelementptr double* %2, i32 1
  %5  = load double* %4
  %6  = getelementptr double* %1, i32 0
  %7  = load double* %6
  %8  = fmul double -1.000000e+00, %7
  %9  = getelementptr double* %1, i32 0
  %10 = load double* %9
  %11 = getelementptr double* %1, i32 0
  %12 = load double* %11
  %13 = fmul double %10, %12
  %14 = fmul double -1.000000e+00, %13
  %15 = fadd double 1.000000e+00, %14
  %16 = fmul double 3.000000e+00, %15
  %17 = getelementptr double* %1, i32 1
  %18 = load double* %17
  %19 = fmul double %16, %18
  %20 = fadd double %8, %19
  %21 = fmul double -1.000000e+00, %20
  %22 = fadd double %5, %21
  %23 = getelementptr double* %3, i32 0
  store double %22, double* %23
  br label %BB_0

BB_0:
  %24 = getelementptr double* %1, i32 1
  %25 = load double* %24
  %26 = getelementptr double* %2, i32 0
  %27 = load double* %26
  %28 = fmul double -1.000000e+00, %27
  %29 = fadd double %25, %28
  %30 = getelementptr double* %3, i32 1
  store double %29, double* %30
  br label %BB_1

BB_1:
  ret void
}
```

Figure 6.14: Unoptimised LLVM code for the parametrised van der Pol oscillator.

```
define void @hydra_residual_main(double, double*, double*, double*) {
entry:
  %4  = getelementptr double* %2, i32 1
  %5  = load double* %4
  %6  = load double* %1
  %7  = fmul double %6, -1.000000e+00
  %8  = fmul double %6, %6
  %9  = fmul double %8, -1.000000e+00
  %10 = fadd double %9, 1.000000e+00
  %11 = fmul double %10, 3.000000e+00
  %12 = getelementptr double* %1, i32 1
  %13 = load double* %12
  %14 = fmul double %11, %13
  %15 = fadd double %7, %14
  %16 = fmul double %15, -1.000000e+00
  %17 = fadd double %5, %16
  store double %17, double* %3
  %18 = load double* %12
  %19 = load double* %2
  %20 = fmul double %19, -1.000000e+00
  %21 = fadd double %18, %20
  %22 = getelementptr double* %3, i32 1
  store double %21, double* %22
  ret void
}
```

Figure 6.15: Optimised LLVM code for the parametrised van der Pol oscillator.

6.5 Numerical Simulation

The default numerical solver used in the current implementation of Hydra is SUNDI-ALS [Hindmarsh et al., 2005]. The solver 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.

Modellers are allowed to replace the default numerical solver. In fact, any solver that implements the interface that is given in Figure 6.16 can be used. The default numerical solver implements this interface by providing Haskell bindings to the SUNDIALS library (which is written in C) using Haskell's foreign function interface.

After each integration step that calculates a numerical approximation of a vector of active signal variables the simulator calls the *defaultTrajectoryVisualiser* function that writes the simulation results to standard output. Hydra users can provide their own signal trajectory visualiser of the following type:

$$
\begin{aligned}
\textbf{type } \textit{TrajectoryVisualiser} = \; &\mathbb{R} && \text{-- Time} \\
&\rightarrow \textit{Int} && \text{-- Variable number} \\
&\rightarrow \textit{Ptr } \mathbb{R} && \text{-- Variables} \\
&\rightarrow \textit{IO } ()
\end{aligned}
$$

For example, the user can animate the trajectories using a suitable graphical programming library.

