DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2018

Hybrid Semantics in Equation-Based Modeling

OSCAR ERIKSSON

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Hybrid Semantics in Equation-Based Modeling

OSCAR ERIKSSON

Master in Computer Science
Date: October 22, 2018
Supervisor: David Broman
Examiner: Mads Dam
Swedish title: Hybrid semantik i ekvationsbaserad modellering
School of Computer Science and Communication

Abstract

Equation-based object-oriented modeling languages represent a highly composable class of modeling languages. In these languages models are expressed as differential-algebraic equations with no explicit causal relation between variables. Modeling of structurally varying systems in such languages is typically done by defining modes that describe the continuous evolution of the system, coupled with mode-switches describing structural changes. Specifically, structural changes can give rise to discontinuities and impulses, which can result in additional changes to the system. This thesis formalizes semantics for the treatment of structurally varying systems in such languages, including automatic handling of discontinuities and impulses from the theory of non-linear circuits. The semantics are implemented as part of an equation-based modeling language, where the treatment of impulses is based on backwards-Euler. The expressiveness of the implementation is evaluated on a number of structurally varying systems, both in the electrical and mechanical domains. We conclude that the semantics are expressive enough to describe some structurally varying systems, but are sensitive to numerical errors. Furthermore, more work is needed to allow the semantics to express inelastic collision in a satisfactory manner.

Sammanfattning

Ekvationsbaserade objektorienterade modelleringsspråk representerar modelleringsspråk med hög grad av kombinerbarhet och modell-återanvändning. I dessa språk uttrycks modeller som differential-algebraiska ekvationer utan kausal koppling mellan variablerna. Modellering av strukturellt varierande system i dessa språk görs typiskt genom att definiera moder, som beskriver systemets kontinuerliga förändring och mod-byten som beskriver strukturella förändringar. Specifikt kan strukturella förändringar kan ge upphov till diskontinuiteter och impulser, som i sin tur kan ge upphov till ytterligare förändringar i systemet. Denna uppsats formaliserar semantik för hantering av strukturellt varierande system, som innefattar automatisk hantering av diskontinuiteter och impulser hämtat från teorin kring icke-linjära kretsar, där implementationen baseras på bakåt-Euler. Uttrycksfullheten i implementationen utvärderas på en rad strukturellt varierande system i de elektriska och mekaniska domänerna. Vi drar slutsatsen att semantiken är uttrycksfull nog för att beskriva vissa strukturellt varierande system, men känslig för numeriska fel. Vidare krävs mer arbete för att kunna uttrycka inelastisk kollision på ett tillfredsställande sätt.

Contents

Glossary

λ_{DILL} The meta-modeling language defined in this thesis. 27–29, 32, 41, 43–45, 55–57, 89, 90

DILL The implementation of λ_{DILL}. vi, 28, 55, 56, 68–71, 74, 76, 79, 80, 83, 86

active mode The mode that is active at the current point in time. 13, 16, 18, 33, 44–47, 52

algebraic equation An equation that does not contain differentiated variables. 8, 9, 12, 13

algebraic variable A variable that does not appear differentiated. 12, 13

connection semantics Semantics relating a model topology to a set of topological equations. 20, 21, 24, 27, 28, 32, 33, 38, 55, 57, 58, 61

constitutive equation Equation describing a relationship between physical quantities. 9, 10, 21, 32, 39, 47, 58, 67, 68, 71, 77, 79

data type A synonym for algebraic data types. 31, 34–38, 41–43, 45, 49, 50, 55–57

dependent variable A function of the independent variable. 9, 12, 13, 16, 21, 30, 32–34, 38, 52, 56, 58, 59, 65, 73

differential equation An equation that contains differentiated variables. 8, 9, 12

differential variable A variable that appears differentiated. 12, 13

elaboration The procedure of transforming a model in equation-based obejct-oriented modeling language to a DAE or hybrid DAE. 20–27, 44, 46, 47, 52, 89

equation transformation The procedure of transforming a DAE or hybrid DAE to a form suitable for solving. 20, 22–27, 89

equation-based modeling language A modeling language based on a DAE representation. 9, 11, 25

hybrid model A model in a hybrid modeling language expressing a structurally varying system. 8, 10, 11, 13, 14, 16–18, 22–25, 27, 33, 39, 41–43, 45, 46, 49, 50, 56, 67, 76, 89

hybrid modeling language A modeling language capable of modeling structurally varying systems using modes and mode-switches. 8, 11, 24, 25, 29, 89

impulse analysis Determines the behavior of a model during a mode switch and directly after the mode switch. 26–28, 49, 66, 71, 84, 88, 90

impulse solving The procedure of finding the values on the state variables at a mode-switch and right after the mode-switch, consistent with the DEA of the successor mode, and given values on the sate variables in the predecessor mode right before the mode-switch. 27, 28, 48, 55, 62, 64–66, 72, 84–86, 88, 89

independent variable A variable that is independent and usually represents time. 9, 12, 16, 32, 33, 37

index A measure of distance between a DAE and its corresponding ODE. 12–14, 23–25, 27, 55, 65, 66, 89, 90

index reduction The process of bringing a DAE closer to an ODE. 12–14, 20, 24–27, 37, 55, 65, 66

initial value Values on the state variables at a given value on the independent variable. 13, 14, 19–21, 34, 37, 43–45, 48, 49, 51, 53, 55, 62–64, 71, 76, 78–80, 85, 86, 88

invalid region The region where a switch is invalid. 43

micro-step One discrete step in superdense time. 46, 48, 51, 86

mode A model of a particular state of a structurally varying system. 10, 13, 14, 18–20, 22–25, 29, 32, 33, 35, 38–41, 45–47, 49, 51–53, 57, 69, 71, 77, 79, 84, 86, 89

mode switch A transition between modes. 13–16, 19, 20, 22–26, 32, 35, 41, 43–46, 48, 51–53, 62–64, 71, 73, 86, 89

mode trace The simulation trace of a single mode. 37, 45, 49–51, 53

model time Time in the sense of real time. 46, 50, 51, 73, 74

modeling language A language table to express models of systems, which then can be simulated. 8, 9, 11, 12, 20, 22, 23, 26

mythical state A state in a model with multiple modes that do not have an correspondence in the physical system. 19, 51

predecessor mode The mode directly before a mode switch. 13–15, 41, 42, 48, 62, 65, 79, 80

stable A switch is stable if the solution to a model is inside its valid region. 43, 45, 46, 49, 51, 53, 64, 72

state The state of a model. 13, 19, 49–51

state variable A variable in a dynamic system that contains information needed to evolve the system. 13, 19, 20, 24, 25, 27, 30, 32, 34–37, 41–43, 45, 48, 50–53, 62–66, 71, 86, 89

structurally varying system Systems with varying structure resulting from switches, clutches, collisions or other events. 7, 8, 10, 17, 26, 67, 90

successor mode The mode directly after a mode switch. 13–15, 19, 41, 45, 48, 62, 63, 65, 79, 85, 86

superdense time A representation of time modeling both ordinary time and discrete events occurring at the same time. 46, 51

switch A syntactical construct use to define modes. 43, 44, 46, 47, 49, 63, 68–70, 73, 82, 84, 86

time-stamp A tuple (t, n) representing a point in superdense time. 46, 51

topological equation An equation that is deduced from the topological description of a model. 10, 20–22, 26, 27, 32, 33, 38, 57, 58, 60, 61, 72, 77, 79

transient state A synonym to mythical state. 19, 33, 51, 82, 85

unstable A switch is unstable if the solution to a model is inside its invalid region. 43–46, 49, 50, 52

valid region The region where a switch is valid. 43, 63, 73, 85, 86, 89

Acronyms

CDAE Conditional Differential Algebraic Equations. 34–37, 46, 47, 52, 53, 64

CIVP Conditional Initial Value Problem. 34, 37, 44, 45, 52, 53

DAC Differential Algebraic Conditions. 34, 36, 37, 39, 53

DAE Differential Algebraic Equations. 8–14, 19, 20, 23–27, 32, 33, 36, 48, 55, 62, 65, 77

DSL Domain Specific Language. 25, 27, 28, 55

EOO Equation-Based Object-Oriented. 11, 20–26, 29, 55, 67, 75, 76, 89, 90

JIT Just in Time. 23, 24, 26

ODE System of Ordinary Differential Equations. 12, 19

Chapter 1

Introduction

Physical systems, or simply *systems*, range from simple analog electrical circuits to complex mechanical systems of multiple bodies. Some of these systems change their structure over time, either at a known time instant, or as a result of the time evolution of some physical properties in the system. An electrical circuit might contain a switch, which can effectively change the number of active component in the circuit, or the string of a pendulum might break if the tension in the string becomes too high. We call such system structurally varying systems, also known as *hybrid systems* [39], and this thesis concerns the *modeling and simulation* of such systems.

Modeling and simulation are important methods in the analysis of physical systems. One could analyze a system, by observing it directly. However, such observations might not always be possible, for several reasons; It might be costly, dangerous, or some times not even physically possible. Instead, we can construct *models* of the system, and instead of analyzing the system directly, perform an analysis on the model.

We can construct models in several ways. A model can even be a physical system itself, but more commonly in natural sciences, a model comprises a *mathematical representation* of the system. Regardless of representation, a model is *always* a *simplification* of the underlying system, which is both a strength and a weakness. A model's behavior does not mirror the behavior of the system under all conditions. However, abstractions of obscuring details in the system might ease the analysis of the system's behavior.

One such *abstraction*, which we adopt in our discussion on mod-

els of structurally varying systems, is that structural changes occur as *discrete changes* in the model, which is otherwise described by *continuous dynamics*. This is an idealization of reality because the behavior of a structurally varying system is not truly discrete. One could try to model these events as detailed continuous dynamics, but as Lee [25] argues; "such detailed modeling rarely helps in developing insight about macroscopic system behavior". A model of a structurally varying systems, where we model structural changes by discrete changes to the model, is called a hybrid model.

Even though we can learn a lot from a mathematical model by analytic methods, applying these methods to more complex models are generally not possible. Instead, we often have to resort to *simulation*. By observing a models dynamic behavior on different input, we are able to simulate how the underlying system behaves under similar conditions. In essence, we are able to experiment on the system without having to perform actual physical experiments.

At the core of simulation lies a computer program that given some input, interprets and solves a model. Because this computer program needs to understand our models, we need a *syntax* for defining models, and *semantics* giving meaning to this syntax. The syntax and semantics form a language, a modeling language. A model is then defined as a program in this modeling language, and a simulation is the evaluation or meaning of such a program. A modeling language should be able to express some appropriate subset of the language of mathematics, and might also contain other constructs for increased expressiveness. We call a modeling language able to express hybrid models a hybrid modeling language.

A limitation in defining modeling languages is the practical aspects of *solving models* defined in such languages. This restricts our choice of mathematical representations. We also have to consider how much details of this solving procedure we want to expose to the end-user (the modeler). Another important aspect of a modeling language, shared among programming languages in general, is the *composability* and *reusability* of models. As we will see later on, this relates to the mathematical representation at the base of the modeling language.

This thesis considers hybrid modeling languages, where the underlying mathematical representation consist of systems of differential equations and algebraic equations, with no explicit *causality* between the unknowns in the equations. Differential Algebraic Equations

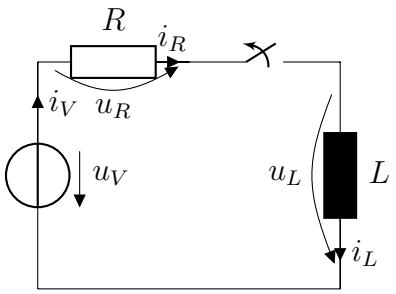


Figure 1.1: *Switched RL-circuit*

(DAE) is the term used for these systems of equations, and equation-based modeling languages is the name for modeling languages based on this mathematical representation. We now proceed with an example introducing some of these concepts.

1.1 Example of a Switched LR-Circuit

Consider the circuit depicted in Figure 1.1, which contains a voltage source, a resistor, and an inductor. To form a DAE of this system, we begin by defining equations for these components; these are the constitutive equations in (1.1).

$$u_V = V \quad u_R = R \cdot i_R \quad u_L = L \cdot \frac{\mathrm{d}i_L}{\mathrm{d}t} \tag{1.1}$$

In (1.1), V, R, and L are constants, and u_x, i_x, for $x \in \{V, R, L\}$, are so called dependent variables, which we should see as functions of the independent variable, t. The dependent variables represent quantities in the model we want to measure (or rather simulate), in this case, currents and voltages over the components. The independent variable is usually interpreted as time, and in this thesis we will interchangeably use time and independent variable to denote this variable.

The first two equations are algebraic equations, as they contain no differentiated dependent variables, and the last equation is a differential equation, as i_L appear differentiated. A system of equations, containing both differential equations and algebraic equations, is a DAE.

(1.1) contains three equations and six dependent variables. To find a unique solution on the dependent variables, as functions of the independent variable, we need an equal number of independent equations and dependent variables.

Fortunately, we can deduce the missing equations from the *topology* of the circuit. Because this circuit contains a switch, we have a structurally varying system and two cases.

Assuming a closed switch, and using *Kirchhoff's circuit-laws*, we complement (1.1) with what we call topological equations, here consisting of:

$$i_V = i_R \quad i_R = i_L \quad u_V = u_R + u_L \tag{1.2}$$

An open switch would instead give the following topological equations:

$$i_V = i_R \quad i_R = 0 \quad i_L = i_V \tag{1.3}$$

A hybrid model of this circuit would allow the switch to change its state during the course of simulation, resulting in a change in the equations of the underlying DAE, describing the dynamics of the circuit. We call each such configuration of the model (and thus the DAE) a mode.

We make a few observations on this model. The constitutive equation in (1.1) remain unchanged, regardless of the circuit topology and the resulting topological equations. This is typical for many physical systems where the topological equations stem from *energy conservation laws*. Further, if we wanted to replace the resistor in this circuit, with a capacitor, thus creating a switched LC-circuit, it would be sufficient to replace the middle equations in (1.1) with:

$$i_L = C \cdot \frac{\mathrm{d}u_L}{\mathrm{d}t}$$

Thus, the rest of the DAE is unchanged. This is an attractive feature of the DAE representation, which allows model *composability* and *model re-use*. Furthermore, analogies to Kirchhoff's circuit laws lets us mix components from different physical domains. Moreover, methods exist for *systematically deriving* the topological equations, given an appropriate topological description of the system and *generalizations* of Kirchhoff's circuit laws allowing us to apply these methods to multiple physical domains [37].

1.2 Equation-Based Object-Oriented (EOO) Modeling Languages

A subset of equation-based modeling languages, exploiting the features of DAEs [17], are the so called Equation-Based Object-Oriented (EOO) modeling languages [8]. In these modeling languages, models typically consists of sub-models, modeling *components* of the system. These *components* are then *connected together*, forming a model topology, resulting in the final model definition. The components can model individual physical component like resistors, or inductors; or themselves consists of connected components. This *component abstraction* allows easy replacement of components and model re-use. In a sense, components are like *objects* in the *object-oriented programming paradigm*. Moreover, EOO languages offer an intuitive way of constructing models, which resembles the construction of the physical system. A graphical representation of a circuit model more or less maps one-to-one to the physical circuit modeled.

This thesis will discuss the *formalization* of a EOO hybrid modeling language. In the next section, we give a more formal definition of DAEs and discuss some of their properties. We then proceed to discuss hybrid models in Section 1.4, followed by some background on EOO languages in Section 1.5. In Section 1.6, we account for some related work in the field of hybrid modeling language within equation-based modeling languages. Finally, in Sections 1.7 to 1.10, we state the *research problem, method, and contribution*. In Section 1.11, we give a short discussion on sustainability, ethics, and societal aspects of this thesis.

1.3 Differential-Algebraic Equations (DAE)

A large class of physical systems are naturally expressed using DAEs. In (1.4) we define a first-order DAE. We can restrict ourselves to first-order DAEs, as we can always rewrite a higher-order DAE into a first-order.

$$F(\mathbf{x}, \frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t}, \mathbf{w}, t) = \mathbf{0} \tag{1.4}$$

where $\mathbf{x} \in \mathbb{R}^n$, $\frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t} \in \mathbb{R}^n$, $\mathbf{w} \in \mathbb{R}^m$, $t \in \mathbb{R}$, $F : D \subseteq \mathbb{R}^{n+n+m+1} \rightarrow \mathbb{R}^{n+m}$, and $\mathbf{0}$ is a vector of zeros of sufficient size. The vector $\mathbf{x}$ is the vector of

differential variables, and $\mathbf{w}$ the vector of algebraic variables. Both are vectors of dependent variables, and t is the independent variable. The vector $\frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t}$ is the element-wise derivative of $\mathbf{x}$, with respect to t, and F is a vector-valued function containing both differential equations and algebraic equations. We generally seek the dependent variables as functions of the independent variable, consistent with (1.4) for all $t \in \mathbb{R}$ in some interval.

$$\begin{aligned} \frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t} &= G(\mathbf{x}, \mathbf{u}, t) \\ \mathbf{y} &= H(\mathbf{x}, \mathbf{u}, t) \end{aligned} \tag{1.5}$$

DAEs are generalizations of Systems of Ordinary Differential Equations. Compare (1.4) to the state-space model of (1.5), where $\mathbf{u} \in \mathbb{R}^p$ is the *input*-vector and $\mathbf{y} \in \mathbb{R}^q$ the *output*-vector. In (1.5) we have a *causal* relation between $\mathbf{u}$ and $\mathbf{y}$, and restrict algebraic equations to the form $\mathbf{y} = H(\mathbf{x}, \mathbf{u}, t)$. In (1.4), however, the relation between the variables are *acasual*. Thus, these variables have no explicit input or output relations. Furthermore, Equation (1.4) may also have arbitrary algebraic equations of variables from $\mathbf{x}$ and $\mathbf{w}$. Examples of modeling languages based on (1.5) are *Simulink*[1] and *Ptolemy II*[2].

As we illustrated in Section 1.1, problems on the form (1.4) naturally express physical systems, and allow model composability and model re-use. The composability and model re-use are not as easily achieved in models requiring a representation on the form (1.5) [20, 27]. However, there are advantages to (1.5). Analytic solutions are generally not within reach, and efficient numerical methods exist for solving these problems [12]. This is not generally the case for (1.4), and it might be necessary to *transform the problem* before moving on to the *numerical solving* [22].

Solving DAEs typically involves *symbolic transformations* of (1.4), to a form closer to that of (1.5), before *numerical solving* proceeds. The (differentiation-) index of a DAE denotes a notion of distance between the DAE and its corresponding ODE. Consequently, an ODE has an index of zero. Index reduction [14, 34, 36, 41] is the procedure of *lowering* the index of a DAE, by *systematic differentiation* of its individual equations.

[1] https://www.mathworks.com/products/simulink.html
[2] https://ptolemy.eecs.berkeley.edu/ptolemyII/

Moreover, a necessary condition for finding a *unique* solution to both (1.4) and (1.5), is a set of *consistent* initial values on the dependent variables. Additionally, in the former case, we generally also need to supply initial values for the vector $\frac{d\mathbf{x}}{dt}$. In DAEs of higher index, the differential variables might be subject to so called *latent-constraints*. These constraints make it hard to find consistent initial values to DAEs. Index reduction was initially proposed as a method to reveal these *latent-constraints* and to help in finding consistent initial values. Due to the differentiation of the individual equations in (1.4) during the index reduction, new variables might surface in the form of differentiated algebraic variables, or differentiated differential variables [34].

Further, the algebraic equations constraining dependent variables in (1.4) become implicit under index reduction. Problems arise in the discretization of index-reduced DAEs, as these constraints are not generally preserved, which can lead to numerical *drifting problems*. However, both symbolic [29] and numeric [40] techniques exist for handling this drifting.

As finding consistent initial values for higher index DAEs is difficult in general, we typically have to resort to *numerical methods*, given a *partial assignment* of the initial values [4]. Numerical non-linear DAE-solvers, such as *Sundials*[3], can reliably find consistent initial values and solve DAEs with index less than two.

1.4 Hybrid Models

Returning to the switched RL-circuit depicted in Figure 1.1, we saw that a *closed* switch gave a different DAE compared to an *open* switch. In the former case, we have (1.1) and (1.2), and in the latter case, (1.1) and (1.3). In a hybrid model of the circuit, the switch defines two modes, and we call a transition between two modes a mode switch.

Given a mode switch, we denote the mode before the mode switch, the predecessor mode, and the mode after the mode switch, the successor mode. Further, we denote the mode describing the behavior of a model at the current point in time, as the active mode. We collectively refer to the dependent variables and their derivatives, as state variables, and the active mode together with the values on the state variables, as a state.

[3] https://computation.llnl.gov/projects/sundials

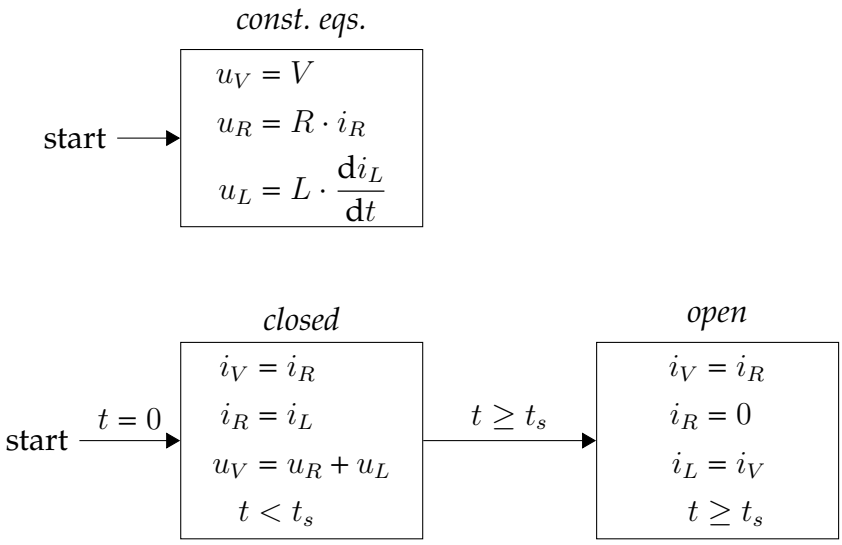


Figure 1.2: *Hybrid model of switched LR-Circuit.*

To ease our discussion, the *solution* of a mode refers to the solution of the DAE of this mode. Likewise, when referring to the initial values of a mode, we refer to the initial values of the DAE of that mode, if not stated otherwise.

Figure 1.2 displays a hybrid model of the switched RL-circuit, where $t_s > 0$, in a representation resembling that of hybrid automatons [2, 16]. This model states the switch being *closed* at the start of simulation, and remaining closed until $t = t_s$, when the switch *opens*.