6.6 Performance

In this section we provide a performance evaluation of the implementation of Hydra. The aim of the evaluation is to communicate to noncausal modelling language designers and implementers performance overheads of Hydra's language constructs and implementation techniques that are absent from mainstream noncausal languages. Specifically, we are mainly concerned with the overheads of mode switching (computing new structural configurations at events, runtime symbolic processing of the equations, and

```
type SolverHandle = Ptr Void
data NumericalSolver = NumericalSolver {
    createSolver   :: ℝ             -- Starting time
                   → ℝ             -- Stopping time
                   → Ptr ℝ         -- Current time
                   → ℝ             -- Absolute tolerance
                   → ℝ             -- Relative tolerance
                   → Int           -- Number of variables
                   → Ptr ℝ         -- Variables
                   → Ptr ℝ         -- Differentials
                   → Ptr Int       -- Constrained differentials
                   → Int           -- Number of events
                   → Ptr Int       -- Events
                   → Residual      -- Initialisation equations
                   → Residual      -- Main equations
                   → Residual      -- Event Equations
                   → IO SolverHandle
    , destroySolver :: SolverHandle → IO ()
    , solve         :: SolverHandle → IO CInt
        -- Return value 0: Soulution has been obtained succesfully
        -- Return value 1: Event occurence
        -- Return value 2: Stopping time has been reached
    }
```

Figure 6.16:   Numerical solver interface.

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 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, 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, noncausal modelling and simulation language implementations do not carry out dynamic mode switching, we do not compare the performance to other simulation software. The results would not be very meaningful.

The implementation of Hydra provides for user-defined symbolic processors and numerical solvers. It does not provide for a user-defined JIT compiler. In the following we evaluate the performance of the default symbolic processor, the default numerical solver and the built-in LLVM-based JIT compiler.

The evaluation setup is as follows. The numerical simulator integrates the system using variable-step, variable-order BDF (Backward Differentiation Formula) solver [Brenan et al., 1996]. 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 Haskell-embedded models 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.

To evaluate how the performance of the implementation scales with an increasing number of equations, we constructed a structurally dynamic model of an RLC circuit (i.e., a circuit consisting of resistors, inductors and capacitors) with dynamic structure. In the initial mode of operation 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.

Tables 6.1 and 6.2 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.

103

	200 Components 1000 Equations $t \, \epsilon \, [0, 10)$		400 Components 2000 Equations $t \, \epsilon \, [10, 20)$		600 Components 3000 Equations $t \, \epsilon \, [20, 30)$	
	CPU Time		CPU Time		CPU Time	
	s	%	s	%	s	%
Symbolic Processing	0.067	0.6	0.153	0.6	0.244	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
Total	10.397	100.0	25.501	100.0	48.598	100.0

Table 6.1: Time profile of structurally dynamic RLC circuit simulation (part I).

	800 Components 4000 Equations $t \, \epsilon \, [30, 40)$		1000 Components 5000 Equations $t \, \epsilon \, [40, 50)$		1200 Components 6000 Equations $t \, \epsilon \, [50, 60]$	
	CPU Time		CPU Time		CPU Time	
	s	%	s	%	s	%
Symbolic Processing	0.339	0.4	0.454	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
Total	91.317	100.0	111.179	100.0	159.624	100.0

Table 6.2: Time profile of structurally dynamic RLC circuit simulation (part II).

While JIT compilation remains the dominating part of the time spent at mode switches, Figure 6.17 demonstrates that the performance of the JIT compiler scales well. In particular, compilation time increases roughly linearly in the number of equations. 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.

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, 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 think it is worthwhile to investigate reusable code generation aspects in the context of noncausal modelling and simulation of structurally dynamic systems, and the suitability of the proposed execution model
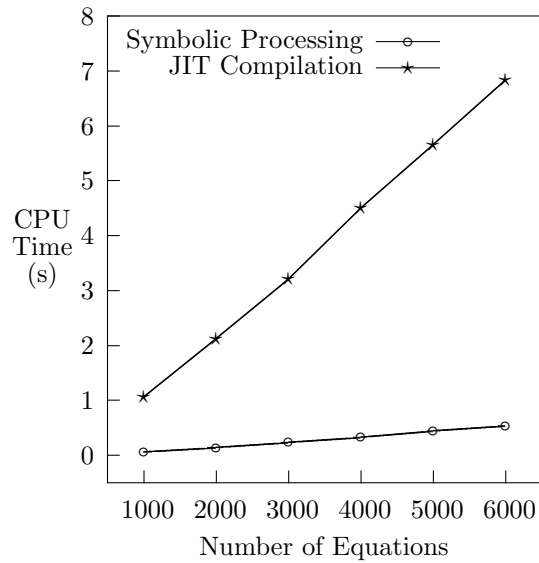
Figure 6.17:   Plot demonstrating how CPU time spent on mode switches grows as number of equations increase in structurally dynamic RLC circuit simulation.