Two concerns arise in the simulation of hybrid models. *First*, the index and the related index reduction, may differ between modes. *Secondly*, one has to relate the initial values of the successor mode, to the left-limit of the solution to the predecessor mode, in a way that is consistent with the successor mode, and the interpretation of the model at the mode switch.

Let us examine the hybrid model of Figure 1.2, when $t = t_s$. At that time a mode switch occurs and in the successor mode, we have $i_L = i_V = i_R = 0$. If we assume that the mode switch takes place without time advancing, and that i_L is non-zero before the mode switch, we have a *discontinuous jump* on i_L at $t = t_s$. We can model i_L using the *Heaviside step function*, defined as:

$$\Theta(t-a) = \begin{cases} 0 & \text{if } t < a \\ 1 & \text{if } t \geq a \end{cases} \tag{1.6}$$

Using (1.6) we can express the current over the inductor, over the mode switch as:

$$i_L(t) = [1 - \Theta(t - t_s)]i_L^p(t) + \Theta(t - t_s)i_L^s(t) \tag{1.7}$$

Where $i_L^p(t)$ is the solution of i_L given by the predecessor mode, comprised of *const. eqs.* and *closed*, and $i_L^s(t)$ the solution given by the successor mode, comprised of *const. eqs.*, *open*. If $i_L^p(t_s) \neq 0$, we have a discontinuity on i_L, at the mode switch, and consequently an undefined derivative on i_L at $t = t_s$. To continue our analysis we introduce the *Dirac delta function* (which is a distribution rather than a function). We will here present it informally to convey its intuition, rather than providing a formal definition using distributions.

$$\int_{-\infty}^{\infty} \delta(t - a)dt = 1 \quad \delta(t - a) = 0 \text{ if } t \neq a \tag{1.8}$$

The definition in (1.8) tells us that the *Dirac delta function* is zero everywhere except at $t = a$, but *integrates to unity* over $\mathbb{R}$. The pointwise value of $\delta(t - a)$ is not well defined at $t = a$, but from (1.8), we can conclude that it should be positive and unbounded. We can envision the *Dirac delta function*, as a function, where the area under the function is finite, but *squeezed* into a single point, positioned at a. This allows us to model *impulses*, and we can relate the *derivative of the Heaviside function* to the *Dirac delta function* as:

$$\frac{\mathrm{d}\Theta}{\mathrm{d}t}(t - a) = \delta(t - a) \tag{1.9}$$

We now use (1.8) and (1.9), to determine the derivative of (1.7) as:

$$\frac{\mathrm{d}i_L}{\mathrm{d}t}(t) = [1 - \Theta(t - t_s)]\frac{\mathrm{d}i_L^p}{\mathrm{d}t}(t) + \Theta(t - t_s)\frac{\mathrm{d}i_L^s}{\mathrm{d}t}(t) + \delta(t - t_s)(i_L^s(t_s) - i_L^p(t_s)) \tag{1.10}$$

As $i_L^s(t) = 0$, (1.10) simplifies to:

$$\frac{\mathrm{d}i_L}{\mathrm{d}t}(t) = [1 - \Theta(t - t_s)]\frac{\mathrm{d}i_L^p}{\mathrm{d}t}(t) - \delta(t - t_s)i_L^p(t_s) \tag{1.11}$$

Equation (1.11) implies that the voltage $u_L(t)$ over the inductor, will behave as $L\frac{\mathrm{d}i_L^p}{\mathrm{d}t}$ until right before the mode switch. At the mode switch, in the case where $i_L^p(t_s) \neq 0$, we get an impulse on the voltage, directed so that it would drive a current in the same direction as before the switch opened, in accordance with *Lenz law*. In the case of

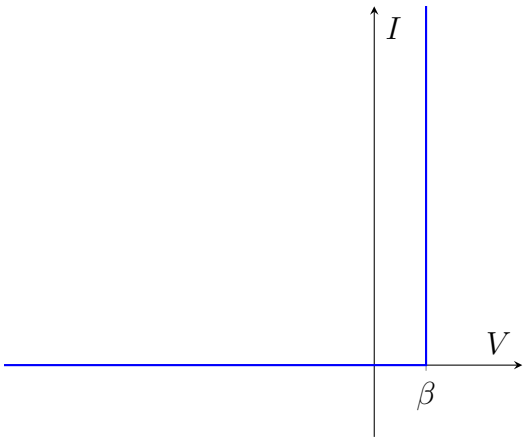


Figure 1.3: *I-V Characteristics of ideal diode with forward bias* β.

$i_L^p(t_s) = 0$, (1.11) becomes zero and i_L is continuous. In both cases, the voltage is identically zero for $t > t_s$. This is the behavior we expect from an ideal inductor.

In the above analysis, we identified an impulse occurring on u_L when the switch opened. As we shall see in Section 1.4.2, in some systems, such impulses can affect other components that in turn trigger additional structural changes.

1.4.1 Ideal Diodes

In the hybrid model considered in the previous section, the mode switch depended on the value of the independent variable. However, to express more systems we want to be able to define mode switch that depend on the dependent variables.

A diodes is an electrical component that changes behavior depending on the value of its current and voltage. If the diode is ideal, we can draw its I-V characteristics as depicted in Figure 1.3. The diode's I-V characteristics tell us that if the external voltage applied over the diode is less than β, the diode behaves as a perfect insulator. We say that the diode is *reverse biased*. On the other hand, if the external voltage across the diode is greater than β, the diode behaves as a one-way perfect conductor. We say that the diode is *forward biased*.

We can express a hybrid model of the diode as depicted in Figure 1.4. In this model, the mode switches depend on the values of u_D and i_D, which in turn depend on the solution to the active mode.

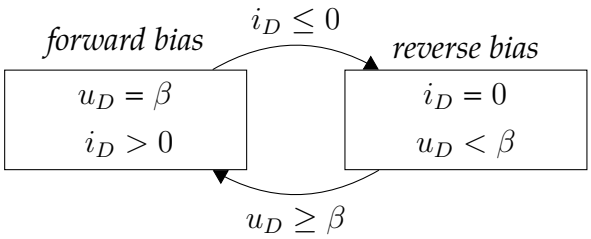


Figure 1.4: *Hybrid model of an ideal diode.*

1.4.2 Example of a switched RLD-Circuit

To give an example of a structurally varying system, where *impulses change the structure of the system*, we once again consider the switched RL-circuit of Section 1.1. The voltage impulse, resulting from opening the switch, might disturb, and even cause damage to the rest of the circuit. To prevent possible damage, we can place a diode, known as a *flyback-diode*, in parallel with the inductor, resulting in the switched RLD-circuit shown in Figure 1.5. This circuit was considered by Mosterman and Biswas [31], and later Lee [25].

Initially, with a closed switch, the diode is *reverse biased*, as a result of the voltage applied from the voltage source, and no current passes through the diode. At this point, the circuit behaves like the RL-circuit. When the switch opens, the induced voltage from the inductor will instantly change the diode into *forward biased*. This has two consequences, *first* current can now flow through the diode-inductor loop. *Secondly*, the diode applies its small *forward bias voltage* over the inductor, making the rate of change of the current constant. Therefore, the inductor can release its energy in a controlled manner by driving a current through the inductor-diode loop. Let us examine this system expressed as a

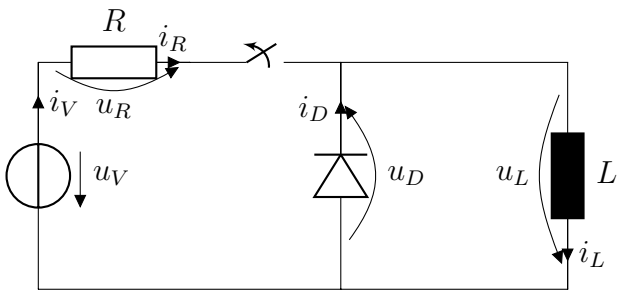


Figure 1.5: *Switched RLD-circuit*

hybrid model.

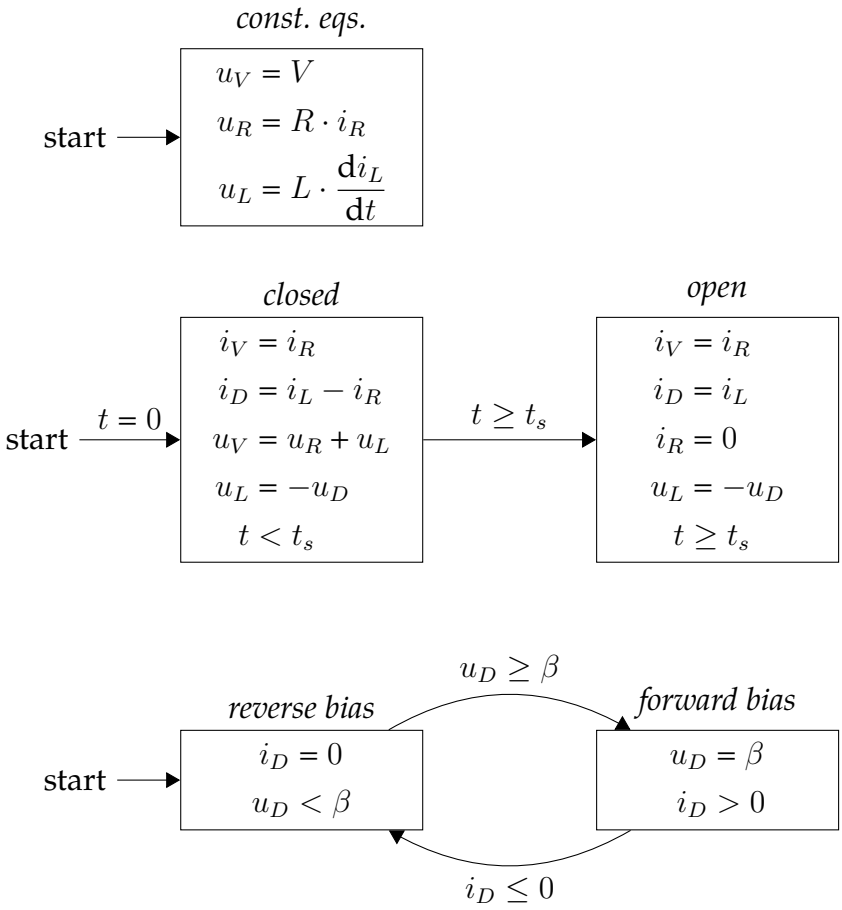


Figure 1.6: *Hybrid model of switched RLD-circuit.*

We express a hybrid model of the switched RLD-circuit in Figure 1.6 and enumerate its modes as follows:

- mode (1) consists of *const. eqs.*, *closed*, and *reverse bias*
- mode (2) consists of *const. eqs.*, *open*, and *reverse bias*
- mode (3) consists of *const. eqs.*, *open*, and *forward bias*
- mode (4) consists of *const. eqs.*, *closed*, and *forward bias*

We assume an initially $t = 0$, $i_L(0) = 0$, and that the model is initially in mode (1). Mode (1) will remain the active mode while $t < t_s$. To see this, we can find u_D for $t \in [0, t_s)$, by first solving mode (1) for i_L.

$$V = R \cdot i_R + L \cdot \frac{\mathrm{d}i_L}{\mathrm{d}t} \Leftrightarrow \frac{\mathrm{d}i_L}{\mathrm{d}t} \cdot \frac{1}{V - R \cdot i_R} = 1 \tag{1.12}$$

and as $i_D = 0 \Rightarrow i_L = i_R$, the solution to this separable ODE with the initial value $i_L(0) = 0$ is:

$$i_L = \frac{V}{R}(1 - e^{-\frac{R}{L} \cdot t}) \tag{1.13}$$

using (1.13), we get:

$$u_L = V - R \cdot i_L = V \cdot e^{-\frac{R}{L} \cdot t} \geq 0 \Rightarrow u_D < \beta \quad \forall t \in [0, t_s) \tag{1.14}$$

At $t = t_s$, a mode switch occurs and mode (2) becomes the successor mode. With a similar analysis as in the beginning of this section, we find that $\frac{\mathrm{d}i_L}{\mathrm{d}t}$ has a negative impulse at t_s, which implies a positive impulse on u_D, as $\frac{\mathrm{d}i_L}{\mathrm{d}t} = \frac{u_L}{L}$ and $u_L = -u_D$. We say that the impulse propagates in the DAE (or mode), which essentially corresponds to solving the DAE in the presence of impulses.

This state is a transient state, denoted by Mosterman and Biswas [31] as a mythical state, as $u_D > \beta$, which results in an immediate mode switch to mode (3). The model spends zero time in this state, and from the physical systems point of view, the transition occurs from the mode switch to mode (1) to mode (3) directly, and the state variables never acquires the values from the mode switch to mode (2). The diode-inductor loop appears instantly, and consequently the inductor does not induce an impulse voltage. Thus, when calculating consistent initial values for the state variables in mode (3), we should base these calculations on the values on the state variables right before the mode switch in mode (1) [31].

1.4.3 Impulse Analysis

We saw in the previous section (1.4.2) how impulses can be an integral part of the behavior of a system. Methods for handling discontinuities and impulses on the state variable exist for *non-linear circuits* (see for example Yuan and Opal [44]).

Algorithm 1 outlines the key steps in their treatment of impulses in response to mode switch. In accordance with the discussion by Mosterman and Biswas [31], this algorithm does not update the state variables when a mode is transient (mythical), as seen at step 1.

Algorithm 1: Impulse Analysis

1. Generate impulses (and propagate these impulses) in response to a mode switch, based on values on the state variables right before the mode switch from last non-transient mode;
2. If the impulses in 1 result in a new mode switch, go to 1;
3. Calculate consistent initial values for current mode based on impulses in 1;
4. If initial values in 3 result in a new mode switch, go to 1, otherwise continue continuous-time simulation of current mode;

1.5 Background on EOO Languages

We introduced EOO modeling languages in Section 1.1, a class of modeling languages constructing models from smaller connected sub-models, somewhat analogous to objects in the objects-oriented programming paradigm. Some EOO languages support *parametrized models*, *inheritance*, and *re-definitions* [27, 45]. Another class of EOO languages are instead functional languages, supporting these features through *higher-order functions* [8, 10, 23].

The elaboration is the procedure of transforming the EOO model into a *global* set of equations representing a DAE. This procedure typically includes type-checking and collapsing of the instance hierarchy. Furthermore, the topological equations are deduced from the connection semantics [9, 21], as part of the elaboration. In practice, the output from the elaboration can be a sort of *hybrid* DAE, which in addition to equations contains other constructs such as `if then else` expressions, to govern non-continuous behavior.

After the elaboration, index reduction is typically performed and possibly other symbolical transformations on the DAE (or *hybrid* DAE), to bring it to a form suitable for numerical solving. We use the term equation transformation to denote this procedure. The final step before numerical solving is to find consistent initial values on all state variables [8].

1.5.1 Example of a EOO model

In Listing 1.1, we define a EOO model of the switched RL-circuit we discussed in Section 1.1, omitting the electrical switch, in the language *Modelyze* [10, 11]. We define the inductor in Listing 1.2, where the last two arguments of type `Electrical` on line 1, represent nodes in the topology of the model. On line 2, and line 3, we create two new fresh symbols for the dependent variables representing the current and voltage over the inductor. On line 4, `Branch` associates the current and voltage of the inductor with a position in the topology of the model. On line 5, `init` assigns an initial value to the current, and on line 6, we state the constitutive equation of the inductor.

On line 2 in Listing 1.1, we define nodes in the model topology, and on line 3 to line 5, we connect models of the electrical components to these nodes.

```
1 def LR = {
2   def n1, n2, n3: Electrical;
3   VoltageSource 1. n1 n3;
4   Resistor 1. n1 n2;
5   Inductor 1. 0. n2 n3
6 }
```

Listing 1.1: *EOO model of an RL-circuit in* Modelyze.

```
1 def Inductor(L: Real, i0: Real, p: Electrical, n: Electrical) = {
2   def i_L: Current;
3   def u_L: Voltage;
4   Branch i_L u_L p n;
5   init i_L i0;
6   L * (der i_L) = u_L
7 }
```

Listing 1.2: *EOO model of an inductor in* Modelyze.

Note how each individual component *hides its definition* from the complete circuit. Note also that we do not explicitly state any topological equation, as we derived these equations from the model topology and the connection semantics during the elaboration.

```
1 u_V = 1.;
2 u_R = 1. * i_R;
3 init i_L 0.;
4 1. * (der i_L) = u_L;
5 i_V = i_R;
6 i_R = i_L;
7 u_V + u_R + u_L = 0
```

Listing 1.3: *After the elaboration of the model in Listing 1.1.*

In Listing 1.3 we show a pretty print of the elaboration of the model in Listing 1.1, where line 5 to line 7 contains the topological equations.

1.5.2 Static or Dynamic Elaboration and Equation Transformation

Both the elaboration, and the equation transformation are generally not invariant with respect to structural changes [45]. If we add or remove equations, or change the model topology, we might have to redo these two steps. Two main strategies exists to address this problem in EOO modeling languages.

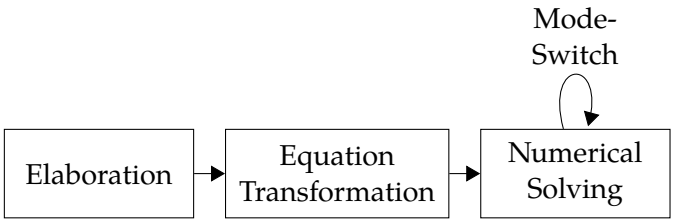


Figure 1.7: *Static elaboration and equation transformation.*

The *Static* approach, outlined in Figure 1.7, performs elaboration, and equation transformation once, before numerical solving. The benefit of this approach is that it allows compilation of the computationally expensive numerical solving procedure, which in turn, must handle all mode switches.

Explicitly exploring all mode, prior to solving, can lead to a *combinatorial explosion*. As an example, a hybrid model with n *diodes* has a minimum of 2^n modes. Instead, current research focuses on generalizing the equation transformation, and to a lesser extent the elaboration, to include hybrid models [5, 6, 18, 28].

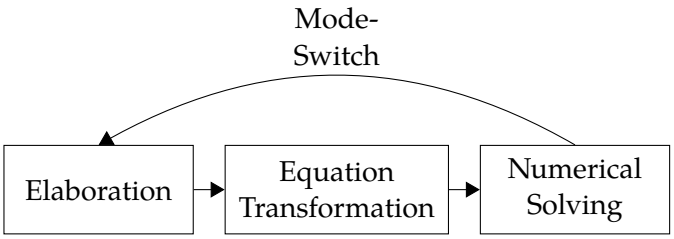


Figure 1.8: *Dynamic elaboration and equation transformation.*

Even though there have been progress in generalizing elaboration and equation transformation, the *static* approach does not offer as much flexibility when defining hybrid models, as the *dynamic* approach, outlined in Figure 1.8, which performs elaboration and equation transformation after each mode switch. The strength of this approach is flexibility. In theory we can dynamically create all modes expressible in the modeling language. Its weakness is performance, as we generally have to run the simulation using an interpreter. Research on the dynamic approach includes incorporating Just in Time (JIT) compilation [23], and reducing the extent of equation transformation after each mode switch [45].

1.6 Related Work

Modelica [20, 27] is an EOO modeling language for a wide range of complex physical systems, including electrical systems, thermal systems, and mechanical systems. *The Modelica Association*[4] maintains a freely available standard library and language specification. Modelica is widely used in the industry, and there exist several open source implementations such as *OpenModelica*[5] and *JModelica*[6], as well as commercial implementations, such as *Dymola*[7] and *MapleSim*[8].

Modelica employs *static* elaboration and equation transformation, and has some, but limited support for hybrid systems [6, 25, 28, 32, 45]. Most notably, Modelica does not handle impulses resulting from mode switches. Moreover, the number of equations and the index of DAEs

[4] www.modelica.org
[5] www.openmodelica.org
[6] jmodelica.org
[7] www.dymola.com
[8] www.maplesim.com

between modes cannot differ [6].

Mattsson, Otter, and Elmquvist [28] proposes an extension to Modelica, where one can define multi-mode components using *state machines*, which involves modifications to the index reduction procedure, and connection semantics [18], allowing support for a larger class of hybrid systems, with varying index. This modification maintains Modelicas static treatment of elaboration and index reduction.

Their approach allows successful modeling of e.g., ideal diodes, but impulses do not propagated through the DAE [28]. Moreover, the approach does not handle impulses resulting from mode switches [6].

Sol [45] is a research EOO hybrid modeling language accompanied by the interpreter *Solsim*, employing *dynamic* elaboration and equation transformation. *Sol* intentionally resembles Modelica, but the use of an interpreter allows for a higher flexibility in the creation of modes, where modes can have varying index.

The main contribution, as we see it, of Sol and Solsim, is what the author denotes as Dynamic DAE Processing (DPP). By maintaining a *causality graph*, on can confine the extent of index reduction, after each mode switch, to the parts of a DAE affected by a change of equations. In other words, the complete DAE of each mode does not have to be index reduced after each mode switch.

Sol allows modeling of impulse events, occurring between modes, where the model defines the impulse behavior explicitly. Impulse generation and propagation, resulting from steps on the state variables are not automatic.

Hydra [23], a research EOO hybrid modeling language of the Functional Hybrid Modeling (FHM) paradigm [33] is an interpreted language with dynamic elaboration, which together with the FHM paradigm, gives large flexibility in the definition of hybrid models. Hydra includes JIT-compilation of the simulation procedure. It attempts to combine the best of two worlds, *the flexibility of an interpreter, and the performance of a compiler*, by dynamically compiling the simulation code prior to numerical solving. To our knowledge, there has not yet been any attempt to evaluate its performance compared to purely compiled EOO languages.

The implementation does not handle high index problems, and impulses resulting from steps on the state variable, and impulse propagation are not considered. Nevertheless, the formalization in this thesis is much inspired by Hydra.

Modelyze [10, 11] is a *gradually typed* language, designed to serve as a host language for Domain Specific Languages (DSLs) in the field of equation-based modeling. Modelyze is a functional, interpreted language, and a continuation of the language *MKL* [8]. Part of the evaluation of Modelyze includes a prototype EOO hybrid modeling language, implemented as a DSL, and based on *hierarchical state machines* [43]. The DSL does not include any treatment of impulses, but similar to Hydra, it allows flexible definition of modes, due to the functional nature of the host language, and dynamic elaboration and equation transformation.

However, the main focus of the authors has been on formalization and implementation of the host language, and the adoption of gradual typing in equation-based modeling language. Thus, Modelyze has served as good base for the implementation in this thesis.