for (soft) real-time simulation. Currently, for large structurally dynamic systems, the implementation is only suitable for offline simulation. This is also true for other implementations of unbounded structurally dynamic languages, such as Sol.

The implementation of Hydra offers new functionality in that it allows modelling and simulation of structurally dynamic systems that simply cannot be handled by static approaches. Thus, when evaluating the inherent overheads of our approach, one should weigh them against the limitation and inconvenience of not being able to model and simulate such systems at all.

Chapter 7

Related Work

7.1 Embedded Domain Specific Languages

The deep-embedding techniques used in the Hydra implementation for domain-specific optimisations and efficient code generation draws from the extensive work on compiling embedded, staged DSLs. Examples include Elliott et al. [2000] and Mainland et al. [2008]. 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. Hydra combines the aforementioned deep-embedding techniques with shallow embedding techniques in order to allow the host language to participate in runtime generation, optimisation, compilation and execution of embedded programs.

The use of quasiquoting in the implementation of Hydra draws its inspiration from Flask, a domain-specific embedded language for programming sensor networks [Mainland et al., 2008]. 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 runtime. 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 also ensures that only well-typed programs are generated at runtime.

As discussed in Chapter 1, many language features of Hydra follow closely those proposed by Nilsson et al. [2003], in the context of the FHM framework. The FHM

framework itself was originally inspired by Functional Reactive Programming (FRP) [Elliott and Hudak, 1997], particularly Yampa [Nilsson et al., 2002]. A key difference between FHM and FRP 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 modelling, while FHM is a framework for noncausal modelling. Signal functions are first class entities in most incarnations of FRP, and new ones can be computed and integrated into a running system dynamically. As we have seen, this capability has also been carried over to Hydra. This means that these FRP versions, including Yampa, are also examples of iteratively staged languages. However, as all FRP versions supporting unbounded structural dynamism so far have been interpreted, the program generation aspect is much less pronounced than what is the case for Hydra. That said, in Yampa, program fragments are generated and then optimised dynamically [Nilsson, 2005]. It would be interesting to try to apply the implementation approaches described in this thesis (i.e., runtime symbolic processing and JIT compilation) to a version of FRP, especially in the context of the recently proposed optimisations by Liu et al. [2009] and Sculthorpe [2011].

7.2 Noncausal Modelling and Simulation Languages

7.2.1 Modelling Kernel Language

Broman [Broman, 2007, Broman and Fritzson, 2008] developed Modelling Kernel Language (MKL). The language is intended to be a core language for noncausal modelling languages such as Modelica. Broman takes a functional approach to noncausal modelling, similar to the FHM approach proposed by Nilsson et al. [2003].

MKL is based on an untyped, effectful λ-calculus. The effectful part of the underling λ-calculus is used for specification of noncausal connections. Similarly to Hydra, MKL provides a λ-abstraction for defining functions and an abstraction similar to *rel* for defining noncausal models. Both functions and noncausal models are first-class entities in MKL, enabling higher-order, noncausal modelling. The similarity of the basic abstractions in Hydra and MKL leads to a similar style of modelling in both languages.

The work on MKL has not considered support for structural dynamics, meaning that its expressive power in that respect is similar to current mainstream, noncausal modelling and simulation languages like Modelica. However, given the similarities

between MKL and 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. The language design and implementation approaches (especially those related to structural dynamism) discussed in this paper should be of interest in the MKL setting.

7.2.2 Sol

Sol is a Modelica-like language [Zimmer, 2007, 2008]. It introduces language constructs that enable the description of systems where objects are dynamically created and deleted, thus supporting modelling of unbounded structurally dynamic systems. The work on Sol is complementary to ours in a number of respects outlined in the following.