Benveniste et al. [5, 6] proposes a *constructive semantics* [7] based on *nonstandard analysis* [26] that, if successful, produces a simulation execution scheme, of what the authors denote as a *multi-mode* DAE (mDAE). Informally, a mDAE is a DAE where a *predicate guard each individual equation*. These predicates are functions of the state variables and modes corresponds to the DAEs, resulting from the equations where the predicates are true. The predicates will generally change values as the state variables change values during simulation, thus giving rise to hybrid models. The authors take a formal mathematical approach in their treatment of mDAEs, which includes a generalization of index reduction to encompass mDAEs, treatment of impulses resulting from mode switches, and their propagation in the mDAE.

To our knowledge there exists no implementation of their semantics. Further, it is not clear to us how to apply the authors work to a EOO language, without first exploring all modes during the elaboration.

SPICE [1] is a circuit simulation software, that has grown into a *de facto standard* when it comes to integrated circuit simulation, originally developed in the early 1970s at the University of California, Berkeley. The modeling approach in SPICE handles ideal components such electrical switches by numerical approximations.

Hybrid χ [38] is a highly formal modeling and simulation language, capable of expressing hybrid models. The language translates into a *hybrid process algebra*, and can model both continuous-time behavior, discrete behavior, or an intermix of both. Hybrid χ models continuous-time behavior using DAEs, which can be of high index. The language

bear resemblance to *Hybrid Automata* [2, 16], although more expressive. Hybrid χ is more geared towards formal verification, and consequently more transparent in its model definitions, compared to EOO languages. Therefor, it does not include automatic generation of topological equation, and index reduction requires intervention from the modeler [19].

1.7 Research Problem

EOO modeling languages stands out for their strong composability and model re-use. They are also intuitive in that physical systems are naturally expressed using DAEs, and the graphical representation of EOO models bear close resemblance to the modeled system in many domains.

Static handling of elaboration and equation transformation certainly has its advantages, as it allows compilation. However, in our opinion, for structurally varying systems, the static approach to EOO languages is still too restrictive. Furthermore, the possibility of JIT-compilation further advocates a dynamic approach to elaboration, and equation transformation.

As we saw in the previous section (1.4.2), impulses can be an integral part of the behavior of a system. This is certainly true for electrical circuits, but also in the mechanical domain, e.g. modeling of collisions and friction between rigid bodies [25].

Despite this and to the best of our knowledge, automatic generation, propagation, and handling of impulses due to mode switches is not incorporated into any EOO modeling languages. One could argue that impulses should be treated explicitly, but it weakens the composability and model re-use features of EOO languages. As we discussed in Section 1.4.2, methods for handling discontinuities and impulses due to mode switches already exists for non-linear circuits.

> This thesis considers the problem of formalizing a semantics for hybrid modeling in a *dynamic* EOO modeling language, including automatic impulse analysis.

Note that the term *dynamic* refers to dynamic elaboration and equation transformation, and the impulse analysis is according to Algorithm 1.

1.8 Delimitations

We make the following delimitations:

- We assume the existence of functions performing step 1 and step 3 in Algorithm 1, given appropriate inputs, if at all possible. We collectively refer to these two functions as impulse solving.
- We assume the possibility of index reduction of DAEs, of arbitrary index, to an index appropriate for solving, if at all possible.
- We assume we can find a unique solution to an DAE, of appropriate index, if such a solution exists.
- We assume the existence of a connection semantics, that can deduce the topological equations, given an appropriate description of a model topology.

1.9 Research Method

We formalize a *meta-modeling language* (Chapter 2), fulfilling the criterion's stated in the research problem in Section 1.7. We then implement this *meta-modeling language* as a DSL in the host language *Modelyze* (Chapter 3). Finally, we evaluate the expressiveness of the implementation, on a set of hybrid models, that represents corner-cases in the electrical, and mechanical domain (Chapter 4). This includes the switched RLD-circuit defined in Figure 1.2. Moreover, we model other systems exhibiting discontinuities and impulses on the state variables due to structural changes. In other words, we want to evaluate if we can model these systems and attain physically sensible simulation results in our implementation, and thus if our proposed semantics are capable of expressing such systems.

1.10 Contribution

- We formalize a *meta-modeling language*, named λ_{DILL}, with dynamic elaboration and equation transformation. λ_{DILL} includes an impulse analysis capable of modeling several electrical circuits and circuit like systems, exhibiting discontinuities and impulses.

Moreover, we discuss an extension to this impulse analysis to allow modeling of inelastic collisions.

- We implemented λ_{DILL} as a DSL named **DILL**, in the host language *Modelyze*. This implementation includes an connection semantics, based on *linear graph theory*, and impulse solving, based on *backwards Euler*.

1.11 Sustainability, Ethical Aspects and Societal Relevance

Modeling and simulation, considered in a broad sense are relevant in regards to sustainability and society. Computer aided simulations generally have a smaller environmental footprint than physical experiments. Further, modeling and simulation provide a tool for researchers and engineers to analyze and draw conclusions about physical systems, conclusion which might impact society.

This thesis contribute to an increased use of computer aided simulation, as we try to extend the application of modeling and simulation to larger class of systems, and make it more accessible. However, we estimate this contribution to be minimal. We see no ethical aspects to this thesis.

Chapter 2

Design

In this chapter we formalize our hybrid EOO language. Our formalization will take the form of a *meta-modeling language* named λ_{DILL}, consisting of two parts:

1. We represent the main building blocks of our hybrid modeling language such as equations, modes, and the topology using algebraic data types. We choose to base our formalization on algebraic data types as it allows us to map the implementation close to the formalization, increasing the confidence in our evaluation.

2. An enriched λ-calculus modeling the rest of the EOO language such as `if then else` expressions, function definitions, and function application. This enriched λ-calculus will also be the language we use to define the semantics of our algebraic data types, together with some basic mathematical operations.

In Section 2.1, we define some primitive semantic domains. In Section 2.2, we define the syntax of the enriched λ-calculus. We assume the enriched λ-calculus has well defined semantics and only discuss the semantics briefly and informally. In Section 2.3, we give a brief introduction to algebraic data types. For a more complete introduction to both λ-calculus and algebraic data types, see for example Peyton Jones [35] (where algebraic data types are discussed under the name Structured Types). In Section 2.4, we formalize modes using algebraic data types, and in Section 2.5, we complete the formalization of λ_{DILL}.

$$
\begin{aligned}
e &\to x \mid c \mid \lambda x.e \mid e_1\ e_2 \mid \texttt{let}\ p\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \\
&\quad \mid \texttt{if}\ b\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid (e_1, e_2, \ldots, e_n) \\
p &\to x \mid (x_1, x_2, \ldots, x_n)
\end{aligned}
$$

Figure 2.1: *Enriched λ-calculus.*

2.1 Primitive Semantic Domains

We start by defining some primitive semantic domains. To represent the unbound values of impulses on the state variable, we extend the real numbers with the symbols for positive and negative infinity $\pm\infty$. Thus we define $\mathbb{R}^* = \mathbb{R} \cup \{-\infty, +\infty\}$.

Let $\mathbb{X}$ be the set of real-valued dependent variables and let $\mathbb{B}$ be the set of Boolean values $\{true, false\}$.

We model undefined behavior and non-termination with the symbol $\perp$. For a domain $\mathbb{A}$, $\mathbb{A}_\perp$ denotes $\mathbb{A} \cup \{\perp\}$. We also have to define operators over domains containing $\perp$. For an operator op, defined on the domain $\mathbb{A}$, let $op_\perp$ be an operator defined on $\mathbb{A}_\perp$, such that $op_\perp$ returns $\perp$ if any of its arguments are $\perp$, otherwise $op_\perp$ returns the same value as op.

2.2 Enriched λ-calculus

In formalizing our semantics, we make use of the enriched lambda calculus defined in Figure 2.1, where $x \in X$ are variables, c are constants, including real numbers, natural numbers, Boolean values and primitive functions. Further, $\lambda x.e$ is the lambda abstraction, and $e_1\ e_2$ is function application. Moreover, $b \in \mathbb{B}_\perp$ and p are patterns,
and $(e_1, e_2, \ldots, e_n)$ and $(x_1, x_2, \ldots, x_n)$ are n-tuples. The expression $\texttt{let}\ p\ \texttt{=}\ e_1\ \texttt{in}\ e_2$ binds e_1 to the variable or tuple given by p in e_2, and we interpret the `if then else` expression in the usual way.

To handle $\perp$, we define $\texttt{if}\ b\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \doteq \perp$ if $b = \perp$. Similarly, we define $\texttt{let}\ p\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \doteq \perp$ if $e_1 = \perp$.

2.3 Algebraic Data Types

We introduce algebraic data types, that we from here on will refer to as data types, through an example. In Listing 2.1, `List` A is the name of a data type encoding a list of items of type A, where A denotes a generic type variable and `::=` denotes a definition. The right-hand side, `nil | cons` A `(List` A`)`, is the definition of the `List` A. We think of `|` as an or saying `List` A can be either `nil` or `cons` A `(List` A`)`. We refer to each of these constructs as *terms* and call `nil` and `cons` *constructors*. The constructor `nil` has zero *fields* and `cons` has two *fields*, the first of type A, and the second of type `List` A. We write types in uppercase camelcase and constructors in lowercase. Sometimes we make use of *infix* constructors, which we denote by surrounding the constructor in parentheses.

```
List A ::= nil | cons A (List A)
```

Listing 2.1: *Data type representing a list.*

To define a semantics for `List` A in 2.1, we define functions using pattern matching as shown in Listing 2.2. The first function returns the first element of a list, and the second function returns the list resulting from removing the first element of a list. We sometimes use the *wildcard* _ to denote match everything, and we evaluate the patterns in the order from top to bottom.

```
head : (List A) → A⊥
head nil ≐ ⊥
head cons a as ≐ a

tail : (List A) → (List A)⊥
tail nil ≐ ⊥
tail cons a as ≐ as
```

Listing 2.2: *Pattern matching on data types.*

Sometimes we will use *denotational semantics* to define the meaning of a data type, by a meaning function denoted by $\mathcal{M}$. This meaning function maps the *terms* of the data type to some mathematical representation describing its meaning. As an example, Listing 2.3 shows the definition and meaning of the data type `SimpleAdd`. For a more thorough introduction on denotational semantics see for example Bruni and Montanari [13].

```
SimpleAdd ::= 0 | 1 | add SimpleAdd SimpleAdd
```

$$\mathcal{M}[\![\texttt{0}]\!] \doteq 0$$
$$\mathcal{M}[\![\texttt{1}]\!] \doteq 1$$
$$\mathcal{M}[\![\texttt{add}\ s_1\ s_2]\!] \doteq \mathcal{M}[\![s_1]\!] + \mathcal{M}[\![s_2]\!]$$

Listing 2.3: *Meaning function example.*

2.4 Modes

We proceed with the formalization of λ_{DILL} by formalizing a mode. In essence, a mode will consists of a set of constitutive equations, a model topology, and some conditions on the state variables defining when a mode switch should occur. Associated to each mode is a DAE, resulting from combining the constitutive equation with the topological equation, resulting from the connection semantics and the model topology.

2.4.1 Example: A Mode of The Switched RLD-Circuit

As an example, we can define a mode for the switched RLD-circuit we discussed in Section 1.4.2, for the case when the electrical switch is closed and the diode is *reverse biased*. We show this circuit in Figure 2.2.

$$\begin{aligned} u_V &= V \quad u_R = R \cdot i_R \quad u_L = L \cdot \frac{\mathrm{d}i_L}{\mathrm{d}t} \\ i_D &= 0 \quad \frac{\mathrm{d}t}{\mathrm{d}t} = 1 \end{aligned} \tag{2.1}$$

$$\begin{aligned} u_D &< \beta \\ t &< t_s \end{aligned} \tag{2.2}$$

In (2.1) and (2.2), V, R, L, β, and t_s are constants. We define the constitutive equations in (2.1). We also include the independent variable time as a dependent variable, as time governs the state of the electrical switch. We define conditions in (2.2), originating from the diode and the electrical switch and governing when the mode is valid. Figure 2.2 defines the model topology.

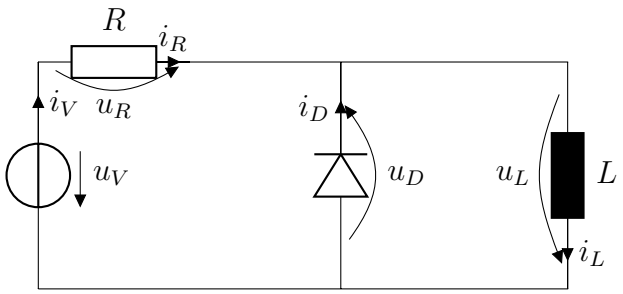


Figure 2.2: *RLD-circuit*

If this mode becomes the active mode during simulation, and the model is not in a transient state, we want to find the continuous-time solution to this mode. Let us denote the topological equations, resulting from Figure 2.2 and the connection semantics, as E_t. A solution to (2.1) together with E_t has to respect the inequalities in (2.2), because otherwise this mode, and consequently its DAE is no longer valid. We want this solution to be *maximal* in the sense that the solution should be defined over as large interval on the independent variable t as possible. We also need to let this solution evolve just enough to violate the first (or first few if they happen to occur at the same time) condition(s) in (2.2), to find the next mode if there is one.

2.4.2 Differential Algebraic Equations with Conditions

In this section, we give a more formal definition of what we mean by a solution to a mode.

$$F(\mathbf{x}, \frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t}, \mathbf{w}, t) = \mathbf{0}$$

We restate Equation (1.4) here for convenience. To make the notation a bit more compact we can form a new vector $\mathbf{z}$, containing the elements of $\mathbf{x}$, $\mathbf{w}$, and a new dependent variable χ. Replacing all occurrences of t in (1.4) with χ and adding the equation $\frac{\mathrm{d}\chi}{\mathrm{d}t} = 1$, we can re-write (1.4) as:

$$F_c(\mathbf{z}, \frac{\mathrm{d}\mathbf{z}}{\mathrm{d}t}) = \mathbf{0} \tag{2.3}$$

where, $\mathbf{z} \in \mathbb{R}^{n+m+1}$, $\frac{\mathrm{d}\mathbf{z}}{\mathrm{d}t} \in \mathbb{R}^{n+m+1}$, and $F_c : D_F \subseteq \mathbb{R}^{n+m+1} \times \mathbb{R}^{n+m+1} \rightarrow \mathbb{R}^{n+m+1}$. As we want to model hybrid models, we add conditions to the dependent variables and extend the definition of (2.3) into:

$$F_c(\mathbf{z}, \frac{\mathrm{d}\mathbf{z}}{\mathrm{d}t}) = \mathbf{0} \tag{2.4a}$$

$$G(\mathbf{z}, \frac{\mathrm{d}\mathbf{z}}{\mathrm{d}t}) > \mathbf{0} \tag{2.4b}$$

where, $G : D_G \subseteq \mathbb{R}^{n+m+1} \times \mathbb{R}^{n+m+1} \to \mathbb{R}^{n+m+1}$. We call equations on the form (2.4) Conditional Differential Algebraic Equations (CDAE), and refer to G as a Differential Algebraic Conditions (DAC). We define a solution to (2.4) as follows:

$$\hat{\mathbf{z}} : I \subset \mathbb{R} \to \mathbb{R}^{n+m+1}$$

A function $\hat{\mathbf{z}}$ is a *solution* to (2.4) on the half open interval $I = [t_0, t_f)$ if:

1. $\hat{\mathbf{z}}$ is differentiable on the open interval (t_0, t_f) and right-differentiable on t_0.
2. $\hat{\mathbf{z}}(t)$ fulfills (2.4) for all $t \in I$

Given initial values $\hat{\mathbf{z}}_0$ and $\frac{\mathrm{d}\hat{\mathbf{z}}_0}{\mathrm{d}t}$, we say that $\hat{\mathbf{z}}$ is a *solution* to the Conditional Initial Value Problem (CIVP), consisting of (2.4), $\hat{\mathbf{z}}_0$, and $\frac{\mathrm{d}\hat{\mathbf{z}}_0}{\mathrm{d}t}$, if $\hat{\mathbf{z}}$ is a solution to (2.4), $\hat{\mathbf{z}}_0 = \hat{\mathbf{z}}(t_0)$, and $\frac{\mathrm{d}\hat{\mathbf{z}}_0}{\mathrm{d}t} = \frac{\mathrm{d}\hat{\mathbf{z}}}{\mathrm{d}t}(t_0)$. Finally, we want to make sure that the interval I is as long as possible. We say that $\hat{\mathbf{z}}$ is a *maximal solution* to a CIVP, if for all *solutions* to this CIVP with intervals $I' = [t_0, t'_f)$, it holds that $t'_f \leq t_f$.

We also need to find the values on the dependent variables at t_f when the DAC is first violated. If $\hat{\mathbf{z}}$ is a *maximal solution* to a CIVP, the *invalidating extension* of $\hat{\mathbf{z}}$ is a function $\hat{\mathbf{z}}_e : [t_0, t_f] \subset \mathbb{R} \to \mathbb{R}^{n+m+1}$ such that $\hat{\mathbf{z}}(t) = \hat{\mathbf{z}}_e(t)$ for all $t \in I$ and $\hat{\mathbf{z}}_e$ is left-differentiable and fulfills (2.4a) on $[t_0, t_f]$.

2.4.3 State Variables

We define a data type representing state variables. As we discussed in Section 1.5, the state variables include the dependent variables, either differentiated in the first order or in the zeroth order (equivalent to no differentiation). We defined this data type in Listing 2.4 and its meaning in Listing 2.5.

```
SVar ::= d_0 𝕏 | d_1 𝕏
```

Listing 2.4: *Data type representing state variables.*

$$\mathcal{M}[\![\texttt{d\_0}\ x]\!] \doteq x$$
$$\mathcal{M}[\![\texttt{d\_1}\ x]\!] \doteq \frac{\mathrm{d}x}{\mathrm{d}t}$$

Listing 2.5: *Meaning function over state variables.*

2.4.4 Expressions

In Listing 2.6 we define expressions `Expr`, parameterized by type variable V. For brevity, we use an infix type constructor on line 4, and we define the meaning of `Expr` in Listing 2.7. The constructors `uop` and `bop`, on line 3 and 4, represents unary and binary operators over $\mathbb{R}$. We require these operators to remain continuous and bounded on continuous and bounded input. This is to ensure that the solution to the CDAE of a mode is well-behaved and does not jump between mode switches. In Listing 2.7, *uop* and *bop*, on line 3 and 4, are the appropriate mathematical operators of `uop` and `bop`, respectively.

```
1 Expr V ::= const ℝ
2           | var V
3           | uop (Expr V)
4           | (bop) (Expr V) (Expr V)
```

Listing 2.6: *Data type representing expressions.*

$$\mathcal{M}[\![\texttt{const}\ r]\!] \doteq r$$
$$\mathcal{M}[\![\texttt{var}\ v]\!] \doteq \mathcal{M}[\![v]\!]$$
$$\mathcal{M}[\![\texttt{uop}\ e]\!] \doteq uop\ \mathcal{M}[\![e]\!]$$
$$\mathcal{M}[\![e_1\ \texttt{bop}\ e_2]\!] \doteq \mathcal{M}[\![e_1]\!]\ bop\ \mathcal{M}[\![e_2]\!]$$

Listing 2.7: *Meaning function over expressions.*

2.4.5 Equations and Inequalities

The data types in Listing 2.8 represents systems of equations and inequalities, where we have parameterized these data types with the variable type.

```
1  Eqs V    ::= (=) (Expr V) (Expr V)
2             | nil
3             | (;) (Eqs V) (Eqs V)
4
5  Ieqs V   ::= (>) (Expr V) (Expr V)
6             | nil
7             | (;) (Ieqs V) (Ieqs V)
```

Listing 2.8: *Data type representing systems of equations and systems of inequalities.*

In Listings 2.9 and 2.10, we define the meaning of the data types shown in Listing 2.8 as sets of equations and inequalities.

$$\mathcal{M}[\![e_1 \texttt{ = } e_2]\!] \doteq \{\mathcal{M}[\![e_1]\!] = \mathcal{M}[\![e_2]\!]\}$$
$$\mathcal{M}[\![\texttt{nil}]\!] \doteq \emptyset$$
$$\mathcal{M}[\![e_1 \texttt{ ; } e_2]\!] \doteq \mathcal{M}[\![e_1]\!] \cup \mathcal{M}[\![e_2]\!]$$

Listing 2.9: *Meaning function over systems of equations.*

$$\mathcal{M}[\![e_1 \texttt{ > } e_2]\!] \doteq \{\mathcal{M}[\![e_1]\!] > \mathcal{M}[\![e_2]\!]\}$$
$$\mathcal{M}[\![\texttt{nil}]\!] \doteq \emptyset$$
$$\mathcal{M}[\![e_1 \texttt{ ; } e_2]\!] \doteq \mathcal{M}[\![e_1]\!] \cup \mathcal{M}[\![e_2]\!]$$

Listing 2.10: *Meaning function over systems of inequalities.*

2.4.6 CDAEs, States and Continous Simulation Traces

Using the definitions in 2.8, we define data types representing DAEs, DACs, and CDAEs, as shown in Listing 2.11.

$$\texttt{DAE} \doteq \texttt{Eqs SVar}$$
$$\texttt{DAC} \doteq \texttt{Ieqs SVar}$$
$$\texttt{CDAE} \doteq \texttt{DAE} \times \texttt{DAC}$$

Listing 2.11: *Data types representing DAEs, DACs, and CDAEs.*

To denote a particular assignment of values to the state variables, we define the type `State` in Listing 2.12. Here we include $\perp$ to denote

an undefined value on a state variable. Further, we will possibly encounter unbound values on the state variables in the form of impulses and thus define a lifted version of `State` in `LState`.

$$\texttt{State} \doteq \texttt{SVar} \to \mathbb{R}_\perp$$
$$\texttt{LState} \doteq \texttt{SVar} \to \mathbb{R}^*_\perp$$

Listing 2.12: *State of state variables.*

To represent the continuous time solution of a CIVP, we define `CTrace` in Listing 2.13. The first element of this tuple is a mapping from the independent variable to `State`, over some interval $[t_0, t_f]$, representing the continuous time evolution of the state variables up to, and including the time of the first violation of the DAC. In essence a representation of the *invalidating extension* (see Section 2.4.2). We refer to this element as a mode trace. The second element in the tuple is the time of the first violation of the DAC.

$$\texttt{CTrace} \doteq (I \subset \mathbb{R} \to \texttt{State}_\perp) \times \mathbb{R}$$

Listing 2.13: *Data type representing a mode trace and its interval.*

2.4.7 Solving CIVPs

We are now ready to define a function that solves a CIVP given an interval over the independent variable. The function `solve` takes a $(d, c) \in$ `CDAE` representing a CDAE and a state $s \in$ `State`, assigning initial values to the state variables.