Sol explores how structurally dynamic systems can be modelled in an object-oriented, noncausal language. Hydra extends a purely functional programming language with constructs for structurally dynamic, noncausal modelling.

The implementation of Sol makes use of symbolic methods that for each structural change aim to identify the smallest number of equations that need to be modified or added in order to model the structural change. It would be interesting to combine these symbolic methods with the runtime code generation approach used in Hydra in order to reduce the JIT compilation overheads by only compiling the modified and added equations for each structural change.

Sol features only an interpreted implementation. The dynamic compilation techniques featured in the implementation of Hydra would be of interest in the context of Sol to enable it to target high-end simulation tasks.

7.2.3 MOSILAB

MOSILAB is an extension of the Modelica language that supports the description of structural changes using object-oriented statecharts [Nytsch-Geusen et al., 2005]. This enables modelling of structurally dynamic systems. The language extension has a compiled implementation. However, the statechart approach implies that all structural modes must be explicitly specified in advance, meaning that MOSILAB does not support unbounded structural dynamism. Even so, if the number of possible configurations is large (perhaps generated mechanically by meta-modelling), higher-order

and structurally dynamic modelling techniques and their implementations investigated here might be of interest also in the implementation of MOSILAB.

7.2.4 Acumen

Acumen is a language for modelling and simulation of cyber-physical systems [Zhu et al., 2010]. In Acumen, a digital component can be modelled using a variant of FRP called Event-driven FRP [Wan et al., 2002], while a continuous component can be modelled using a combination of DAEs and partial differential equations (PDEs). The implementation of Acumen features advanced symbolic processing methods that reduce a combination of DAEs and PDEs to the corresponding system of ODEs whenever possible. Acumen supports bounded structural dynamism, but unbounded structural dynamism is not supported.

The symbolic processing methods developed for Acumen and its tight integration with an FRP variant would benefit Hydra, while Hydra's support for unbounded structural dynamism would benefit Acumen. Such a combination is feasible. Currently, the development of a new version of Acumen is underway[1]. This new version aims to support modelling and simulation of unbounded structurally dynamic systems. New language constructs for dynamic addition and removal of equational constraints, at discrete points in time, have already been introduced (see the work-in-progress report by Taha et al. [2011] for details). In this new version of Acumen, a model is simulated through interpretation. Because, in Acumen, a noncausal model is converted into a flat list of differential equations for numerical simulation purposes, it is feasible to integrate the JIT compilation approach described in this thesis in the implementation of Acumen.

7.3 Semantics

Proposed by Henzinger [1996], a hybrid automaton is a formal model for a hybrid system. The formalism allows a hybrid system to be specified in terms of a finite set of continuous, time-varying variables and a graph with DAEs constraining the variables at the graph nodes and switching conditions at the graph edges. Noncausal, hybrid languages can be given semantics by translation into the formalism. The modelling and simulation language Chi [Beek et al., 2008] takes this approach. Because,

---

[1]The development is being documented on the `www.acumen-language.org` website.

a hybrid automaton can only describe a bounded structurally dynamic system and does not allow new equations to be computed at switches (i.e., does not feature equational constrains as first-class entities) we did not use the hybrid automata as a target formalism when defining the ideal semantics of Hydra.

A formal semantics for the MKL language was defined by Broman [2007]. A (higher-order) model is given semantics by translation into a flat system of equations. The support for structural dynamism and its formal specification has not been considered in the setting of MKL.

Wan and Hudak [2000] define an ideal semantics for a simple FRP language. In addition to the ideal semantics, they provide an operational semantics that makes use of discrete sampling. They show that the operational semantics converges to the ideal semantics when the discrete sampling rate tends to zero. Applying a similar approach to an FHM language implementation, in order to prove its correctness, is a subject of future work.

Chapter 8

Directions for Future Work and Conclusions

In this thesis, we described a new approach to the design and implementation of non-causal modelling and simulation languages. From the language design point of view, the key idea was to embed equational models as first-class entities into a functional programming language. We provided a range of examples demonstrating how the notion of first-class models can be used for higher-order and (unbounded) structurally dynamic modelling, and thus going beyond to what is expressible in current noncausal modelling languages. From the language implementation point of view, the key idea was to enable efficient simulation of noncausal models that are generated at simulation runtime by runtime symbolic processing and just-in-time compilation. We defined a formal semantics for the language developed in this thesis and provided an in-depth description of its implementation. We hope that this work will facilitate adoption of the aforementioned approaches by designers and implementers of modelling and simulation languages.

Throughout the thesis we have identified a number of directions for future work. Let us conclude the thesis by consolidating these directions in the list given below.

- Introduce the notion of first-class models in mainstream noncausal modelling languages such as Modelica. This would allow for improved higher-order and structurally dynamic modelling capabilities, as demonstrated in this thesis. Sol [Zimmer, 2008], which is a Modelica-like language, already supports language constructs for dynamic addition and removal of equational constraints. Coupling the Modelica language extensions suggested by Zimmer with the just-in-time

compilation techniques described in this thesis would be a good starting point for extending Modelica and its implementations.

In addition to enabling the introduction of the aforementioned new language constructs to Modelica, the just-in-time compilation techniques described in this thesis allow for some restrictions on existing language constructs to be lifted. Such restrictions include the requirements on conditional equations described in Section 2.7. Lifting the restrictions on conditional equations would enable bounded structurally dynamic modelling. The new language constructs proposed by Zimmer [2008] would still be needed for unbounded structurally dynamic modelling. Zimmer [2007] gives a list of restrictions on Modelica language constructs that can be lifted with a more dynamic treatment of equational constraints. Devising of a comprehensive list of such restrictions is a subject of future work.

Although the notion of first-class models and the aforementioned just-in-time compilation techniques would benefit Modelica, full realisation of Hydra's two-level design, which extends a purely functional programming language with non-causal modelling capabilities, in the Modelica setting is difficult. This is because Modelica's syntax and semantics are deeply rooted in the object-oriented paradigm. Integration of Hydra's design into Modelica is an instance of a more general problem of integration of purely functional and object-oriented programming paradigms. This is something that we have not yet considered.

- Make use of the ideal semantics for verification of simulation results. In particular, based on the ideal semantics, it should be possible to develop a tool for automatic verification of simulation results.

- Investigate properties of the ideal semantics developed in this thesis and apply them to the problem of verification of the language implementation. Although challenging, it would be interesting to investigate possibilities of producing formally verified symbolic processors and numerical simulators for noncausal languages like Hydra.

- Develop symbolic methods for reducing mode switching overheads, especially those overheads that are associated to just-in-time compilation. Merging of Hydra's implementation approach to that of Sol would be a good starting point.

- Entirely avoid recompilation for discrete changes that are not structural changes. Introduction of the notion of impulses in Hydra would be a good starting point.

- Combine FHM and FRP frameworks in a single coherent language. The first step into this direction would be to introduce support for stateful signal functions in Hydra.

Bibliography

D. A. Van Beek, A. T. Hofkamp, M. A. Reniers, J. E. Rooda, and R. R. H. Schiffelers. Syntax and formal semantics of Chi 2.0, 2008.

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

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

David Broman and Peter Fritzson. Higher-order acausal models. In Peter Fritzson, Francois 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 59–69, Paphos, Cyprus, 2008. Linköping University Electronic Press.

Rod Burstall. Christopher strachey understanding programming languages. *Higher Order Symbol. Comput.*, 13:51–55, April 2000.

John Joseph Capper and Henrik Nilsson. Static balance checking for first-class modular systems of equations. In *Proceedings of the 11th Symposium on Trends in Functional Programming*, Oklahoma City, Oklahoma, U.S.A, May 2010.

Francois Cellier. *Continuous System Modeling.* Springer-Verlag, 1991.

Francois Cellier. Object-oriented modelling: Means for dealing with system complexity. In *Proceedings of the 15th Benelux Meeting on Systems and Control*, pages 53–64, 1996.

Francois Cellier and Ernesto Kofman. *Continuous System Simulation*. Springer-Verlag, 2006.