The last argument to `solve` is the half open interval $I = [t_0, t_f)$ over which we seek a solution to the state variables. The requested interval of simulation is encoded in the CDAE as, $\chi < t_f$ and $\frac{\mathrm{d}\chi}{\mathrm{d}t} = 1$, in addition to the initial value $\chi_0 = t_0$.

$$\texttt{solve} : \texttt{CDAE} \to \texttt{State} \to I \subset \mathbb{R} \to \texttt{CTrace}_\perp$$

Listing 2.14: *Function solving CIVPs.*

The function `solve` performs index reduction as necessary. We do not require that s assigns an initial value to all state variable in d, instead we try to finds consistent initial values with respect to (d, c) on those state variables in (d, c) with undefined initial values.

The function `solve` then finds the *maximal solution* and corresponding *extended violation* to the resulting CIVP, as defined in Section 2.4.2,

and maps these solutions to `CTrace` as appropriate. We include $\perp$ in the co-domain of `solve` to model failure and non-termination.

2.4.8 Model Topology

In Listing 2.15, we defined a model topology as a directed graph where each edge $e(t, a) \in E$ is associated with two dependent variables $t, a \in \mathbb{X}$. By $e_{ij}(t, a)$, we denote an edge directed from node i to j. We use this encoding of the topology in our implementation of the connection semantics but other encoding are of-course possible.

```
Topology ≐ G(E, V)
```

Listing 2.15: *Definition of a model topology.*

In Listing 2.16, we define the functionality of a function defining the connection semantics. This function should take the topology as input and return the resulting topological equation. We include $\perp$ in its co-domain to model failure and non-termination.

```
elab_topology : Topology → (Eqs SVar)⊥
```

Listing 2.16: *Function defining connection semantics.*

2.4.9 Data type Representing a Mode

Finally, In Listing 2.17, we define the data type of a mode. The term `edge` t a p n on line 3 represents an edge $e(t, a)$ in the model topology, directed from n to p, where nodes are enumerated by integers. The constructor `nil` on line 4 represents an empty mode or no mode, and the constructor `;` on line 5 represents a continuation.

```
1 Mode V ::= eqs (Eqs V)
2          | ieqs (Ieqs V)
3          | edge 𝕏 𝕏 ℕ ℕ
4          | nil
5          | (;) (Mode V) (Mode V)
```

Listing 2.17: *Data type representing a mode.*

Example

Using `Mode`, we can define the mode of the RLD-circuit discussed at the beginning of Section 2.4.1 as shown in Listing 2.18. Lines 2 to 6

represent the constitutive equation, line 9 and 10 represent the DAC of the mode, and lines 12 to 15 represent the model topology.

```
1  lrd_mode1 ≐ eqs (
2                (var (d_0 u_V)) = (const V);
3                (var (d_0 u_R)) = ((const R) * (d_0 i_R));
4                (var (d_1 i_L)) = ((var (d_0 u_L)) / (const L));
5                (var (d_0 i_D)) = (const 1);
6                (var (d_1 t)) = (const 1)
7              );
8              ieqs (
9                (const β) > (var (d_0 u_D));
10               (const t_s) > (var (d_0 t))
11             );
12             (edge i_V u_V 1 3);
13             (edge i_R u_R 1 2);
14             (edge i_L u_L 2 3);
15             (edge i_D u_D 3 2);
```

Listing 2.18: *Example: A mode of the switched RLD-circuit.*

2.4.10 Helper Functions

Before providing the semantics for hybrid models, we define some helper functions we will use later on. In Listing 2.19, we define a function `eval_expr` evaluating expressions to a value in $\mathbb{R}^*$ given a $s \in$ `LState`.

```
eval_expr : (Expr SVar) → LState → ℝ*_⊥
eval_expr const r ≐ λs. r
eval_expr var v ≐ λs. s v
eval_expr uop e ≐ λs. uop*_⊥ (eval_expr e s)
eval_expr e_1 bop e_2 ≐ λs. bop*_⊥ (eval_expr e_1 s) (eval_expr e_2 s)
```

Listing 2.19: *Function evaluating expressions.*

The operators $uop^*_\perp$ and $bop^*_\perp$ denotes lifted versions of $uop_\perp$ and $bop_\perp$, respectively. We assume these operators treat $\pm\infty$ appropriately, possibly returning $\perp$ if an operation is *undefined*. As an example, we show the lifted addition operator in Table 2.1.

In some cases, we need to make sure that we have bounded values and therefor define the function `unlift` in Listing 2.20.

Table 2.1: *Lifted addition operator* $(+)$*, where* $a \in \mathbb{R}$ *and* $b \in \mathbb{R}^*$.

first arg	second arg	result
$\pm\infty$	a	$\pm\infty$
a	$\pm\infty$	$\pm\infty$
∞	∞	∞
$-\infty$	$-\infty$	$-\infty$
∞	$-\infty$	$\bot$
$-\infty$	∞	$\bot$
b	$\bot$	$\bot$
$\bot$	b	$\bot$

```
unlift : ℝ*⊥ → ℝ⊥
unlift ≐ λr. if r = ±∞ then ⊥ else r
```

Listing 2.20: *Function unlifting expressions.*

Further, we defined helper functions in Listings 2.21, 2.22, and 2.23, that retrieves equations, conditions, and the model topology from a mode, respectively. On line 3 in Listing 2.23, e is an edge.

```
get_eqs : (Mode V) → (Eqs V)
get_eqs m₁ ; m₂ ≐ (get_eqs m₁) ; (get_eqs m₂)
get_eqs Eqs e ≐ e
get_eqs _ ≐ nil
```

Listing 2.21: *Function retrieving equations from a mode.*

```
get_ieqs : (Mode V) → (Ieqs V)
get_ieqs m₁ ; m₂ = (get_ieqs m₁) ; (get_ieqs m₂)
get_ieqs Ieqs e = e
get_ieqs _ = nil
```

Listing 2.22: *Function retrieving conditions from a mode.*

```
get_topology : (Mode V) → Topology
get_topology m₁ ; m₂ ≐ (get_topology m₂) ∪ (get_topology m₁)
get_topology edge t a p n ≐ {e_np(t, a)}
get_topology _ ≐ ∅
```

Listing 2.23: *Function retrieving the topology from a mode.*

2.5 Hybrid Models

In this section we first introduce left-limits in Section 2.5.1 and then define a data type representing hybrid model in Section 2.5.2. In Section 2.5.3, we give an overview of the procedure of simulating such hybrid models and in Section 2.5.4, we introduce an alternative view on time, capable of modeling both continuous and discrete events. Finally, in Section 2.5.5 to 2.5.11, we formalize the semantics of the data type of Section 2.5.2, thus completing the formalization of λ_{DILL}.

2.5.1 Left-Limits

$$v_a = -ev_b \tag{2.5}$$

We extend the representation of a state variable, we defined in Listing 2.4, with a representation of the left-limit of the state variables.

When defining equations in a mode we sometimes need the left-limit of a state variable in the predecessor mode. As an example, consider the model of inelastic collision between a body of finite mass (we call this body a ball) and a body of infinite mass and zero velocity (we call this body the ground) in (2.5). Here v_a denotes the velocity of the ball after the collision and v_b its velocity before the collision. The constant $0 < e < 1$ is the *coefficient of restitution* and governs the amount of kinetic energy lost in the collision. If $e = 0$, the ball loses all its kinetic energy and we have completely inelastic collision. If $e = 1$, the ball looses no kinetic energy and the collision is completely elastic.

In λ_{DILL} such a collision results in a mode switch because the state variables does not behave continuously at the time of collision. In this case, we need to be able to reference the left-limit of a state variable in predecessor mode. That is, the value the state variable had just before the mode switch occurred. In Listing 2.24, we define a data type extending state variables with the notion of left-limits. Here `leftlim` refers to the left-limit of the state variable in the predecessor mode, and `curr` refers to the state variable considered in the successor mode.

```
HSVar ::= leftlim SVar | curr SVar
```

Listing 2.24: *Data type representing the left-limit and current value of state variables.*

We sometimes need to evaluate expressions containing left-limits. To do this evaluation, we first reduce expressions containing left-limits to expressions without left-limits as defined by `eval_llimit_expr` in Listing 2.25. The second argument to `eval_llimit_expr`, $s \in$ `State`, holds the left-limits of the state variables in the predecessor mode. On line 4 the left-limit is evaluated and returned as a constant.

```
1 eval_llimit_expr : (Expr HSVar) → State → (Expr SVar)⊥
2 eval_llimit_expr const r ≐ λs. const r
3 eval_llimit_expr var (curr v) ≐ λs. var v
4 eval_llimit_expr var (leftlim v) ≐ λs. let r = s v in const r
5 eval_llimit_expr uop e ≐ λs. uop (eval_llimit_expr e s)
6 eval_llimit_expr e₁ bop e₂ ≐ λs.
7   (eval_llimit_expr e₁ s) bop (eval_llimit_expr e₂ s)
```

Listing 2.25: *Function reducing expression containing left-limits to one without.*

Using the functions defined in Listings 2.19 and 2.25, we define a function `eval_hexpr` in Listing 2.26, evaluating expressions containing left-limits to the extended real numbers.

```
1 eval_hexpr : (Expr HSVar) → State → LState → ℝ*⊥
2 eval_hexpr e ≐ λs_. λs₀.
3   let e′ = eval_llimit_expr e s_ in eval_expr e′ s₀
```

Listing 2.26: *Function evaluating expressions containing left-limits.*

Example

Using `HSVar`, we can express the model in (2.5) as shown in Listing 2.27.

```
(var (curr (d_0 v))) = - (const e) * (var (leftlim (d_0 v)))
```

Listing 2.27: *Example: Model of inelastic collision.*

2.5.2 Data Type Representing Hybrid Models

We define the data type representing hybrid models shown in Listing 2.28. We informally present this data type before moving on to the formal definition of its semantics.

```
1 HModel ::= switch HModel (Expr HSVar) HModel
2          | init SVar (Expr HSVar)
3          | eqs (Eqs HSVar)
4          | edge 𝕏 𝕏 ℕ ℕ
5          | (++) HModel HModel
```

Listing 2.28: *Data type representing hybrid models.*

The central *term* in this data type is the `switch` m_1 e m_2 term on line 1. Which we refer to simply as a switch. This switch is analogue to the one used in *Hydra* [23]. The interpretation of a switch is that m_1 defines the current active model (which might contain additional switches). The expression e defines an open region in the solution space of the state variables, where the model m_1 is valid. We call this region the valid region of the switch. The complement to the valid region is the invalid region.

Given a state $s \in$ `LState`, we say that a switch is in its valid region and is stable with respect to s, if we evaluate e using s and find its value to be greater than zero. The opposite of stable is unstable. Similarly, we say that a model m is stable with respect to s, if all switches in the active models m_1, considered recursive in the model, are stable with respect to s. We say that we flip a switch when we replace the switch with m_2. The procedure of flipping all unstable switches of a model is equivalent to performing a mode switch.

The meaning of the term `init` v e (line 2) is to assign the initial value, resulting from evaluating the expression e, to the state variable v. We include this term as new state variables might appear dynamically during simulation.

The term on line 3 represents equations, and the term on line 4 represents edges in the model topology, both discussed in Section 2.4. Finally, we have a continuation on line 5.

Example

As an example on the use of switches, we show in Listing 2.29 part of a model of an ideal diode defined in λ_{DILL}, which we first considered in Section 1.4.1. The state variables u_D and i_D represents the voltage and current over the diode, respectively. The argument f on line 1 is a Boolean defining whether the diode is *forward biased* or *reverse biased*.

```
1  let D = λβ. λf.
2    if f then
3      switch
4        (eqs (var (curr (d_0 u_D))) = (const β))
5        (var (curr (d_0 i_D)))
6        (D β (¬f))
7    else
8      switch
9        (eqs (var (curr (d_0 i_D))) = (const 0))
10       ((const β) - (var (curr (d_0 u_D))))
11       (D β (¬f))
12 in
```

Listing 2.29: *Example: Part of a diode model in* λ_{DILL}.

If we assume f to be $true$, the switch on line 3 defines the behavior of the diode. The semantics of switches interprets into the model behaving as $u_D = \beta$ (line 4) as long as the solution of i_D is greater than zero (line 5), otherwise we flip the switch. When we flip this switch we call D recursively (line 6), which replaces the switch with the model returned from the `else` branch in the `if then else` expression, modeling the diodes behavior when *reverse biased* (line 8 to 11).

Note that to allow recursive model definitions as in Listing 2.29, the evaluation of the last field in the `switch` constructor must be delayed and not evaluated when calling this constructor.

2.5.3 Overview of Simulation Procedure

Before we discuss the details of the semantics of `HModel`, we give an overview of the procedure of simulating a model $m \in$ `HModel` in Figure 2.3. Below follows a short description of each block in this overview. We discuss the formalization of the simulation procedure in Section 2.5.11.

Initializing Model This process can include elaboration and assignment of consistent initial values, but not treatment of impulses. We will not discuss this process in detail.

Mode switch We perform a mode switch, which is equivalent to flipping all unstable switches (Section 2.5.5).

Elaboration We retrieve the CIVP from the active mode (Section 2.5.6).

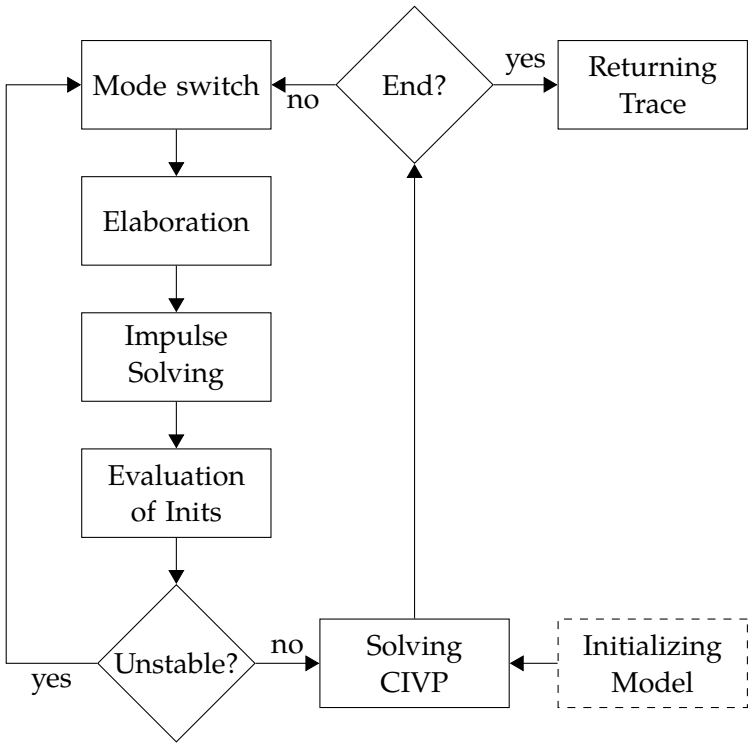


Figure 2.3: *Overview of the procedure of simulating a hybrid model in* λ_{DILL}.

Impulse Solving We retrieve the value on the state variables during the mode switch as well as consistent initial values for the successor mode (Section 2.5.7).

Evaluation of Inits We evaluate assignments of initial values to new state variables, appearing after a mode switch (Section 2.5.8).

Unstable? We check if the model is stable or unstable (Section 2.5.9).

Solving CIVP We solve the CIVP of the active mode (Section 2.4.7).

End? If we have reached the end of our simulation interval, we end the simulation, otherwise we continue.

Returning Trace We return a simulation trace by combining the mode traces from the non-mythical modes. The data type representing a simulation trace is defined in (Section 2.5.10). We regard the simulation trace as the meaning of a model and thus it is returned by the simulation function defined in Section 2.5.11.

2.5.4 Super Dense Time

To help our discussion on the simulation of hybrid models that combine the continuous time behavior with discrete changes, we extend the notion of time into a time format called superdense time [25]. Superdense time is a representation of time in form of a tuple $(t, n) \in \mathbb{R} \times \mathbb{N}$ called a time-stamp, where t represents time in an ordinary sense, also called model time, and n enumerates discrete events occurring at same time instant t. A step from (t, n) to $(t, n + 1)$ is called a micro-step. Superdense time is *lexicographically ordered* by:

$$(t_1, n_1) < (t_2, n_2) \iff t_1 < t_2 \vee (t_1 = t_2 \wedge n_1 < n_2) \tag{2.6}$$

2.5.5 Mode Switches

As discussed previously, a mode switch amounts to flipping all unstable switches in a model. For this purpose we define the function `switch_mode` in Listing 2.30. On line 5 we match a switch pattern and then either recursively call `switch_mode` on m_1, if the switch is stable, or flip the switch if unstable.

```
1  switch_mode : HModel → ((Expr MSVar) → ℝ*⊥) → HModel⊥
2  switch_mode m1 ++ m2 ≐ λs.
3    let m1' = (switch_mode m1 s) in
4    let m2' = (switch_mode m2 s) in m1' ++ m2'
5  switch_mode switch m1 e m2 ≐ λs.
6    if (s e) >⊥ 0 then let m1' = switch_mode m1 s in switch m1' e m2
7    else m2
8  switch_mode init v e ≐ λs. init v e
9  switch_mode eqs e ≐ λs. eqs e
10 switch_mode edge t a p n ≐ λs. edge t a p n
```

Listing 2.30: *Function performing a mode switch.*

2.5.6 Elaboration

The elaboration consists of retrieving the active mode and then retrieving the CDAE related to this mode.

To retrieve the active mode, we first need to remove all switches. In Listing 2.31, we define a function `collapse_switches` that collapses the switches by recursively retrieving the first and second *field* of all switches (line 5), as these fields define the active mode.

```
collapse_switches : HModel → (Mode HSVar)
collapse_switches m₁ ++ m₂ ≐
  (collapse_switches m₁) ; (collapse_switches m₂)
collapse_switches switch m₁ e m₂ ≐
  (collapse_switches m₁) ; (e > (const 0))
collapse_switches eqs e ≐ eqs e
collapse_switches edge t a p n ≐ edge t a p n
collapse_switches _ ≐ nil
```

Listing 2.31: *Function collapsing switches.*

The variables V in the mode retrieved from `collapse_switches` are of type `HSVar`. As `CDAE` consists of expressions of type `SVar`, we need to transform expressions of type `Expr HSVar` to expressions of type `Expr SVar`. This is the purpose of the function `eval_left_limits`, whose functionality we define in Listing 2.32.

```
eval_left_limits:
  (Mode HSVar) → ((Expr HSVar) → (Expr SVar)⊥) → (Mode SVar)⊥
```

Listing 2.32: *Function reducing a mode containing left-limits to one without.*

This function applies its second argument, the function f : `(Expr HSVar)`$\to$ `(Expr SVar)`$_\perp$, to all expressions $e \in$ `Expr HSVar` in the mode $m \in$ `Mode` G `HSVar`. If any of these applications returns $\perp$, then `eval_left_limits` returns $\perp$, otherwise the expression is replaced by the returned value from f.

Using `collapse_switches` and `eval_left_limits`, we can perform elaboration a model $m \in$ `HModel` as defined by the function `elaborate` in Listing 2.33.

This function first retrieves the active mode from the model on line 3. Thereafter, it retrieves the CDAE from this mode by elaborating the topology, combining this result with the constitutive equations, and conditions of the mode, on lines 4 to 5.

```
1 elaborate : (HModel G) → ((Expr HSVar) → (Expr SVar)⊥) → CDAE⊥
2 elaborate ≐ λm. λs_.
3   let m′ = eval_left_limits (collapse_switches m) s_ in
4   let eₜ = elab_topology (get_topology m′) in
5   eqs ((get_eqs m′) ; eₜ) ; ieqs (get_ieqs m′)
```

Listing 2.33: *Function retrieving the CDAE from the active mode.*

2.5.7 Impulse Solving

We abstract impulse solving into a common function `solve_impulses`. Based on the discussion by Vlach, Wojciechowski, and Opal [42], this function takes as input, the DAE $d \in \texttt{DAE}$ of the successor mode and a state $s_- \in \texttt{State}$, holding the left-limits of the state variables of the predecessor mode. This function should then return a tuple $(s_0, s_+) \in \texttt{LState} \times \texttt{State}$, where the first element s_0 holds the values of the state variable at the moment of the mode switch, which might be impulses, and the second element s_+ holds consistent initial values on the state variables for d right after the impulse has died out. Thus we assume s_+ occurs one micro-step after s_0.

We include $\bot$ in the co-domain of this function to model failure and non-termination.

```
solve_impulses : DAE → State → (LState × State)⊥
```

Listing 2.34: *Function performing impulse solving.*

2.5.8 Evaluation of Initial Values on new State Variables

In Listing 2.35, we define a function evaluating initial value assignments to state variables.

```
1 eval_inits : HModel → ((Expr MSVar) → ℝ⊥) → State → State
2 eval_inits m₁ ++ m₂ ≐ λf. λs. eval_inits m₂ f (eval_inits m₁ f s)
3 eval_inits switch m₁ e m₂ ≐ λf. λs. eval_inits m₁ f s
4 eval_inits init v e ≐ λf. λs.
5   if (s v) = ⊥ then let r = f e in s[v = r] else s
6 eval_inits _ ≐ λf. λs. s
```

Listing 2.35: *Function evaluating initial value assignments.*

On line 5, this function updates the state variables $s \in \texttt{State}$, provided as an input to this function, by mapping the state variable of a `init` term to the result of evaluating the expression of this term on the condition that this state variable is previously undefined in s. This is to ensure that we only assign the initial value to a state variable when first introduced into the model. We define the operator $[v = r]$ on $s \in \texttt{State}$ in (2.7).

$$s[v = r] \doteq \lambda v'. \begin{cases} r & \text{if } v' = v \\ s(v') & \text{otherwise} \end{cases} \tag{2.7}$$

We also define a the functionality of a function, evaluating initial value assignment on lifted state $s \in$ `LState`, in Listing 2.36. The definition of this function is analogue to that of `eval_inits`.

```
1  eval_inits_l : (HModel G) → ((Expr MSVar) → ℝ⊥) → LState → LState
```

Listing 2.36: *Lifted version of function defined in Listing 2.35.*

2.5.9 Stable Models

The function `unstable` that we define in listing 2.37, returns a Boolean saying whether a model is unstable or stable given a function mapping expressions to the extended reals. The switches are recursively checked as seen on line 3.

```
1  unstable : (HModel G) → ((Expr SVar) → ℝ*⊥) → 𝔹⊥
2  unstable m₁ ++ m₂ ≐ λf. (unstable m₁ f) ∨⊥ (unstable m₂ f)
3  unstable switch m₁ e m₂ ≐ λf. (f e) ≤ 0 ∨⊥ (unstable m₁ f)
4  unstable _ ≐ λf. false
```

Listing 2.37: *Function determining if a model is unstable.*

2.5.10 Simulation Trace

We want the simulation trace of a hybrid model to consist of the mode traces of each non-transient mode, as they appear during simulation. To represent such a simulation trace, we define a simulation trace as a list of mode traces as shown in Listing 2.38.

```
Trace ::= nil | cons (I ⊂ ℝ → State⊥) Trace
```

Listing 2.38: *Data type representing a simulation trace.*

2.5.11 Simulation of Hybrid Models

We define a state for hybrid models in Listing 2.39. We include $m \in$ `HModel` in this state as it encodes the switch configuration. For the sake of the impulse analysis, the second element of this tuple $s_{\_} \in$ `State` is

the left-limit on the state variables , and the third element $s_0 \in \texttt{LState}$ the values on the state variables at the current time. Finally, we include an interval over the model time, which defines a maximum interval for the simulation.

```
HState ≐ HModel × State × LState × I ⊂ ℝ
```

Listing 2.39: *Data type representing the state of a hybrid model.*

We are now ready to define the function `simulate` in Listing 2.40, which produces a simulation trace $\tau \in \texttt{Trace}$ given a hybrid model state $s \in \texttt{HState}$ and an accumulator. In the next section we first give an informal discussion on this function and then workout an example where we apply this function to a hybrid model.

```
1  simulate : HState → Trace → Trace⊥
2  simulate ≐ λ(m, s_, s_0, [t_0, t_f]). λτ.
3    if t_0 = t_f then τ
4    else
5      let m' = switch_mode m (λe. eval_hexpr e s_ s_0) in
6      let (dae, dac) = elaborate m' s_ in
7      let (s'_0, s_+) = solve_impulses dae s_ in
8      let s''_0 = eval_inits_l m' (λe. unlift (eval_hexpr e s_ s'_0)) in
9      let s'_+ = eval_inits m' (λe. unlift (eval_hexpr e s_ s_+)) in
10     if (unstable m' (λe. eval_hexpr e s_ s'_0)) then
11       simulate (m', s_, s''_0, [t_0, t_f]) τ
12     else if (unstable m' (λe. eval_hexpr e s_ s'_+)) then
13       simulate (m', s'_+, s'_+, [t_0, t_f]) τ
14     else
15       let (τ', t'_f) = solve (dae, dac) s'_+ [t_0, t_f] in
16       simulate (m', τ'(t'_f⁻), τ(t'_f), [t'_f, t_f]) (cons τ' τ)
```

Listing 2.40: *Function simulating a hybrid model, where $t_f'^{-}$ is the left-limit of t'_f.*

2.5.12 Discussion on the Simulation Function

When $t_0 = t_f$ on line 3 in Listing 2.40, the interval in the state $s \in \texttt{HState}$ provided as an argument to `simulate` is a single point which marks an end of simulation. Thus we have accumulated mode traces whose domains spans the interval given at the initial call to `simulate`. We refer to this initial interval as the *simulation interval*.

Otherwise, disregarding the initial call to `simulate`, we have a model $m \in \texttt{HModel}$ in s that is unstable with respect to $s_0 \in \texttt{LState}$ in s, where s_0

holds values of the state variables at time-stamp (t_0, n) for some $n \in \mathbb{N}$. Thus, we need to perform a mode switch as done on line 5. As discussed previously, a mode switch might introduce steps on the state variables, in turn resulting in impulses. Therefor, we call `solve_impulses` on line 7. We also have to handle new state variables introduced in the new mode, as done on line 8 and 9.

We check if s_0'' should trigger a new mode switch (line 10). If true, we recursively call `simulate` without advancing model time t_0. Because model time does not progress, we are in a transient state and we use the same left-limit $s_- \in$ `State` in the recursive call to `simulate`. Superdense time advance to $(t_0, n+1)$.

If instead the same is true for the initial values $s_+' \in$ `State` on line 12, we recursively call `simulate` but the choice of left-limit is not so clear. According to Algorithm 1 we are in a transient state and should treat the left-limit as in the previous case. Unfortunately, doing so would not enable us to model for example inelastic collision. The problem is that we need to define a mode that changes the physical system but where model time does not progress. This would make this state a mythical state, that according to the reasoning for electrical circuits should not affect the physical system, which to us seems like a contradiction. This might very well be a flaw in our semantics but how to solve this problem without weakening the notion of mythical states is not clear to us.

We decide to use s_+' as both left-limit and current value, which seems to be the best choice considering the model discussed in Section 4.4. Another option would be s_0'', but s_0'' might contain unbound values. Further investigation into this problem is needed. In any case the superdense time advances to $(t_0, n+2)$ as s_+' and s_0'' is separated by a microstep in our definition of `solve_impulses`.

If m is stable with respect to both s_0'' and s_+', no more mode switches should occur. Moreover, by construction, the initial values s_+' originating from `solve_impulses` are consistent with the new mode, assuming consistent initial values at line 9. Thus we can call `solve` at line 15 and advance the model time. From the solution to `solve` we retrieve the new left-limit $\tau(t_f'^-)$, where $t_f'^-$ is the left-limit of t_f and the current values on the state variable as $\tau(t_f')$.

Because the initial time t_0 at line 15 is always the end point of the domain for the previous mode trace, the domain of adjacent mode traces in the simulation trace will share end-points. Further, as we assume

the solutions of CIVPs for each non-transient mode to be continuous, we can construct a piecewise-continuous functions for the dependent variables over the modes where they are defined. We only need to decide on how to handle the overlapping points in the intervals.

In the above discussion we have assumed termination and defined behavior on all external function calls. Even under these assumptions its easy to construct models that gets stuck in a *Zeno* state [25]. We make no claims on termination and we leave this discussion as as future work.

Example

Now we continue with the example. Once again, consider the switched RLD-Circuit we analyzed in Section 1.4.2. We assume the initial mode (1), which have a *closed* electrical switch and the diode in *reverse bias*. Assume we enter `simulate` at line 2 with the following state $s \in$ `HState`:

$$s = (m, \{\ldots, i_L \mapsto i_L^-, \ldots\}, \{t \mapsto t_s, \ldots\}, [t_s, t_f])$$

where $i_L^- > 0$. For brevity, we will not explicitly state the model m. The active mode has the electrical switch *closed* and the diode in *reverse bias*. We assume $t_f > t_s$, so the simulation will not end on line 3. The mode switch on line 5 will produce m', where the electrical switch is *open* as $t \mapsto t_s$ in s_0. The elaboration on line 6 will produce a CDAE given by *const. eqs.*, *open*, and *reverse bias* in Figure 1.6.

The `solve_impulses` function on line 7 will produce s_0', where

$$s_0' = \{\ldots, u_L \mapsto \infty, u_D \mapsto -\infty, \ldots\}$$

because of the instant current drop over the inductor. We have not introduced any new state variables and the calls to `eval_inits` and `eval_inits_l` on line 8 and 9 will result in $s_0'' = s_0'$ and $s_+' = s_+$.

As $u_D \mapsto -\infty$ in s_0'' the condition in the **`if then else`** expression on line 10 evaluates to $true$ and `simulate` is recursively called on line 11. Because the model m' is unstable at the mode switch, we use s_- as the left-limit in this call.

Next, we enter `simulate` at line 2 with the state $s' \in$ `HState`, where:

$$s' = (m', \{\ldots, i_L \mapsto i_L^-, \ldots\}, \{\ldots, u_L \mapsto \infty, u_D \mapsto -\infty, \ldots\}, [t_s, t_f])$$

Now `switch_mode` will result in a model $(m')'$ with an *open* electrical switch and a *forward biased* diode. This in turn will make the call to `solve_impulses` result in:

$$s_0' = \{\ldots, i_D \mapsto i_L^-, \ldots\}$$
$$s_+ = \{\ldots, i_L \mapsto i_L^-, i_D \mapsto i_L^-, u_L \mapsto -\beta, u_D \mapsto \beta, \ldots\}$$

As before, $s_0'' = s_0'$ and $s_+' = s_+$. The model is stable at both line 10 and 12, and we proceed by solving the CDAE over the interval $[t_s, t_f)$ given the initial values s_+' at line 15. In the solution to this CIVP the current over the diode will decrease and we assume the current hits zero at time t_r. Thus the function `solve` returns a mode trace τ' representing the behavior of the state variables for the duration of this mode and a time t_r, when the DAC is first violated.

Once again `simulation` is recursively called on line 16, now with the state $s'' \in$ `HState`, where:

$$s' = ((m')', \{\ldots\}, \{\ldots, i_D \mapsto 0, u_D \mapsto \beta, \ldots\}, [t_r, t_f])$$

As $i_D \mapsto 0$ in s_0 a mode switch will occur on the diode, which will switch to *revers bias* once again. Thus `switch_mode`, `elaborate`, and `solve_impulses` results in:

$$s_0' = \{\ldots, i_D \mapsto 0, \ldots, u_D \mapsto 0, \ldots\}$$
$$s_+ = \{\ldots, i_D \mapsto 0, \ldots, u_D \mapsto 0, \ldots\}$$

As before, $s_0'' = s_0'$ and $s_+' = s_+$, and the model $(m')'$ is stable at line 10 and 12. We once again call `solve` at line 15. The solution to all state variables except t are identically zero in the resulting mode trace. Now because we include $\chi \leq t_f$, where $\chi = t$ in the CIVP in the definition of `solve` (Listing 2.14), the returned time will be t_f. This in turn makes the next recursive call to `simulate` return the accumulated simulation trace on line 3, and the simulation has ended.

Chapter 3

Implementation

I this chapter we discuss the more important details of the implementation of λ_{DILL}, we call this implementation **DILL**. In Section 3.1, we discuss the host language and syntactical differences between **DILL** and λ_{DILL}. In Section 3.2, we discuss the implementation of our connection semantics, and in Section 3.3, we discuss the implementation of impulse solving.

3.1 DILL, a DSL in Modelyze

We implement **DILL** in the host-language *Modelyze* [10, 11], which includes a DSL implementation of a EOO language. Below we list the main parts of this EOO language we re-use in our implementation.

- Index reduction using *Pantelides* algorithm.
- Finding consistent initial values and solving index 1 DAEs, which in turn makes use of the *IDA*[1] solver suite.
- Real valued expressions, both numerical and symbolic

Modelyze provides symbolic types similar but not identical to data types for defining DSLs. Because of this, and because we use some of the existing functionality in Modelyze, the types in the implementation differs somewhat from the formalization. The types in the implementation are more general and we compensate for this by input validation.

[1] https://computation.llnl.gov/projects/sundials/ida

```
1 HModel ::= switch HModel (Expr HSVar) HModel
2           | init SVar (Expr HSVar)
3           | eqs (Eqs HSVar)
4           | edge 𝕏 𝕏 ℕ ℕ
5           | (++) HModel HModel
```

Listing 3.1: *Hybrid model in* λ_{DILL}.

```
1 type HModel
2 def switch : HModel -> <Real> -> (() -> HModel) -> HModel
3 def init : <Real> -> <Real> -> HModel
4 def eande : Equations -> HModel
5 def (++) : HModel -> HModel -> HModel
```

Listing 3.2: *Hybrid model in* **DILL**.

In Listing 3.2 we show our **DILL** implementation of the data type in Listing 3.1, representing hybrid models. On line 4 in Listing 3.2, `eande` represents both equations and edges. Otherwise it should be clear from the names which symbolic type corresponds to which constructor[2]. Moreover, the arguments to the symbolic types in Listing 3.2 relates to the fields of the types in Listing 3.1, by their ordering (except for `eande`). The type `<Real>` in Modelyze (**DILL**) corresponds to both `Expr SVar` and `Expr HSVar`, where `der` denotes the first order derivative and `pre` corresponds to `LeftLim`. The real numbers `Real` and operators over real number in Modelyze extends to $\mathbb{R}^*$ by default.

```
def x, y, z: <Real>;
1 + x + der y - pre z
```

Listing 3.3: *Expression in Modelyze (***DILL***).*

```
(Const 1) + (Var (Curr (D_0 x))) + (Var (Curr (D_1 y)))
- (Var (LeftLim (D_0 z)))
```

Listing 3.4: *Corresponding expression in* λ_{DILL}.

Listing 3.3 shows an expression of type `<Real>` in **DILL** and Listing 3.4 shows the corresponding expression of type `Expr HSVar` in λ_{DILL}. The `def` expression in Listing 3.3 binds unique symbols, representing dependent variables, to the variables `x`, `y` and `z`.

[2]We use a dynamic type for the return type in the thunk of the switch. This done to circumvent an issue with restrictive typechecking

Because Modelyze uses an *eager* evaluation strategy, we use a *thunk* in the third argument to the `switch` type on line 2 in Listing 3.2, to prevent premature evaluation of the third argument of `switch`.

Thus we incorporate Modelyze's existing representation of equations and topological elements into our implementation of a mode.

```
1 Mode V ::= eqs Eqs V
2          | ieqs Ieqs V
3          | edge 𝕏 𝕏 ℕ ℕ
4          | nil
5          | (;) (Mode V) (Mode V)
```

Listing 3.5: *A mode in* λ_{DILL}.

```
1 def (=) : <Real> -> <Real> -> Equations
2 def (<) : <Real> -> <Real> -> Equations
3 def Branch : <Real> -> <Real> -> ? -> ? -> Equations
4 def nmil : () -> Equations
5 def (;) : Equations -> Equations -> Equations
6
7 def nil = mnil ()
```

Listing 3.6: *A mode in* **DILL**.

We restate the data type representing a mode in λ_{DILL} in Listing 3.5. In Listing 3.6, we show its implementation in Modelyze. The symbolic types on line 1 to 5 are analogous to the terms on the corresponding lines in Listing 3.5. On line 7, we define an alias we will encounter in later examples.

3.2 Connection Semantics

In this section we present our implementation of the the function `elab_topology`, whose functionality we defined in Listing 2.16. We convenience, we restate this functionality here in Listing 3.7

```
elab_topology : Topology → (Eqs SVar)⊥
```

Listing 3.7: *Function defining connection semantics.*

This function is responsible for associating a topological description of a model with a set of topological equations. We call this function as part of the *elaboration* (see Figure 2.3).

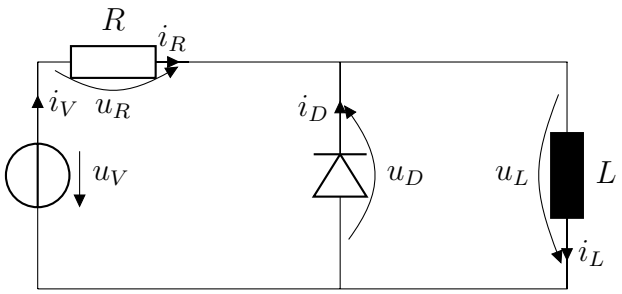


Figure 3.1: *RLD-circuit*

As discussed previously (Section 1.1 and 1.5), there exist several methods for deducing the topological equations. We choose a connection semantics based on linear graph theory [3, 30, 37]. We restrict ourselves to the domain of analog electric circuits and one-dimensional mechanical problems but this method generalizes to multi-domain problems in higher dimensions.

Table 3.1: *Across and through variables for the electrical domain and the domain of rotational mechanics.*

Domain	Across	Through
Electrical	Potential	Current
Rot. Mechanics	Rotation	Torque

In this connection semantics, we represent a topology by a directed graph, where we associate the dependent variable of each constitutive equation (e.g. component) with an edge. We identify physical quantities as being either *across* or *through* -variables. In Table 3.1 we define the choice of *across* and *through* variables of the *Trent* analogy, which we adopt in our model definitions in chapter 4. *Across* variables represent physical quantities measured in parallel, and *through* variables represent physical quantities measured in series, with a component.

The direction of the edge determines the polarity of such a measurement. The choice of *across* and *through* variables determines the graph representation of a system topology, and is usually chosen so that the two resembles each other. Using the choices of *across* and *through* variables defined by Table 3.1, we give the graph of the RLD-circuit (Figure 3.1) in Figure 3.2.

We state the in Algorithm 2, the algorithm defining the connection semantics. We explain this algorithm by discussing its application to

Algorithm 2: Connection Semantics

Data: Directed graph $G = (E, V)$, of e edges and v nodes, describing the topology of the model, where we associated each edge $(t, a) \in E$ with a through (t) and across (a) variable

Result: e topological equations

1 enumerate edges and vertices;
2 form incidence matrix **I**;
3 form **A** by performing Gaussian elimination on **I**;
4 form $\mathbf{A}' = \begin{bmatrix}\mathbf{Id}_{v-1} & \mathbf{A}_c\end{bmatrix}$, $\mathbf{t}'$ and $\mathbf{a}'$ by removing zero-rows and switching columns;
5 form $\mathbf{B}' = \begin{bmatrix}\mathbf{B}_c & \mathbf{Id}_{e-v+1}\end{bmatrix}$, where $\mathbf{B}_c = -\mathbf{A}_c^T$;
6 form e topological equations from $\mathbf{A}'\mathbf{t}' = \mathbf{0}$ and $\mathbf{B}'\mathbf{a}' = \mathbf{0}$;

the RLD-Circuit in Figure 3.1. A proof on the correctness of this algorithm is out of scope for this thesis. For a more thorough discussion we refer to Andrews [3] and Chen [15].

The input to this algorithm is the graph G depicted in Figure 3.2.

$$\mathbf{I} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} L & D & R & V \end{array} \\ \begin{pmatrix} 0 & 0 & 1 & 1 \\ 1 & -1 & -1 & 0 \\ -1 & 1 & 0 & -1 \end{pmatrix} \end{array} \tag{3.1}$$

In step 1 and 2 in Algorithm 2, we enumerate the edges and nodes of G and forms the incidence matrix **I** in (3.1). We enumerate the nodes by their label and the edges by the subscript of their corresponding dependent variables.

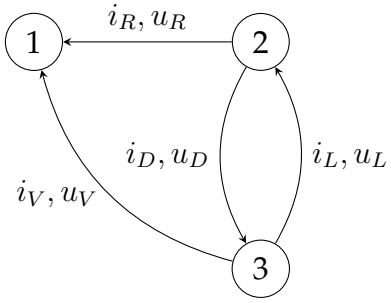


Figure 3.2: *Graph of RLD-circuit depicted in Figure 3.1.*

An incidence matrix $\mathbf{I}$ has $\mathbf{I}_{ij} = -1$, if edge j points away from vertex i, $\mathbf{I}_{ij} = 1$, if edge j points into vertex i, and $\mathbf{I}_{ij} = 0$ otherwise.

To deduce the topological equations from the graph of a system we use the vertex and circuit -postulates, which are generalizations of *Kirchhoff's* circuit laws. Given a graph, where we associate each edge with an *across* and a *through* -varaiable, the following must hold [37]:

Vertex postulate The oriented sum of *through* variables associated with edges connected to a vertex must be zero.

Circuit postulate For each closed loop of the graph, the oriented sum of *across* variables associated with the edges of the loop must be zero.

If we let $\mathbf{t}$ be a vector of *through* variables, enumerated as the columns of $\mathbf{I}$, we can express the *vertex postulate* as:

$$\mathbf{It} = \mathbf{0} \tag{3.2}$$

Using $\mathbf{I}$ from (3.1), (3.2) gives:

$$\begin{pmatrix} 0 & 0 & 1 & 1 \\ 1 & -1 & -1 & 0 \\ -1 & 1 & 0 & -1 \end{pmatrix} \begin{pmatrix} i_L \\ i_D \\ i_R \\ i_V \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \tag{3.3}$$

By using results from linear graph theory, we can genereate e independent topological equations from $\mathbf{I}$. We start by performing *Gaussian elimination* on $\mathbf{I}$, forming the matrix $\mathbf{A}$ (step 3 in Algorithm 2).

$$\mathbf{A} = \begin{pmatrix} 1 & -1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix} \tag{3.4}$$

In our example, the resulting $\mathbf{A}$ is that of (3.4). The elementary row operations preserves the relation in (3.3) thus $\mathbf{At} = \mathbf{0}$ holds. So far we have managed to produce two linearly independent equations. In order to find the third, we arrange the columns of $\mathbf{A}$ to retrieve what is knows as the *cutset* and *circuit* -equations. We swap the 2:nd and 3:rd column as well as removing the last row (which only contains zeros and does not give any useful information). This is step 4 in Algorithm 2. In our example, these operations results in the following matrix:

$$\mathbf{A}' = \begin{bmatrix}\mathbf{Id}_2 & \mathbf{A}_c\end{bmatrix} = \begin{pmatrix}1 & 0 & -1 & 1\\ 0 & 1 & 0 & 1\end{pmatrix} \tag{3.5}$$

where $\mathbf{Id}_2$ is the 2×2 identity matrix. Swapping columns is equivalent to changing the enumeration of the corresponding edges. Thus we form $\mathbf{t}' = \begin{pmatrix}i_L & i_R & i_D & i_V\end{pmatrix}^T$ and it then holds that $\mathbf{A}'\mathbf{t}' = \mathbf{0}$. We also form a vector $\mathbf{a}' = \begin{pmatrix}u_L & u_R & u_D & u_V\end{pmatrix}^T$, containing the *across* variables with the same enumeration as in $\mathbf{t}'$.

Swapping the columns of $\mathbf{A}$ to produce $\mathbf{A}'$ is equivalent to choosing a spanning tree of the corresponding graph (Theorem 2.2 in Chen [15]), where the edges corresponding to the columns of the identity matrix defines the spanning tree. In general there are multiple ways of forming $\mathbf{A}'$ given $\mathbf{A}$. Equivalently, we can choose multiple spanning trees of the graph of $\mathbf{A}$. The choice of spanning tree affects the form of the resulting topological equations and thus is a source for optimization. Here we will suffice to find some $\mathbf{A}'$, allowing us to find a sufficient number of topological equations (Corollary 2.9 in Chen [15]).