Antony Courtney, Henrik Nilsson, and John Peterson. The Yampa arcade. In *Proceedings of the 2003 ACM SIGPLAN Haskell Workshop*, pages 7–18, New York, NY, USA, 2003. ACM.

Dymmola. Dynamic Modeling Laboratory, 2008. URL http://www.dynasim.se/.

Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of International Conference on Functional Programming*, pages 163–173, June 1997.

Conal Elliott, Sigbjørn Finne, and Oege de Moor. Compiling embedded languages. In *Semantics, Applications, and Implementation of Program Generation*, volume 1924 of *Lecture Notes in Computer Science*, pages 9–27, Montreal, Canada, September 2000. Springer-Verlag.

P. Fritzson, P. Aronsson, A. Pop, H. Lundvall, K. Nystrom, L. Saldamli, D. Broman, and A. Sandholm. OpenModelica - a free open-source environment for system modeling, simulation, and teaching. *2006 IEEE International Symposium on Computer-Aided Control Systems Design*, pages 1588–1595, October 2006.

George Giorgidze and Henrik Nilsson. Embedding a Functional Hybrid Modelling language in Haskell. In *Revised selected papers of the 20th international symposium on Implementation and Application of Functional Languages, Hatfield, England*, volume 5836 of *Lecture Notes in Computer Science*. Springer, 2008.

George Giorgidze and Henrik Nilsson. Higher-order non-causal modelling and simulation of structurally dynamic systems. In *Proceedings of the 7th International Modelica Conference, Como, Italy*. Linköping University Electronic Press, 2009.

George Giorgidze and Henrik Nilsson. Mixed-level embedding and JIT compilation for an iteratively staged DSL. In *Revised selected papers of the 19th international workshop on Functional and (Constraint) Logic Programming, Madrid, Spain*, volume 6559 of *Lecture Notes in Computer Science*. Springer, 2010.

Thomas Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, LICS '96, pages 278–, Washington, DC, USA, 1996. IEEE Computer Society.

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.

Paul Hudak. Modular domain specific languages and tools. In *Proceedings of Fifth International Conference on Software Reuse*, pages 134–142, June 1998.

Paul Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia.* Cambridge University Press, New York, NY, USA, 1999.

Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. In Johan Jeuring and Simon Peyton Jones, editors, *Advanced Functional Programming, 4th International School 2002*, volume 2638 of *Lecture Notes in Computer Science*, pages 159–187. Springer-Verlag, 2003.

Graham Hutton. *Programming in Haskell.* Cambridge University Press, New York, NY, USA, 2007.

Carol Karp. *Languages with expressions of infinite length.* North-Holland, Amsterdam, 1964.

Chris Lattner. LLVM: An infrastructure for multi-stage optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, December 2002.

Edward Lee. Cyber physical systems: Design challenges. In *International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, May 2008.

Hai Liu, Eric Cheng, and Paul Hudak. Causal commutative arrows and their optimization. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09, pages 35–46, New York, NY, USA, 2009. ACM.

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

Geoffrey Mainland, Greg Morrisett, and Matt Welsh. Flask: Staged functional programming for sensor networks. In *Proceedings of the Thirteenth ACM SIGPLAN*

International Conference on Functional Programming, Victoria, British Columbia, Canada, September 2008. ACM Press.

Simulink. *Using Simulink Version 7*. The MathWorks, Inc., March 2008. URL `http://www.mathworks.com/products/simulink/`.

Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

Modelica Tutorial. *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling: Tutorial Version 1.4*. The Modelica Association, December 2000.

Modelica. *A Unified Object-Oriented Language for Physical Systems Modeling : Language Specification Version 3.2*. The Modelica Association, March 2010. URL `https://www.modelica.org/documents/ModelicaSpec32.pdf`.

Pieter 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.

Pieter 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.

Pieter 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.

Henrik Nilsson. Functional automatic differentiation with Dirac impulses. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, pages 153–164, Uppsala, Sweden, August 2003. ACM Press.

Henrik Nilsson. Dynamic optimization for functional reactive programming using generalized algebraic data types. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming (ICFP'05)*, pages 54–65, Tallinn, Estonia, September 2005. ACM Press.