From $\mathbf{A}'\mathbf{t}' = \mathbf{0}$ we get $v - 1$ linearly independent *cutset*-equations. The theory of linear graphs allows us to retrieve the $e - v + 1$ remaining, linearly independent *circuit*-equations, directly from $\mathbf{A}'$. From the *circuit postulate* and the *principle of orthogonality* [3], it holds that $\mathbf{B}'\mathbf{a}' = \mathbf{0}$, where $\mathbf{B}' = \begin{bmatrix}\mathbf{B}_c & \mathbf{Id}_{e-v+1}\end{bmatrix}$ and $\mathbf{B}_c = -\mathbf{A}_c^T$, and we have e number of linearly independent equations.

$$\mathbf{B}'\mathbf{a}' = \begin{pmatrix}1 & 0 & 1 & 0\\ -1 & -1 & 0 & 1\end{pmatrix}\begin{pmatrix}u_L\\ u_R\\ u_D\\ u_V\end{pmatrix} = \begin{pmatrix}0\\ 0\end{pmatrix} \tag{3.6}$$

We form $\mathbf{B}'$ at step 5 in Algorithm 2, in our example resulting in $\mathbf{B}'$ given by (3.6).

$$\begin{aligned} i_L - i_D + i_V &= 0\\ i_R + i_V &= 0\\ u_L + u_D &= 0\\ -u_L - u_R + u_V &= 0 \end{aligned} \tag{3.7}$$

In conclusion, the connection semantics results in the topological equations in (3.7), which is step 6 in Algorithm 2 and we can verify that (3.7) fulfills *Kirchhoff's* circuit laws.

3.3 Impulse Solving

In Chapter 2 we defined the functionality of a function performing impulse solving, restated here in Listing 3.8. This function take as arguments the DAE of the predecessor mode and the values on the state variable just before the mode switch. The function should return a tuple containing the values on the state variable at the time of the mode switch and consistent initial values for the DAE of the successor mode.

```
solve_impulses : DAE → LState → (LState × State)⊥
```

Listing 3.8: *Functionality of impulse solving function.*

This function is called at the *impulse solving* step shown in Figure 2.3. We implement this function by adapting a numerical method based on *backwards Euler*, proposed by Yuan and Opal [44] for switched nonlinear circuits. We use *backwards Euler* as it handles impulses and finding subsequent consistent initial value [42]. We describe our implementation through an example.

We begin by stating the implicit Euler approximation for the time derivative in (3.8).

$$\frac{\mathrm{d}x_k}{\mathrm{d}t} = \frac{x_k - x_{k-1}}{h} \tag{3.8}$$

here $h \in \mathbb{R}$ is a small positive step-size.

$$\Theta_k^n = \begin{cases} 1 \text{ if } k = n \\ 0 \text{ otherwise} \end{cases} \tag{3.9}$$

$$\delta_k^n = \frac{\mathrm{d}\Theta_k^n}{\mathrm{d}t} = \frac{\Theta_k^n - \Theta_{k-1}^n}{h} = \frac{\Theta_k^n}{h} \tag{3.10}$$

We can define the discrete version of the *Heviside function* as in (3.9). Using (3.8) and (3.9) we can write the discrete *Dirac delta function* as shown in (3.10). We can observe that $\delta_k^n = 0$ if $n \neq k$ and that $\delta_k^n \to \infty$, if $h \to 0$ and $k = n$ as we expect from the *Dirac delta function*. Note that to view δ_k^n as a point-wise approximation of $\delta(x - a)$ is not sensible, as approximating an unbound value with a bound value would lead to an unbound error. We should rather use δ_k^n to determine if an impulse has occurred between the step k and $k-1$. The condition of an impulse on a state variable $x = x(h)$ is that $x(h) \to \pm\infty$ when $h \to 0$.

For our example, we once again consider the switched RLD-circuit defined in Figure 1.5 in Section 1.4.2. Assume that the electrical switch

opens at $t = t_s$. For each state variable x, let x^- denote the known left limit on the solution of x right before the mode switch occurs. Let x^0 denote the unknown value of x at the moment of the mode switch. Let x^+ denote the unknown value on the state variable right after the mode switch. We use x^+ as initial values for the successor mode.

After the mode switch, we see the resulting systems of equations, rewritten using *backwards Euler* with a small positive step h in Equation (3.11).

$$\begin{aligned} u_{V,h} &= V \\ u_{R,h} &= R \cdot i_{R,h} \\ i_{L,h} &= \frac{h}{L} \cdot u_{L,h} + i_L^- \\ u_{L,h} &= -u_{D,h} \\ i_{D,h} &= i_{V,h} = i_{R,h} = i_{L,h} = 0 \end{aligned} \tag{3.11}$$

We can solve (3.11). Specifically, we get the result $u_{L,h} = -\frac{L}{h} \cdot i_L^-$ which implies $u_{D,h} = \frac{L}{h} \cdot i_L^-$. We can conclude that we have a positive impulse on $u_{L,h}$ as $u_{L,h} \to \infty$ when $h \to 0$. Similarly, we have a negative impulse on $u_{D,h}$.

To find the consistent initial values, we could take another step forward. However, the error would be proportional to the step-size h. To improve this proportion to h^2, we can instead take a step $-h$ backwards, as shown by Yuan and Opal [44], to calculate x^+.

$$\begin{aligned} u_V^+ &= V \\ u_R^+ &= R \cdot i_R^+ \\ i_L^+ &= \frac{-h}{L} \cdot u_L^+ + i_{L,h} \\ u_L^+ &= -u_D^+ \\ i_D^+ &= i_V^+ = i_R^+ = i_L^+ = 0 \end{aligned} \tag{3.12}$$

In (3.12) we show the system of equations of the backwards step. The solutions to (3.12) are:

$$\begin{aligned} u_V^+ &= V \\ i_V^+ &= i_R^+ = i_L^+ = i_D^+ = u_R^+ = u_L^+ = u_D^+ = 0 \end{aligned}$$

Our semantics requires defined values for all state variables present in the boundaries defining the valid region of a switch. We use the

value of x^+ for any x not having an impulse at the mode switch. Thus we form x^0 as shown in (3.13).

$$x^0 = \begin{cases} x^+ + \infty & \text{if positive impulse} \\ x^+ + -\infty & \text{if negative impulse} \\ x^+ & \text{otherwise} \end{cases} \tag{3.13}$$

In our example we get the following values on the state variables at the mode switch:

$$\begin{aligned} u_V^0 &= V \\ u_L^0 &= \infty \\ u_D^0 &= -\infty \\ i_V^0 = i_R^0 = i_L^0 = i_D^0 &= u_R^0 = 0 \end{aligned}$$

The value $-\infty$ on u_D^0 will result in another mode switch, when the diode switches from *reversed biased* to *forward biased*. For completeness we show the impulse solving of this second mode switch. For brevity, we do not include the resistor and voltage source.

$$\begin{aligned} i_{L,h} &= \frac{h}{L} \cdot u_{L,h} + i_L^- \quad & u_{L,h} &= -u_{D,h} \\ i_{L,h} &= i_{D,h} & u_{D,h} &= \beta \end{aligned} \tag{3.14}$$

$$\begin{aligned} i_L^+ &= \frac{-h}{L} \cdot u_L^+ + i_{L,h} \quad & u_L^+ &= -u_D^+ \\ i_D^+ &= i_C^+ & u_D^+ &= \beta \end{aligned} \tag{3.15}$$

The forward step results in the system of equations in (3.14), and the backwards step those in (3.15). From (3.14) and (3.15), we get the following solutions:

$$\begin{aligned} i_D^+ = i_L^+ &= \frac{h}{L} \cdot \beta - \frac{h}{L} \cdot \beta + i_L^- = i_L^- \\ u_D^+ &= \beta \\ u_L^+ &= -\beta \end{aligned} \tag{3.16}$$

As there are no impulses present, (3.16) are also the values on the state variables at the mode switch. Further, the model is now stable and solving of the appropriate CDAE can proceed with initial values given by (3.16).

3.3.1 Solving Backwards Euler and Detecting Impulses

In general we have to find a solution to a system of non-linear equations when finding solutions in the forward and backwards steps. In our implementation, we find the solution numerically using the *KINSOL*[3] solver suite. Moreover the condition that an impulses occurs on a state variable $x = x(h)$, if $x(h) \to \pm\infty$ when $h \to 0$, cannot easily be treated by means of symbolic manipulation. In our implementation we try to find these impulses numerically.

3.3.2 Undefined State Variables and Equations

The successor mode might contain dependent variables, undefined in the predecessor mode. Any equation containing the derivative of such a dependent variables will also be undefined during the forward step in the impulse analysis. Thus we exclude these equations before we proceed with impulse solving.

3.3.3 High Index Problems & Backwards Euler

The *backwards Euler* discretization scheme applied in our impulse solving is not suitable for higher index DAEs [22, 29]. We could try to apply index reduction to our problem before the impulse solving. However, consider the following problem discussed by Benveniste et al. [6].

$$\begin{aligned} \frac{\mathrm{d}\omega_1}{\mathrm{d}t} &= \alpha_1\omega_1 + \beta_1\tau_1 \\ \frac{\mathrm{d}\omega_2}{\mathrm{d}t} &= \alpha_2\omega_2 + \beta_2\tau_2 \\ \omega_1 &= \omega_2 \end{aligned} \tag{3.17}$$

where $\alpha_{1,2}$ and $\beta_{1,2}$ are constants. Applying index reduction to (3.17) using *Pantelides* algorithm yields (3.18).

$$\begin{aligned} \frac{\mathrm{d}\omega_1}{\mathrm{d}t} &= \alpha_1\omega_1 + \beta_1\tau_1 \\ \frac{\mathrm{d}\omega_2}{\mathrm{d}t} &= \alpha_2\omega_2 + \beta_2\tau_2 \\ \frac{\mathrm{d}\omega_1}{\mathrm{d}t} &= \frac{\mathrm{d}\omega_2}{\mathrm{d}t} \end{aligned} \tag{3.18}$$

[3] https://computation.llnl.gov/projects/sundials/kinsol

Solving (3.18), after applying the *backwards Euler* scheme using *KINSOL* with $\alpha_1 = -0.1$, $\alpha_2 = -0.2$, $\beta_1 = 0.3$, and $\beta_2 = 0.4$; initial guesses $\omega_1 = 1$, $\omega_2 = 2$, and $\tau_1 = \tau_2 = 0$ gives the result in Table 3.2.

ω_1	ω_2	τ_1	τ_2
1.0	2.0	-4.3	4.3

Table 3.2: *An approximate solution to* (3.18).

This result is clearly not consistent with (3.17) as $1 \neq 2$. The algebraic constraint $\omega_1 = \omega_2$ is not present in (3.18) but appending this constraint to (3.18) would make this system over-determined. Finding approximate solutions to over-determined systems could be attempted using for example *Gauss-Newton* methods [24]. Moreover, introduction of new state variables, in the form of higher order derivatives as a result of the index reduction, has to also be considered for high index problems.

We leave combining impulse solving and high index problems as future work and refrain from performing index reduction during the impulse solving in our implementation. In effect this limits us to index 1 problems when we want to apply impulse analysis. This is a severe limitation in the implementation but we deem our implementation sufficient to evaluate the semantics of Chapter 2.

Chapter 4

Evaluation

In this chapter, we evaulate our implementation by modeling and simulating a set of structurally varying system as hybrid models. In all simulations we have used at step-size $h = 0.01$.

We mainly define models in a EOOs manner, building the models from models of their individual components placed in a model topology. Later on we will also discuss two non EOO, one-dimensional, mechanical models, exposing some weaknesses in our semantics. However, we start with the good news.

Table 4.1: *Across and through variables for the electrical domain and the domain of rotational mechanics.*

Domain	Across	Through
Electrical	Potential	Current
Rot. Mechanics	Rotation	Torque

We will discuss hybrid EOO models from two different domains, the electrical domain and the domain of rotational mechanics. Table 4.1 shows the definitions of *across* and *through* variables, discussed in Section 3.2, for these two domains.

We start by defining a model of a dissipator in Listing 4.1. The last two arguments `p` and `n`, on line 1, are two nodes in the model topology. The `Branch` construct defines a directed edge from `n` to `p` and associates the *through* and *across* variables `t` and `a` to this edge. On line 4, we state the constitutive equations of the component.

```
1 def Dissipator(C: Real, t: Through, a: Across, p: Node, n: Node) = {
2   eande (
3     Branch t a p n;
4     C * a = t
5   )
6 }
7
8 def Damper = Dissipator
9 def Resistor = (fun R: Real -> Dissipator (1. / R))
```

Listing 4.1: **DILL** *model of a dissipator.*

We defined the interpretation of the dissipator in our two domains on line 8 to 9; a damper, in the domain of rotational mechanics, and a resistor in the electrical domain. The anonymous function on line 9 is there to ensure that the constitutive equations of the resistor follows the convention $u = R \cdot i$. The dissipator is a single-mode component free from any switches. We define the signature of some additional components in Listing 4.2, which are all single-mode components. Their definitions should be clear from their naming, otherwise refer to Appendix A.

```
1 def AcrossGenerator(C: Real, t: Through, a: Across, p: Node, n: Node)
2 def VoltageSource = AcrossGenerator
3 def Motor = AcrossGenerator
4
5 def ThroughGenerator(C: Real, t: Through, a: Across, p: Node,
6   n: Node)
7 def CurrentSource = ThroughGenerator
8 def ConstantForceSpring = ThroughGenerator
9
10 def AcrossStorage(C: Real, t: Through, a: Across, p: Node, n: Node)
11 def Capacitor = AcrossStorage
12 def Mass = AcrossStorage
13
14 def ThroughStorage(C: Real, t: Through, a: Across, p: Node, n: Node)
15 def Inductor = ThroughStorage
16 def Spring =  ThroughStorage
```

Listing 4.2: *Single-mode components in* **DILL**.

In Listing 4.3 we define a *onewaythroughstop*, which in the electrical domain corresponds to an ideal diode. The *onewaythroughstop* does not

have a clear interpretation in the domain of rotational mechanics. In fluid mechanics it models a one-way valve.

The *onewaythroughstop* defines two modes. We will here describe this component interpreted as a diode. Assuming `open` is *true*, `d` returns the switch on line 6, acting like a voltage source (line 7) as long as the current is greater than zero. If the current becomes less than or equal to zero, `d` is recursively called with the negation of `open`, thus returning the switch on line 11. This switch acts as a current source outputting zero current, i.e. a perfect insulator (line 12), as long as the voltage is less than the `bias` voltage.

```
1  def OneWayThroughStop(bias: Real, open: Bool, t: Through, a: Across,
2    p: Node, n: Node) = {
3
4    def d(open: Bool) -> HModel = {
5      if open then
6        switch
7          (AcrossGenerator bias t a p n)
8          (t)
9          (fun thnk: () -> d (!open))
10     else
11       switch
12         (ThroughGenerator 0. t a p n)
13         (bias - a)
14         (fun thnk: () -> d (!open))
15   };
16   d open
17 }
18
19 def Diode = OneWayThroughStop
```

Listing 4.3: DILL *model of a* onewaythroughstop *(a Diode in the electrical domain).*

Not to be mistaken by a switch, the `Switch` model defined in Listing 4.4 models an electrical switch in the electrical domain or a clutch in the domain of rotational mechanics. For simplicity, we will present it as an electrical switch. The `AAcrossGenerator` on line 6 is an *anonymous* `AcrossGenerator` with *across* and *through* variables hidden from the outside. If `open` is *false*, then the switch on line 6 behaves as an voltage source outputting zero voltage, i.e. a perfect conductor. This is true as long as the signal `s` on line 8 is less than one. Otherwise the inner function, on line 2, is recursively called in the same manner as for the

onewaythroughstop discussed earlier.

Given `open` evaluated to *true*, the inner function returns a switch (line 4), modeling the electrical switch as `eande nil` which is equivalent to no model at all. Specifically, removing the `Branch` between `n` and `p`. Thus closing and opening the electrical switch results in a change in the models topology. The electrical switch remains open as long as the signal `s` is greater than zero. It is possible to model the electrical switch by replacing the first field in the switch on line 4 with a perfect insulator. However, we choose the definition in Listing 4.4 to evaluate a changing topology during simulation.

```
1  def Switch(open: Bool, s: Signal, p: Node, n: Node) = {
2    def sw(open: Bool) -> HModel = {
3      if open then
4        switch (eande nil) (s) (fun thk: () -> sw (!open))
5      else
6        switch
7          (AAcrossGenerator 0. p n)
8          (1. - s)
9          (fun thk: () -> sw (!open))
10   };
11   sw open
12 }
13
14 def ElectricalSwitch = Switch
15 def Clutch = Switch
```

Listing 4.4: *Switch model in* **DILL**.

4.1 LCD-Circuit

We start our set of example systems with the circuit discussed by Lee [25] and depicted in Figure 4.1. This circuit contains a diode, a resistor, and a capacitor. Given an initial positive current i, negative voltage u_C, and positive voltage u_L, the diode will initially be *forward biased* and the model will behave as an LC-circuit. In a LC-circuit the current oscillates around zero as the capacitor and inductor alternately stores and releases energy into the circuit.

In the LCD-Circuit, however, as the current reaches zero, the energy stored in the capacitor reaches its maximum. At the same time, the diode will change from *forward biased* to *reverse biased*, preventing any

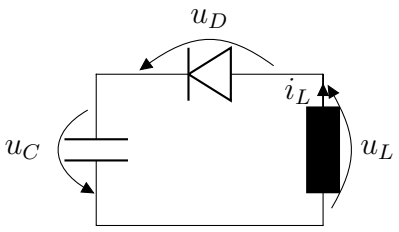


Figure 4.1: *LCD-circuit*

current from flowing in the circuit. This will lock the charge built up in the capacitor, which will apply a constant positive voltage across the diode and inductor. This ensures that the diode remains *reverse biased*. As the current instantly stops flowing when it reaches zero, its rate of change will also instantly become zero, resulting in an instant drop in the voltage over the inductor. This simple model provides a basic evaluation of the mode switch semantics, as well as evaluation of the initial values returned from the impulse analysis.

In Listing 4.5 we define a model of the LCD-circuit. This model consists of two modes defined by the `Diode` component. On line 6 we define nodes in the model topology. Note that we only supply initial values for a subset of the state variable at line 7 to 9 and let the initialization procedure search for consistent initial values for all state variables.

```
1  def L = 1.
2  def C = 1.
3  def bias = 0.
4
5  def LCD = {
6    def n1, n2, n3: Node;
7    init i_L 1. ++
8    init u_C (-1.) ++
9    init i_D 1. ++
10   Inductor L i_L u_L n1 n2 ++
11   Diode bias true i_D u_D n2 n3 ++
12   Capacitor C i_C u_C n3 n1
13 }
```

Listing 4.5: DILL *model of LCD-circuit depicted in Figure 4.1.*

We will analyze the expected behavior during the impulse analysis. When the current reaches zero, a mode switch should occur. As a result, the constitutive equation of the diode should change from $u_D = 0$ to $i_D = 0$.

$$\begin{aligned} u_{L,h} &= L\frac{i_{L,h} - i_L^-}{h} \\ i_{C,h} &= C\frac{u_{C,h} - u_C^-}{h} \\ i_{D,h} &= 0 \end{aligned} \tag{4.1}$$

$$\begin{aligned} i_{D,h} &= i_{L,h} = i_{C,h} \\ u_{D,h} &= -u_{C,h} - u_{L,h} \end{aligned} \tag{4.2}$$

In (4.1) and (4.2), we state the expected system of equations of the forward step in the impulse solving, where $h > 0$ is the step-size. In (4.1) we list the constitutive equations and in (4.2) the topological equations. Solving (4.1) together with (4.2) gives us the solutions:

$$\begin{aligned} i_{D,h} = i_{L,h} = i_{C,h} &= 0 \quad & u_{C,h} &= u_C^- \\ u_{L,h} &= -\frac{L}{h}i_L^- & u_{D,h} &= \frac{L}{h}i_L^- - u_D^- \end{aligned} \tag{4.3}$$

Because i_L^- should be close to zero (ideally identical to zero), we should not have any impulse on $u_{D,h}$, and the model should be stable. Note, however, in practice and for numerical reasons that i_L^- will have a small positive or negative value, depending on whether the simulation approaches zero from above or below. Problems can arise if $h \ll i_L^-$, which might result in a false impulses on u_D. Next we take a backwards step $-h$.

$$\begin{aligned} u_L^+ &= L\frac{i_L^+ - i_{L,h}}{-h} \\ i_C^+ &= C\frac{u_C^+ - u_{L,h}}{-h} \\ i_D^+ &= 0 \end{aligned} \tag{4.4}$$

$$\begin{aligned} i_D^+ &= i_L^+ = i_C^+ \\ u_D^+ &= -u_C^+ - u_L^+ \end{aligned} \tag{4.5}$$

The solution to (4.4) together with (4.5) are:

$$\begin{aligned} i_D^+ = i_L^+ = i_C^+ &= 0 \quad & u_C^+ &= u_C^- \\ u_L^+ &= 0 & u_D^+ &= u_C^- \end{aligned} \tag{4.6}$$

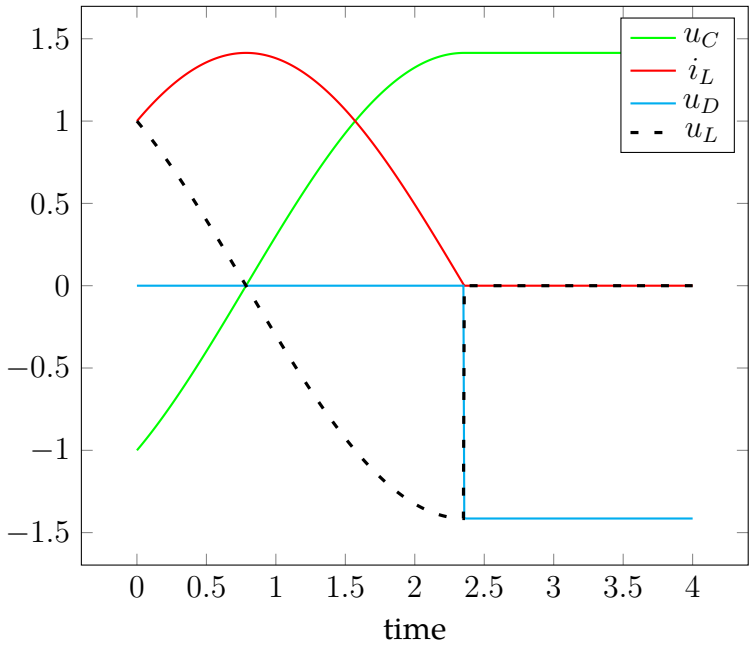


Figure 4.2: *Simulation trace of LCD-circuit model in Listing 4.5.*

which is within the valid region of the diodes switch and simulation should proceed in continuous time. We plot the simulation trace for relevant dependent variables in Figure 4.2. We can see that the simulation trace indeed behaves as expected.

4.2 Switched RLD-Circuit

Our next system to evaluate is the switched RLD-circuit (Figure 4.3), which we have discussed extensively throughout this thesis (in particular, see Section 1.4.2 and Section 3.3). This model is interesting to evaluate as it contains both a change in topology and a mode switch, resulting from an impulse during the same model time instant. Opening the electric switch at time 3 should result in a voltage impulse over the diode. This should change the diode from *reverse biased* to *forward biased*, in turn creating a circuit consisting of the inductor and the diode. Moreover, the diode should apply its constant *forward bias* voltage over the inductor, which should lead to a linear decrease in current as the inductor releases its energy into this newly formed circuit. Physically, as the inductor is an energy storing component, we expect the current over the inductor to be continuous over time.

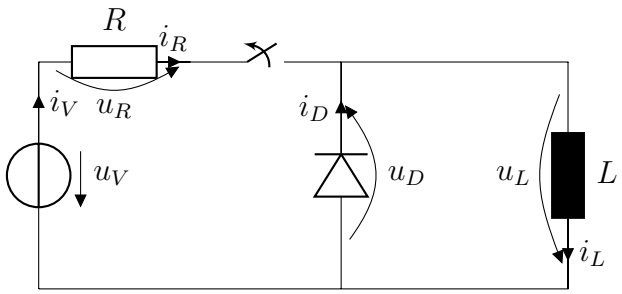


Figure 4.3: *Switched RLD-Circuit with inner LD-Circuit.*

We define our model of this system in Listing 4.6. The `UnitStep` component on line 12 makes the signal `s` remain zero until the model time is equal to `t_s`, at which time `s` is instantly changed to one. The signal `s` in turn opens the `ElectricalSwitch` on line 15.

```
1  def L = 1.
2  def R = 1.
3  def V = 1.
4  def bias = 0.7
5  def t_0 = 0.
6  def t_s = 3.
7
8  def LRD = {
9    def n1, n2, n3, n4: Node;
10   init i_L 0. ++
11   init u_D (-1.) ++
12   UnitStep t_0 t_s s ++
13   VoltageSource V i_V u_V n1 n4 ++
14   Resistor R i_R u_R n1 n2 ++
15   ElectricalSwitch false s n2 n3 ++
16   Inductor L i_L u_L n3 n4 ++
17   Diode bias false i_D u_D n4 n3
18 }
```

Listing 4.6: **DILL** *model RLD-circuit with inner LD-circuit depicted in Figure 4.3.*

As we can see from the simulation trace in Figure 4.4, we indeed get a step on the voltage over the diode and inductor at model time 3 when the electrical switch opens. Thereafter, as expected, the current through the inductor decreases linearly to zero.

Finally, as we expect, the diode changes into *reverse bias* between time 4 and 5, resulting in all currents and voltages being identically

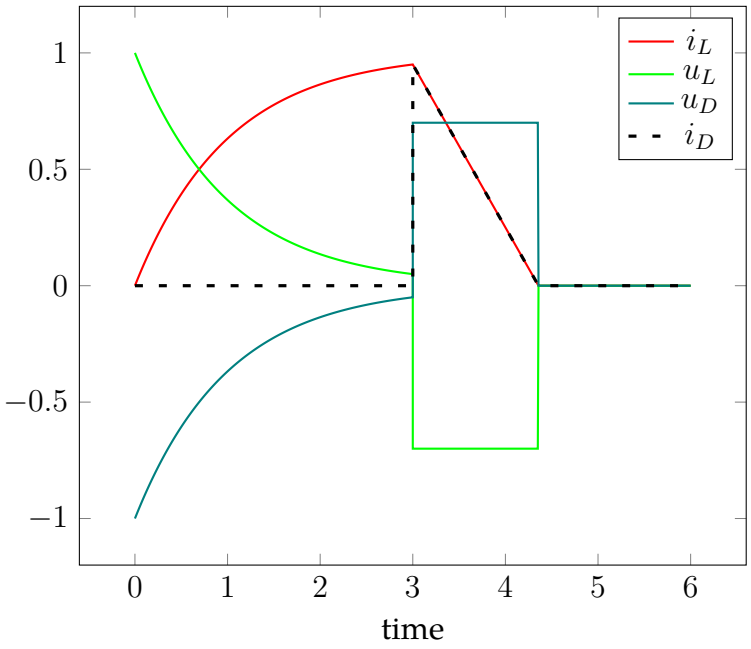


Figure 4.4: *Simulation trace of switched RLD-circuit model with inner LD-circuit in Listing 4.6.*

zero.

As a variant to the switched RLD-Circuit in Figure 4.3, we also evaluate the RLD-Circuit depicted in Figure 4.5. Here the resistor is part of the loop formed after the diode flips from *reverse bias* to *forward bias*. We define a model of this circuit in Listing 4.7. Note here that the nature of EOO language allows us to define this new model simply by changing the way we connect the components. We expect this model to behave similarly to the model in Figure 4.6, but instead of a linear decrease in the current after time 3, we should see a exponential decay. This

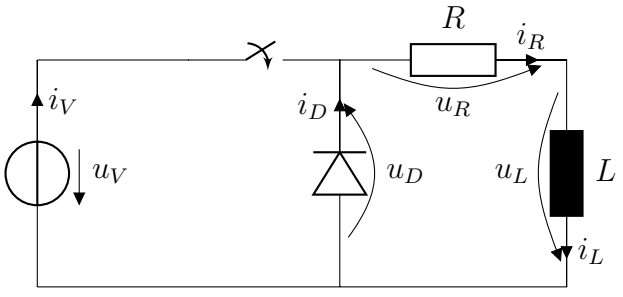


Figure 4.5: *Switched RLD-circuit with inner RLD-circuit.*

due to the voltage over the inductor decreasing rather than remaining constant. We can verify this behavior from the plot in Figure 4.6.

```
1  def L = 1.
2  def R = 1.
3  def V = 1.
4  def bias = 0.7
5  def t_0 = 0.
6  def t_s = 3.
7
8  def LRD = {
9    def n1, n2, n3, n4: Node;
10   init i_L 0. ++
11   init u_D (-1.) ++
12   UnitStep t_0 t_s s ++
13   VoltageSource V i_V u_V n1 n4 ++
14   ElectricalSwitch false s n1 n2 ++
15   Resistor R i_R u_R n2 n3 ++
16   Inductor L i_L u_L n3 n4 ++
17   Diode bias false i_D u_D n4 n2
18 }
```

Listing 4.7: **DILL** *model of switched RLD-circuit with inner RLD-circuit depicted in Figure 4.5.*

4.3 Two Bodies Connected By a Clutch

Our second physical domain is the domain of rotational mechanics. We will evaluate a system in this domain that consist of two rotating bodies joined together by a clutch. A hybrid model of this system was first discussed by Benveniste et al. [6], although not modeled as a EOO model. The bodies have some mass and are also damped. Here we consider the model where the clutch is initially open and the two bodies rotate with different initial values on their angular velocities. At some time instant, the clutch will close and the two bodies will instantly have the same angular velocity. After some additional time the clutch once again opens. We depict this system in Figure 4.7. To show why this system is interesting to evaluate we will analytically analyze its behavior.

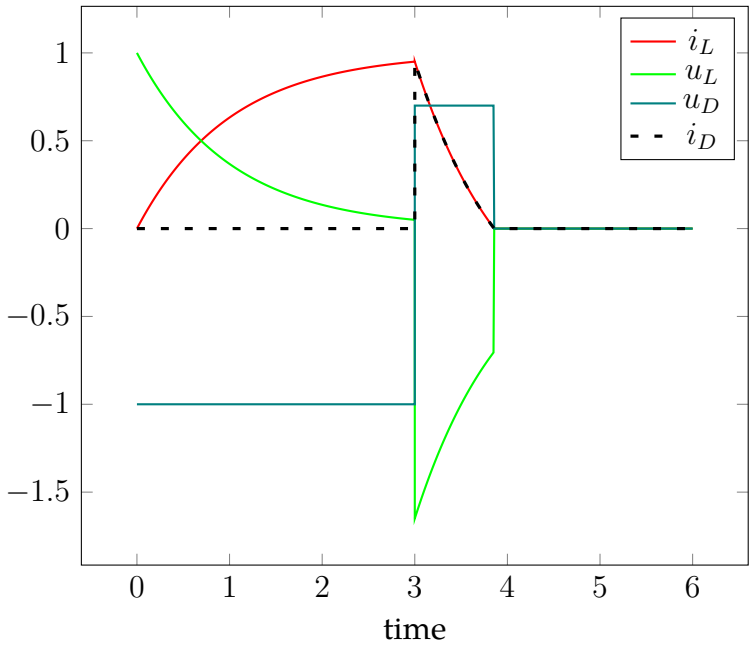


Figure 4.6: *Simulation trace of RLD-circuit model with inner RLD-circuit in Listing 4.7.*

$$\begin{aligned} D_1 \cdot \omega_{D_1} &= \tau_{D_1} \quad I_1 \cdot \frac{\mathrm{d}\omega_{I_1}}{\mathrm{d}t} = \tau_{I_1} \\ D_2 \cdot \omega_{D_2} &= \tau_{D_2} \quad I_2 \cdot \frac{\mathrm{d}\omega_{I_2}}{\mathrm{d}t} = \tau_{I_2} \end{aligned} \tag{4.7}$$

$$\begin{aligned} \omega_{D_1} &= \omega_{I_1} \quad \tau_{D_1} = -\tau_{I_1} \\ \omega_{D_2} &= \omega_{I_2} \quad \tau_{D_2} = -\tau_{I_2} \end{aligned} \tag{4.8}$$

The constitutive equations in (4.7) and the topological equations in (4.8) makes up the DAE of the mode before the clutch closes. We can note, as expected, that the system of equations governing each damped body are independent of each other. Solving (4.7) together with (4.8) gives the solutions:

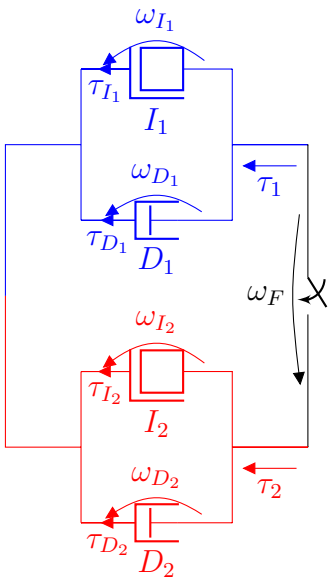


Figure 4.7: *Two rotating damped bodies connected trough a clutch. Components with subscript* 1 *represents the first damped body, and components with subscript* 2 *represents the second damped body.*

$$\begin{aligned}
\omega_{D_1}(t) &= \omega_{I_1}(t) = \omega_1(t) = \omega_1(0) \cdot e^{-t\frac{D_1}{I_1}} \\
\tau_{D_1} &= -\tau_{I_1} = D_1\omega_1(0) \cdot e^{-t\frac{D_1}{I_1}} \\
\omega_{D_2}(t) &= \omega_{I_2}(t) = \omega_2(t) = \omega_2(0) \cdot e^{-t\frac{D_2}{I_2}} \\
\tau_{D_2} &= -\tau_{I_2} = D_2\omega_2(0) \cdot e^{-t\frac{D_2}{I_2}}
\end{aligned} \tag{4.9}$$

where $\omega_{1,2}(0)$ are the initial values on $\omega_{1,2}$. We can get the external torque $\tau_{1,2}$ on each body by the sum $\tau_{D_{1,2}} + \tau_{I_{1,2}}$, which we can conclude is zero from (4.9). This is what we expect as no external forces act upon the bodies. More interesting is the scenario when we close the clutch. Assume the left-limit of the angular velocities $\omega_{1,2}^-$, right before the clutch closes fulfills $\omega_1^- \neq \omega_2^-$.

$$\begin{aligned}
D_1 \cdot \omega_{D_1} &= \tau_{D_1} \quad I_1 \cdot \frac{\mathrm{d}\omega_{I_1}}{\mathrm{d}t} = \tau_{I_1} \\
D_2 \cdot \omega_{D_2} &= \tau_{D_2} \quad I_2 \cdot \frac{\mathrm{d}\omega_{I_2}}{\mathrm{d}t} = \tau_{I_2} \\
\omega_F &= 0
\end{aligned} \tag{4.10}$$

$$\begin{aligned} \omega_{D_1} &= \omega_{I_1} & \tau_{D_1} &= -\tau_{I_1} \\ \omega_{D_2} &= \omega_{I_2} & \tau_{D_2} &= -\tau_{I_2} \\ \omega_F &= \omega_{D_1} - \omega_{D_2} \end{aligned} \tag{4.11}$$

We describe this new mode with the constitutive equations in (4.10) and the topological equations in (4.11). The equations involving ω_F in (4.10) and (4.11) results from the now closed clutch. Combining these two equations gives us the equation $\omega_{D_1} = \omega_{D_2}$, which is inconsistent with the left-limit from the predecessor mode, as we assumed $\omega_1^- \neq \omega_2^-$. Thus the change in the model topology requires a step on ω_{D_1} and/or ω_{D_2} before simulation can proceed. The question is what value to assign to the initial values of $\omega_{D_1} = \omega_{D_2}$ in the successor mode.

$$\omega_{1,2}^+ = \frac{I_1\omega_1^- + I_2\omega_2^-}{I_1 + I_2} \tag{4.12}$$

Benveniste et al. [6], derived the result in (4.12), using nonstandard analysis and algebraic manipulation. This is the weighted mean of the left-limits of the angular velocities and the moments of inertia.

In Listing 4.8 we define a model of the two damped bodies and a clutch in **DILL**. On line 10 we define a model for the Bodies. We only need to know the angular velocity of either the `Mass` (line 20) or the `Damper` for each rotating body because they are parallel. However, we need both the torque from the `Mass` and the `Damper` (line 18) to get the resulting torque on the body as the sum of these torques. We also include the angular momentum on line 19, to examine the behavior of `t_1` and `t_2` when the clutch closes. The `TwoUnitSteps` model on line 28 controls the closing and opening of the `Clutch` model on line 31.

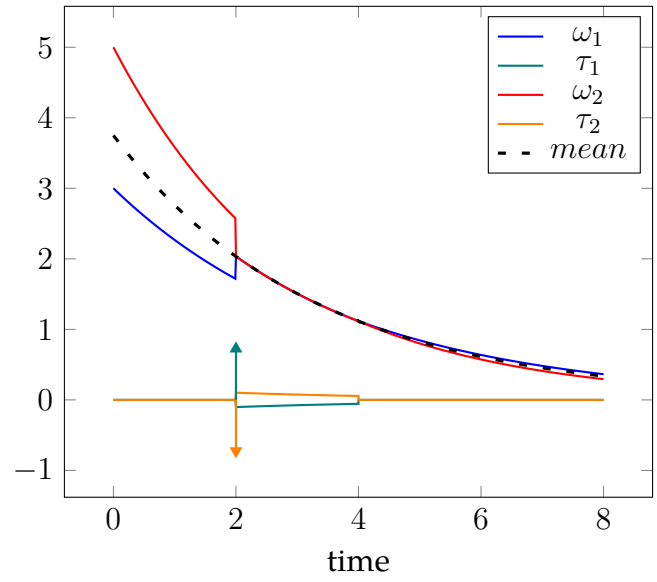
We depict the simulation trace of the model defined in Listing 4.8 in figure 4.8a, where $mean$ is:

$$\frac{I_1\omega_1 + I_2\omega_2}{I_1 + I_2}$$

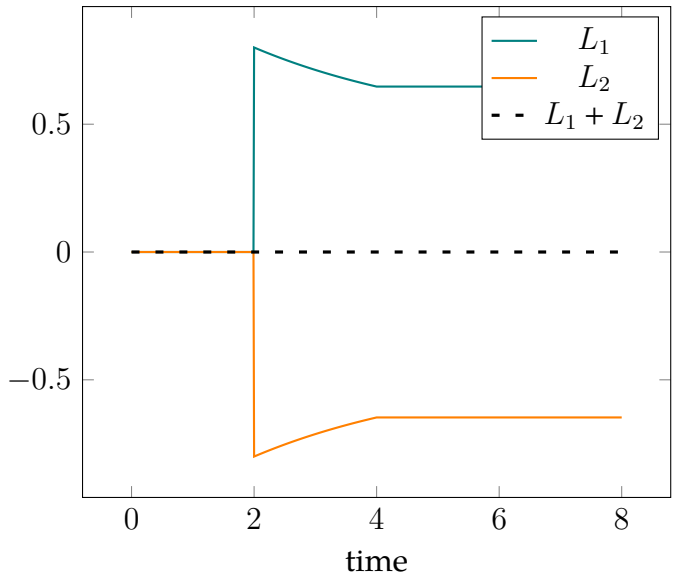
```
def o_1_0 = 3.
def o_2_0 = 5.
def I_1 = 2.5
def D_1 = 0.7
def I_2 = 1.5
def D_2 = 0.5
def t_0 = 0.
def t_s = 2.

def Body(I: Real, D: Real, o_0: Real,
  t: Torque, o: AngularVelocity, L: AngularMomentum,
  p: Node, n: Node) = {

  def t_m, t_d: Torque;
  def o_m, o_d: AngularVelocity;
  init o_m o_0 ++
  mode(
    t = t_m + t_d;
    der L = t;
    o = o_m
  ) ++
  Mass I t_m o_m p n ++
  Damper D t_d o_d p n
}

def ClutchModel = {
  def n1, n2, n3: Node;
  TwoUnitSteps t_0 t_s (2. * t_s) s ++
  Body I_1 D_1 o_1_0 t_1 o_1 L_1 n1 n2 ++
  Body I_2 D_2 o_2_0 t_2 o_2 L_2 n1 n3 ++
  Clutch true (1. - s) n2 n3
}
```

Listing 4.8: **DILL** *model of the two damped bodies connected by a clutch depicted in Figure 4.7.*

We can verify from the simulation trace that the initial values of the predecessor mode after the clutch closes initializes as given by (4.12). It is also possible to verify this behavior analytically by solving the impulse analysis procedure explicitly. We have also manually added impulse arrows for the torques, which we derived from the simulation trace of the angular momentum shown in Figure 4.8b. This trace also ensures us that energy is conserved in the system.



(a) *Simulation trace of angular velocities and torques.*



(b) *Simulation trace of angular momentum.*

Figure 4.8: *Simulation trace of clutch model in Listing 4.8.*

4.4 Bouncing Ball

Next we consider a simple one-dimensional mechanical problem, sometimes used as a "Hello World" problem in the field of *Hybrid Systems* modeling. This system consists of a body (we assume its mass is unity)

dropped from some height y, say y_0, and that is subject to the downwards gravitational force g. The body will begin to fall, accelerating because of the gravitational force. When the body reaches $y = 0$, we assume an *inelastic* collision with the floor, where the body loses a portion of its kinetic energy. This will result in the body bouncing back up, but at a lower magnitude of velocity, hence we call this model a *bouncing ball*. In Figure 4.9, the left-hand side depicts the body some time before it reaches the floor and the right-hand side depicts the body just after the collision has occurred.

In Listing 4.9 we define a first attempt to model this system. The constant `e`, called the *coefficient of restitution*, governs the amount of kinetic energy lost at collision. The model `fall` (line 4) models the behavior of the body when it falls and `floor` (line 12) models its behavior as it hits the floor.

The final model `bouncing_ball` on line 20 represents the complete system. Initially `bouncing_ball` behaves as `fall`, as defined on line 22. When `y` becomes less or equal to zero, the switch on line 21 switches to the switch on line 24 and `bouncing_ball` now behaves as `floor`, as given on line 25. This is true as long as the acceleration `a` is greater than zero. Because the velocity `v` will change sign, we expect an impulse on `a`. When this impulse has died out we want to immediately recursively call `bouncing_ball` once again to make it behave as `fall`, as given on line 22. Hence `floor` should never appear in continuous time and only ensure that the velocity gets set to its proper value. This an example of where we want to introduce a transient state that changes the physical system. However, simulating `bouncing_ball` fails.

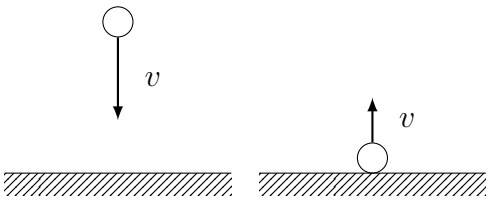


Figure 4.9: *Bouncing ball*

```
1  def g = 9.81
2  def e = 0.7
3
4  def fall(y: <Real>, v: <Real>, a: <Real>) = {
5    eande (
6      der y = v;
7      der v = a;
8      a = -g
9    )
10 }
11
12 def floor(y: <Real>, v: <Real>, a: <Real>) = {
13   eande (
14     v = -e * pre v;
15     der y = v;
16     der v = a
17   )
18 }
19
20 def bouncing_ball(y: <Real>, v: <Real>, a: <Real>) -> HModel = {
21   switch
22     (fall y v a)
23     (y)
24     (fun t:() -> switch
25                    (floor y v a)
26                    (a)
27                    (fun t: () -> bouncing_ball y v a))
28 }
```

Listing 4.9: *First attempt to model the bouncing ball system depicted in Figure 4.9 in* **DILL**.

$$\begin{aligned} v_h &= -ev^- \\ a_h &= \frac{v_h - v^-}{h} \\ v_h &= \frac{y_h - y^-}{h} \end{aligned} \tag{4.13}$$

$$\begin{aligned} v^+ &= -ev^- \\ a^+ &= \frac{v^+ - v_h}{-h} \\ v^+ &= \frac{y^+ - y_h}{-h} \end{aligned} \tag{4.14}$$

To see why, we look a bit closer on the impulse analysis for this model. In Equation (4.13) we state the forward step of the impulse solving, as we come from the free-falling mode to the floor mode, which has the following solution:

$$v_h = -ev^- \quad y_h = y^- - hev^- \quad a_h = \frac{-(1+e)v^-}{h}$$

We see that we have a positive impulse on a as $a_h \to \infty$ if $h \to 0$. Solving (4.14) gives us the solutions:

$$v^+ = -ev^- \quad y^+ = y^- \quad a^+ = 0$$

which are the values we expect on v and y. As $a^+ = 0$, the switch on 24 flips and we set these values as left-limits during the next impulse analysis going from the floor mode to the free-falling mode. Let y^{++}, v^{++}, and a^{++} denote the solution of y, v, and a during the forward step of the impulse solving.

$$\begin{aligned} a'_h &= -g \\ a'_h &= \frac{v'_h - v^+}{h} \\ v'_h &= \frac{y'_h - y^+}{h} \end{aligned} \tag{4.15}$$

$$\begin{aligned} a^{++} &= -g \\ a^{++} &= \frac{v^{++} - v'_h}{-h} \\ v^{++} &= \frac{y^{++} - y'_h}{-h} \end{aligned} \tag{4.16}$$

(4.15) and (4.16) show the forward and backward step during the impulse solving. Together they give the solution to y^{++}, v^{++}, and a^{++} as:

$$v^{++} = v^+ = -ev^- \quad y^{++} = y^+ - h^2 g = y^- - h^2 g \quad a^{++} = -g$$

Thus, we can see that the backwards Euler approximation introduces an error on y^{++}, proportional to h^2 that in this case makes the simulation enter a *Zeno* state of non-termination. This analysis also shows the importance of the left-limit being within the valid region, when coming from the continuous simulation. If y^- were not inside the valid region, the same problem would occur even if the error term $-h^2 g$ were not present.