Henrik Nilsson. Type-based structural analysis for modular systems of equations. In Peter Fritzson, Francois 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.

Henrik Nilsson and George Giorgidze. Exploiting structural dynamism in Functional Hybrid Modelling for simulation of ideal diodes. In *Proceedings of the 7th EUROSIM Congress on Modelling and Simulation, Prague, Czech Republic.* Czech Technical University Publishing House, 2010.

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

Henrik Nilsson, John Peterson, and Paul Hudak. Functional hybrid modeling. In *Proceedings of 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.

Henrik Nilsson, John Peterson, and Paul Hudak. Functional hybrid modeling from an object-oriented perspective. In Peter Fritzson, Francois 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.

NSF. Cyber-physical systems (CPS), 2008. URL `http://www.nsf.gov/pubs/2008/nsf08611/nsf08611.pdf`.

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.

Bryan O'Sullivan, John Goerzen, and Don Stewart. *Real World Haskell.* O'Reilly Media, Inc., 1st edition, 2008.

Michael Pellauer, Markus Forsberg, and Aarne Ranta. BNF Converter: Multilingual front-end generation from labelled BNF grammars. Technical report, Computing Science at Chalmers University of Technology and Gothenburg University, Sep 2004.

Simon Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report.* Cambridge University Press, Cambridge, England, 2003.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *International Conference on Functional Programming (ICFP '06)*, pages 50–61. ACM, 2006.

Frank Pfenning and Conal Elliot. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, PLDI '88, pages 199–208, New York, NY, USA, 1988. ACM.

Gordon Plotkin. A structural approach to operational semantics. *The Journal of Logic and Algebraic Programming*, 60-61:17–139, 2004.

Michael Scott. *Programming Language Pragmatics, Third Edition.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2009.

Neil Sculthorpe. *Towards Safe and Efficient Functional Reactive Programming.* PhD thesis, School of Computer Science, University of Nottingham, 2011.

Neil Sculthorpe and Henrik Nilsson. Keeping calm in the face of change: Towards optimisation of FRP by reasoning about change. *Journal of Higher-Order and Symbolic Computation (HOSC)*, 24(1), 2011.

Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. *SIGPLAN Not.*, 37(12):60–75, 2002.

Don Stewart. Domain specific languages for domain specific problems. In *Workshop on Non-Traditional Programming Models for High-Performance Computing.* LACSS, 2009.

Walid Taha. A gentle introduction to multi-stage programming. In *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 30–50. Springer Berlin/Heidelberg, 2004.

Walid Taha, Paul Brauner, Robert Cartwright, Verónica Gaspes, Aaron Ames, and Alexandre Chapoutot. A core language for executable models of cyber physical systems: work in progress report. *ACM SIGBED Review - Work-in-Progress (WiP) Session of the 2nd International Conference on Cyber Physical Systems*, 8(2):39–43, June 2011.

Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999.

Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. In *Proceedings of PLDI'01: Symposium on Programming Language Design and Implementation*, pages 242–252, June 2000.

Zhanyong Wan, Walid Taha, and Paul Hudak. Event-driven FRP. In *Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages*, PADL '02, pages 155–172, London, UK, UK, 2002. Springer-Verlag.

Günther Zauner, Daniel Leitner, and Felix Breitenecker. Modelling structural-dynamics systems in Modelica/Dymola, Modelica/MOSILAB, and AnyLogic. In Peter Fritzson, Francois 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.

Yun Zhu, Edwin Westbrook, Jun Inoue, Alexandre Chapoutot, Cherif Salama, Marisa Peralta, Travis Martin, Walid Taha, Marcia O'Malley, Robert Cartwright, Aaron Ames, and Raktim Bhattacharya. Mathematical equations as executable models of mechanical systems. In *Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems*, ICCPS '10, pages 1–11, New York, NY, USA, 2010. ACM.

Dirk Zimmer. Enhancing Modelica towards variable structure systems. In Peter Fritzson, Francois 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.

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.