Moreover, if we would not allow this transient state to change the left-limit as defined on line 12 in Listing 2.40, $v+$ in Equation (4.15) would be v^- rather than $-ev^-$, which in essence would make the collision disappear.

To side-step the problem of the impulse analysis we can reformulate the model in Listing 4.9 to the model defined in Listing 4.10. Here, we have introduced a new model construct `reinit` on line 5 to 7, which explicitly defines the initial values in the successor mode. In this model, the `reinit`'s are parameterized and assigned values on line 20. We are able to successfully simulate the bouncing ball model as shown in Figure 4.10.

```
1  def g = 9.81
2  def e = 0.7
3
4  def fall(y: <Real>, v: <Real>, y_0: <Real>, v_0: <Real>) = {
5    reinit y y_0 ++
6    reinit v v_0 ++
7    reinit (der y) v_0 ++
8    eande (
9      der y = v;
10     der v = -g
11   )
12 }
13
14 def bouncing_ball(y: <Real>, v: <Real>, y_0: <Real>, v_0: <Real>)
15   -> HModel = {
16
17   switch
18     (fall y v y_0 v_0)
19     (y)
20     (fun t: () -> bouncing_ball y v (pre y) (-e * pre v))
21 }
```

Listing 4.10: *Second attempt to model the bouncing ball system depicted in Figure 4.9 in* **DILL**.

A problem with this approach, however, is that it hides any impulses occurring as a result of jumps on the state variables. Moreover, in more complex models it might be difficult to determine consistent initial values for the successor mode. Another approach to reduce the sensitivity to numerical errors would be to disregard the notion of valid regions and instead considering directed zero-crossings as conditions for switches to flip. This would allow us to simulate the `bouncing_ball` model defined in Listing 4.9, because we could define the outermost switch to trigger when `y` crosses $y = 0$ from above. Then the error from the impulse solving would only put the ball slightly further below $y = 0$, which would not trigger an additional mode switch.

Moreover, the switch feels inconvenient when we want to include impulse events, as the choice of valid region on the switch on line 26 in Listing 4.9 is rather forced. A better approach would be something similar to the approach taken by Zimmer [45], where we include a construct that defines a mode active for just one micro-step.

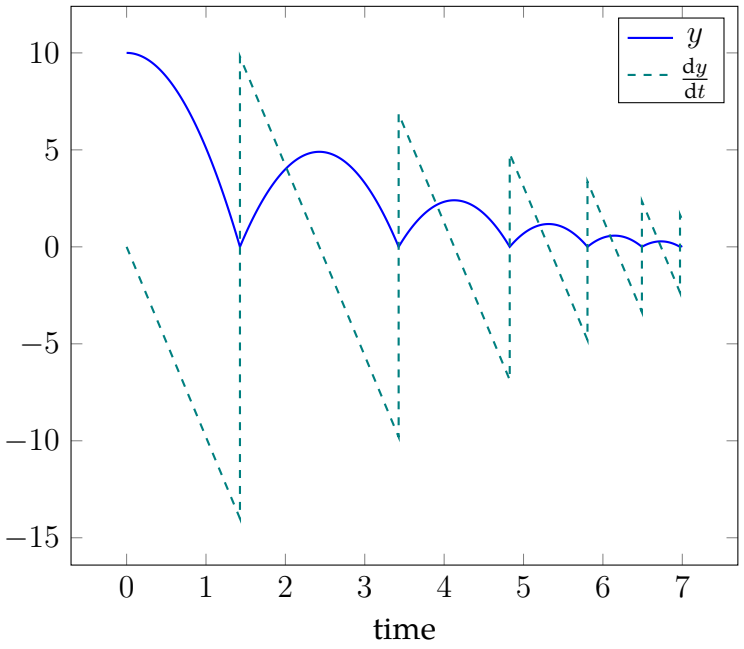


Figure 4.10: *Simulation trace of the second bouncing ball model in Listing 4.10.*

4.5 Summary

We end this chapter with a summary of the evaluated hybrid systems we have discussed.

LCD-Circuit We model and simulate a circuit containing a diode, capacitor and inductor (Figure 4.1). The diode is initially *forward biased* and as the current through the diode reaches zero the diode becomes *reverse biased* resulting in a step on the voltage over the inductor. We define a model for this system in Listing 4.5 which results in a simulation trace (Figure 4.2) that behaves as expected.

Switched RLD-Circuit We model and simulate two circuits containing a voltage source, a resistor and an inductor, in addition to an electrical switch and a diode placed in parallel with the inductor (Figure 4.3 and 4.5). The electrical switch is initially closed and assumed to open at some point in time. The diode is initially *reverse biased*. The instant voltage drop resulting from opening the switch will induce a voltage over the inductor, which in turn

will make the diode *forward biased*, allowing current to flow in the diode-inductor loop and preventing the occurrence of a voltage impulse over the inductor. We define models for these two systems in Listings 4.6 and 4.7. The simulation trace for these two models, shown in Figure 4.4 and 4.6, behaves as expected.

Two Bodies Connected by a Clutch We model and simulate a mechanical system consisting of two rotating damped bodies, shown in Figure 4.7. We assume that the clutch is initially open and closes at some point in time. We also assume that the two bodies have non-equal angular velocities right before the clutch closes. From previous results we expect an impulse on the external torques on the bodies and that the angular velocity of the bodies changes to the weighted mean given by Equation (4.12), as the clutch closes. The simulation traces shown in Figure 4.8 behaves as expected.

Bouncing Ball This mechanical system consists of a ball, initially dropped at some height above the ground and subject to *inelastic collision* when it reaches the ground, resulting in the ball bouncing back up in the air. A first model of this system, including impulse analysis, is defined in Listing 4.9. Simulation of this model fails due to approximation errors in the impulse solving implementation. For comparison, we define a model where the impulse analysis is ignored and consistent initial values are specified explicitly, in Listing 4.10. Simulation of this model results in a correct simulation trace as shown in Figure 4.10.

Chapter 5

Conclusion

5.1 Future Work

We propose the following research topics in no particular order, for future research on dynamic EOO hybrid modeling language.

- Investigating the replacement of valid regions, with mode switch conditions based on directed zero-crossings, to reduce sensitivity to numerical errors.
- Extending λ_{DILL}, with a construct for explicitly defining transient modes changing state variables, and further investigations on a sound theory of such behavior in hybrid models.
- Extending the impulse solving, discussed in Section 3.3, to problems of higher index.
- Extending λ_{DILL} with error handling, and analyzing the correctness of λ_{DILL}.
- Evaluation of the expressiveness of λ_{DILL} with regards to systems exhibiting impulsive friction.

5.2 Conclusion

This thesis has discussed hybrid semantics in equation-based modeling languages. In particular we have discussed hybrid EOO languages, with dynamic elaboration and dynamic equation transformation. We

have formalized a hybrid semantics and implemented it as part of a hybrid EOO language. This formalization includes the adaption of impulse analysis (Section 1.4.3), from the theory of non-linear circuits extended to allow models of *inelastic collision*. We evaluated the expressiveness of the implementation on a set of structurally varying systems in the electrical and mechanical domain. From the evaluation, we conclude that λ_{DILL} is able to express several circuits, or circuit like systems, where impulses and discontinuities are integral parts of the systems behavior. However, the semantics suffers from sensitivity to numerical errors and the implementation is not able to express *inelastic collision* in a satisfactory manner. The current implementation of impulse analysis cannot handle higher index problems.

Bibliography

[1] T .Quarles et al. *SPICE3 User's Manual*. English. Version 3f3. University of California. May 1993. 146 pp.

[2] R. Alur et al. "Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems". In: *Hybrid Systems*. Ed. by Robert L. Grossman et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 209–229.

[3] Gordon C. Andrews. "A General Re-Statement of the Laws of Dynamics Based on Graph Theory". In: *Problem Analysis in Science and Engineering*. Ed. by F.H. Branin and K. Huseyin. Academic Press, 1977, pp. 1–40.

[4] B. Bachmann, P. Aronsson, and P. Fritzson. "Robust Initialization of Differential Algebraic Equations". In: *Proceedings of the 1st International Workshop on Equation-Based Object-Oriented Languages and Tools*. 24. Linköping University Electronic Press; Linköpings universitet, 2007, pp. 151–163.

[5] A. Benveniste et al. *On the index of multi-mode DAE Systems (also called Hybrid DAE Systems)*. Research Report RR-8630. Inria, Nov. 2014, p. 30.

[6] A. Benveniste et al. "Structural Analysis of Multi-Mode DAE Systems". In: *Proceedings of the 20th International Conference on Hybrid Systems: Computation and Control*. HSCC '17. Pittsburgh, Pennsylvania, USA: ACM, 2017, pp. 253–263.

[7] A. Benveniste et al. "The synchronous languages 12 years later". In: *Proceedings of the IEEE* 91.1 (Jan. 2003), pp. 64–83.

[8] D. Broman. "Meta-Languages and Semantics for Equation-Based Modeling and Simulation". PhD thesis. Linköping University, The Institute of Technology, 2010, p. 263.

[9] D. Broman and H. Nilsson. "Node-Based Connection Semantics for Equation-Based Object-Oriented Modeling Languages". In: *Practical Aspects of Declarative Languages: 14th International Symposium, PADL 2012, Philadelphia, PA, USA, January 23-24, 2012. Proceedings*. Ed. by Claudio Russo and Neng-Fa Zhou. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 258–272.

[10] D. Broman and J. G. Siek. "Gradually Typed Symbolic Expressions". In: *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. PEPM '18. Los Angeles, CA, USA: ACM, 2018, pp. 15–29. ISBN: 978-1-4503-5587-2.

[11] D. Broman and J. G. Siek. *Modelyze: a Gradually Typed Host Language for Embedding Equation-Based Modeling Languages*. Tech. rep. UCB/EECS-2012-173. EECS Department, University of California, Berkeley, Apr. 2012.

[12] P. N. Brown, George D. B., and A. C. Hindmarsh. "VODE: A Variable-Coefficient ODE Solver". In: *SIAM Journal on Scientific and Statistical Computing* 10.5 (1989), pp. 1038–1051.

[13] R. Bruni and U. Montanari. *Models of Computation*. Cham, Switzerland: Springer International Publishing, 2017.

[14] S. L. Campbell and C. W Gear. "The index of general nonlinear DAEs". In: *Numerische Mathematik* 72.2 (Dec. 1995), pp. 173–196. ISSN: 0945-3245.

[15] W. K. Chen. *Graph Theory and Its Engineering Applications*. Singapore: World Scientific Publishing Company, 1997, pp. 1–77.

[16] Schaft A. J. van der and J. M. Schumacher. *Introduction to Hybrid Dynamical Systems*. London, UK, UK: Springer-Verlag, 1999. ISBN: 1852332336.

[17] H. Elmqvist. "A Structured Model Language for Large Continuous Systems". eng. PhD thesis. 1978.

[18] H. Elmqvist, S. E. Mattsson, and M. Otter. "Modelica extensions for Multi-Mode DAE Systems". In: *10th International Modelica Conference*. Ed. by Hubertus Tummescheit and Karl-Erik Arzen. Linköping Electronic Conference Proceedings. LiU Electronic Press, 2014, pp. 183–193.

[19] G. Fábián, D.A. Van Beek, and J.E. Rooda. "Index Reduction and Discontinuity Handling Using Substitute Equations". In: *Mathematical and Computer Modelling of Dynamical Systems* 7.2 (2001), pp. 173–187.

[20] P. Fritzson. *Introduction to modeling and simulation of technical and physical systems with Modelica*. Hoboken, N.J: Wiley IEEE Press, 2011.

[21] S. Furic. "Enforcing model composability in Modelica". In: *Proceedings of the 7th International Modelica Conference; Como; Italy; 20-22 September 2009*. 43. Linköping University Electronic Press; Linköpings universitet, 2009, pp. 868–879.

[22] C. W. Gear and L. R. Petzold. "ODE Methods for the Solution of Differential/Algebraic Systems". In: *SIAM Journal on Numerical Analysis* 21.4 (1984), pp. 716–728.

[23] G. Giorgidze and H. Nilsson. "Higher-Order Non-Causal Modelling and Simulation of Structurally Dynamic Systems". In: *Proceedings of the 7th Modelica Conference*. Como, Sept. 2009, pp. 208–218.

[24] J.E. Dennis Jr. "Some computational techniques for the nonlinear least squares problem". In: *Numerical Solution of Systems of Nonlinear Algebraic Equations*. Ed. by G. D. Byrne and C. A. Hall. Academic Press, 1973, pp. 157–183.

[25] E. A. Lee. "Constructive Models of Discrete and Continuous Physical Phenomena". In: *IEEE Access* 2 (2014), pp. 797–821.

[26] T. Lindstrøm and N. Cutland. "An Invitation To Nonstandard Analysis". In: *Nonstandard Analysis and its Applications*. London Mathematical Society Student Texts. Cambridge University Press, 1988, pp. 1–105.

[27] S. E. Mattsson, H. Elmqvist, and M. Otter. "Physical system modeling with Modelica". In: *Control Engineering Practice* 6.4 (1998), pp. 501–510. ISSN: 0967-0661.

[28] S. E. Mattsson, M. Otter, and H. Elmquvist. "Multi-Mode DAE Systems with Varying Index". In: *11th International Modelica Conference*. 2015, pp. 89–98.

[29] S. E. Mattsson and G. Söderlind. "Index reduction in differential-algebraic equations using dummy derivatives". In: *SIAM J. Sci. Comput.* 14.3 (1993), pp. 677–692. ISSN: 1064-8275.

[30] J. J. McPhee. "On the use of linear graph theory in multibody system dynamics". In: *Nonlinear Dynamics* 9.1 (Feb. 1996), pp. 73–90.

[31] P. J. Mosterman and G. Biswas. "Modeling discontinuous behavior with hybrid bond graphs". In: *Proceedings of the 9th Qualitative Reasoning Workshop*. Amsterdam, The Netherlands, May 1995, pp. 139–147.

[32] H. Nilsson and G Giorgidze. "Exploiting structural dynamism in Functional Hybrid Modelling for simulation of ideal diodes". In: *Czech Technical University Publishing House*. 2010.

[33] H. Nilsson, J. Peterson, and P. Hudak. "Functional Hybrid Modeling". In: *Practical Aspects of Declarative Languages*. Ed. by Veronica Dahl and Philip Wadler. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 376–390.

[34] C. C. Pantelides. "The Consistent Initialization of Differential-Algebraic Systems". In: *SIAM Journal on Scientific and Statistical Computing* 9.2 (1988), pp. 213–231.

[35] S. L. Peyton Jones. *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1987.

[36] J.D. Pryce. "Solving high-index DAEs by Taylor series". In: *Numerical Algorithms* 19.1 (Dec. 1998), pp. 195–211. ISSN: 1572-9265.

[37] L. Sass et al. "A Comparison of Different Methods for Modelling Electromechanical Multibody Systems". In: *Multibody System Dynamics* 12.3 (Oct. 2004), pp. 209–250.

[38] R. R. H. Schiffelers et al. "Formal Semantics of Hybrid Chi". In: *Formal Modeling and Analysis of Timed Systems*. Ed. by Kim Guldstrand Larsen and Peter Niebert. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 151–165.

[39] P. Schwarz. "Simulation of Systems with Dynamically Varying Model Structure". In: *Math. Comput. Simul.* 79.4 (Dec. 2008), pp. 850–863.

[40] A. Steinbrecher. "Regularization and Numerical Integration of DAEs Based on the Signature Method". In: *Mathematical and Computational Approaches in Advancing Modern Science and Engineering*. Ed. by Jacques Bélair et al. Cham: Springer International Publishing, 2016, pp. 749–761.

[41] J. Unger, A. Kröner, and W. Marquardt. "Structural analysis of differential-algebraic equation systems—theory and applications". In: *Computers & Chemical Engineering* 19.8 (1995), pp. 867–882.

[42] J. Vlach, J. M. Wojciechowski, and A. Opal. "Analysis of nonlinear networks with inconsistent initial conditions". In: *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications* 42.4 (Apr. 1995), pp. 195–200.

[43] M. Yannakakis. "Hierarchical State Machines". In: *Theoretical Computer Science: Exploring New Frontiers of Theoretical Informatics*. Ed. by Jan van Leeuwen et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 315–330.

[44] F. Yuan and A. Opal. *Computer Methods for Analysis of Mixed-Mode Switching Circuits*. New York: Kluwer Academic, 2004, pp. 90–107.

[45] D. Zimmer. "Diss. ETH No. Equation-Based Modeling of Variable-Structure Systems". PhD thesis. Zurich, Switzerland: ETH Zurich, 2010.

Appendix A

Model Source

```
/*
Modeling Kernel Language (Modelyze) library
Copyright (C) 2010-2012 David Broman

Modelyze library is free software: you can redistribute it and/or modify
it under the terms of the GNU Lesser General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

Modelyze library is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License
along with Modelyze library.  If not, see <http://www.gnu.org/licenses/>.

written by Oscar Eriksson oerikss@kth.se

*/

include DILL

type Through = <Real>
type Across = <Real>
type Time = <Real>
type Signal = <Real>
type Current = Through
type Voltage = Across
```

```
type Torque = Through
type AngularMomentum = Across
type AngularVelocity = Across

def Clock(t_0: Real, t: Time) = {
  init t t_0 ++
  eande (der t = 1.)
}

def UnitStep(t_0: Real, t_s: Real, s: Signal) = {
  def t: Time;
  Clock t_0 t ++
  switch
    (init s 0. ++ eande (s = 0.))
    (t_s - t)
    (fun thk: () -> (init s 1. ++ eande (s = 1.)))
}

def TwoUnitSteps(t_0: Real, t_s1: Real, t_s2: Real, s: Signal) = {
  def t: Time;
  Clock t_0 t ++
  switch
    (init s (-1.) ++ eande (s = -1.))
    (t_s1 - t)
    (fun thk: () ->   switch
         (init s 1. ++ eande (s = 1.))
         (t_s2 - t)
         (fun thk: () -> (init s (-1.) ++ eande (s = -1.))))
}

def Dissipator(C: Real, t: Through, a: Across, p: Node, n: Node) = {
  eande (
    Branch t a p n;
    C * a = t
  )
}

def ADissipator(C: Real, p: Node, n:Node) = {
  def t: Through;
  def a: Across;
  Dissipator C t a p n
}
```

```
def Damper = Dissipator
def Resistor = (fun R: Real -> Dissipator (1. / R))
def ADamper = ADissipator
def AResistor = (fun R: Real -> ADissipator (1. / R))

def AcrossGenerator(C: Real, t: Through, a: Across, p: Node, n: Node) = {
  init a C ++
  eande (
    Branch t a p n;
    a = C
  )
}

def AAcrossGenerator(C: Real, p: Node, n:Node) = {
  def t_AG: Through;
  def a_AG: Across;
  AcrossGenerator C t_AG a_AG p n
}

def VoltageSource = AcrossGenerator
def Motor = AcrossGenerator
def Conductor = AcrossGenerator 0.
def FixedAxis = AcrossGenerator 0.
def AVoltageSource = AAcrossGenerator
def AMotor = AAcrossGenerator
def AConductor = AAcrossGenerator 0.
def AFixedAxis = AAcrossGenerator 0.

def ThroughSensor(t: Through, p: Node, n: Node) = {
  def a: Across;
  AcrossGenerator 0. t a p n
}

def CurrentSensor = ThroughSensor
def TorqueSensor = ThroughSensor

def ThroughGenerator(C: Real, t: Through, a: Across, p: Node, n: Node) = {
  eande (
    Branch t a p n;
    t = C
  )
}
```

```
def AThroughGenerator(C: Real, p: Node, n:Node) = {
  def t_TG: Through;
  def a_TG: Across;
  ThroughGenerator C t_TG a_TG p n
}

def CurrentSource = ThroughGenerator
def ConstantForceSpring = ThroughGenerator
def Insulator = ThroughGenerator 0.
def FreeAxis = ThroughGenerator 0.
def ACurrentSource = AThroughGenerator
def AConstantForceSpring = AThroughGenerator
def AInsulator = AThroughGenerator 0.
def AFreeAxis = AThroughGenerator 0.

def AcrossSensor(a: Across, p: Node, n: Node) = {
  def t: Through;
  ThroughGenerator 0. t a p n
}

def VoltageSensor = AcrossSensor
def AngularVelocitySensor = AcrossSensor

def AcrossStorage(C: Real, t: Through, a: Across, p: Node, n: Node) = {
  eande (
    Branch t a p n;
    C * (der a) = t
  )
}

def AAcrossStorage(C: Real, p: Node, n:Node) = {
  def t: Through;
  def a: Across;
  AcrossStorage C t a p n
}

def Mass = AcrossStorage
def Capacitor = AcrossStorage
def AMass = AAcrossStorage
def ACapacitor = AAcrossStorage

def ThroughStorage(C: Real, t: Through, a: Across, p: Node, n: Node) = {
  eande (
```

```
    Branch t a p n;
    C * (der t) = a
  )
}

def AThroughStorage(C: Real, p: Node, n:Node) = {
  def t: Through;
  def a: Across;
  ThroughStorage C t a p n
}

def Spring =  ThroughStorage
def Inductor = ThroughStorage
def ASpring =  AThroughStorage
def AInductor = AThroughStorage

def OneWayThroughStop(bias: Real, open: Bool, t: Through, a: Across,
  p: Node, n: Node) = {

  def d(open: Bool) -> HModel = {
    if open then
      switch
        (AcrossGenerator bias t a p n)
        (t)
        (fun thnk: () -> d (!open))
    else
      switch
        (ThroughGenerator 0. t a p n)
        (bias - a)
        (fun thnk: () -> d (!open))
  };
  d open
}

def AOneWayThroughStop(bias: Real, open: Bool, p: Node, n:Node) = {
  def t: Through;
  def a: Across;
  OneWayThroughStop bias open t a p n
}

def Diode = OneWayThroughStop
def ADiode = AOneWayThroughStop
```

```
def Switch(open: Bool, s: Signal, p: Node, n: Node) = {
  def sw(open: Bool) -> HModel = {
    if open then
    {
      switch (eande nil) (s) (fun thk: () -> sw (!open))
    }
    else
    {
      switch (AAcrossGenerator 0. p n) (1. - s) (fun thk: () -> sw (!open))
    }
  };
  sw open
}

def ElectricalSwitch = Switch
def Clutch = Switch
```

TRITA EECS-EX-2018:690