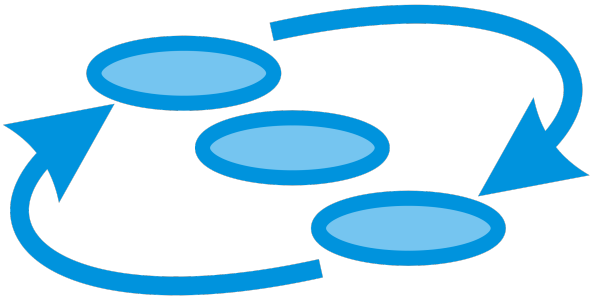
Grant Agreement: 644047

INtegrated TOol chain for model-based design of CPSs



INTO-CPS

Differential Equations in the Unifying Theories of Programming

Technical Note Number: D2.1c

Version: 0.7

Date: December 2015

Public Document

http://into-cps.au.dk

Contributors:

Simon Foster, UY
Bernhard Thiele, LIU
Jim Woodcock, UY

Editors:

Simon Foster, UY

Reviewers:

Stylianos Basagiannis, UTRC
Christian Kleijn, CLP
Ken Pierce, UNEW

Consortium:

Aarhus University	AU	Newcastle University	UNEW
University of York	UY	Linköping University	LIU
Verified Systems International GmbH	VSI	Controllab Products	CLP
ClearSy	CLE	TWT GmbH	TWT
Agro Intelligence	AI	United Technologies	UTRC
Softeam	ST		

Document History

Ver	Date	Author	Description
0.1	18-05-2015	Simon Foster	Initial document version
0.2	24-10-2015	Bernhard Thiele	Glossaries and some introductory text
0.3	26-10-2015	Simon Foster	Added the operator definitions and background on UTP
0.4	11-11-2015	Bernhard Thiele	Foundations of Modelica section
0.5	14-11-2015	Simon Foster	Completed full draft for internal review
0.6	09-12-2015	Simon Foster	Completed integrating the internal review comments for my sections
0.7	15-12-2015	Bernhard Thiele	Integrated the internal review comments for my sections

Abstract

This deliverable reports on the progress towards a definition of formal semantics for the continuous-time modelling language *Modelica* in Task 2.3 of INTO-CPS. The basis of dynamic modelling in Modelica is Differential Algebraic Equations (DAEs) and so this initial work focuses on giving a formal semantics to a simple language with DAEs in the context of the Unifying Theories of Programming (UTP). We describe the start of the art in semantics for Modelica, including its implementation of hybrid DAEs. We then give our UTP theory of differential equations, and an experimental mechanisation of this theory in Isabelle/HOL. This UTP theory will provide the basis for the Modelica semantics in the remaining years of INTO-CPS.

Contents

Acronyms

DAE	differential algebraic equation.
EBNF	Extended Backus–Naur Form.
EOO	equation-based object-oriented.
MLS	Modelica Language Specification.
ODE	ordinary differential equation.
PDE	partial differential equation.
PELAB	Programming Environments Laboratory.
RML	Relational Meta Language.

Symbols

$c(t_e)$	vector containing all Boolean condition expressions, *e.g.*, if-expressions.
$m(t_e)$	vector of discrete-time variables of type **discrete Real**, **Boolean**, **Integer**, **String**. Change only at event instants t_e.
.	
$m_{\text{pre}}(t_e)$	values of m immediately before the current event at event instant t_e.
p	parameters and constants.
t	time.
$v(t)$	vector containing all elements in the vectors $x(t)$, $\dot{x}(t)$, $y(t)$, $[t]$, $m(t_e)$, $m_{\text{pre}}(t_e)$, p.
$x(t)$	vector of dynamic variables of type **Real**, *i.e.*, variables that appear differentiated at some place.
$\dot{x}(t)$	differentiated vector of dynamic variables.
$y(t)$	vector of other variables of type **Real** which do not fall into any other category (= algebraic variables).

1 Introduction

This deliverable presents initial work towards creation of a formal denotational semantics for continuous-time models written using the Modelica language [Mod14]. The creation of such a semantics will provide firm mathematical foundations for the language, allow us to consider formal links between Modelica and other languages in INTO-CPS (such as VDM-RT and FMI), and enable the possibility of theorem-proving support for continuous models for the purpose of verification. The Modelica language supports modelling using a conceptual framework based on ordinary differential equations (ODEs) and differential algebraic equations (DAEs) combined with an event handling mechanism. The aim of this deliverable is three-fold:

1. to provide an overview of the foundations of the Modelica language, including the existing work on semantics;
2. to highlight the main aspects of the language that should be considered;
3. to provide an initial UTP [HH98] theory of continuous time and differential equations that will provide the basis for the development of Modelica semantics in subsequent years.

In Section 2 we provide a detailed description of the Modelica language, explain its semantic foundations, different approaches to formal semantics already available, and provide some aspirations for our work. In particular we examine the *flattening* process whereby a collection of Modelica objects is converted to a pure hybrid DAE system. We then describe the hybrid DAE core of the Modelica language, and compare this to the Functional Mockup Interface's (FMI) representation of Hybrid ODEs. We conclude this Section with a collection of Modelica examples. In Section 3 we describe the UTP (Unifying Theories of Programming) semantic framework [HH98] that we will use as a means to give Modelica a formal semantics, along with other continuous time and dynamical systems modelling languages. In Section 4 we describe our UTP theory of differential algebraic equations. This allows the definition of hybrid programs that mix continuous and discrete behaviour, and also specifications regarding their behaviour. This theory will, in subsequent years, form a key part of *INTO-CSP*, a Hybrid CSP [He94] inspired language that we intend to use as a *lingua franca* for all the notations used in INTO-CPS. In Section 5 we report on the current work towards mechanisation of our UTP theory in the Isabelle/HOL proof assistant [NWP02]. Finally in Section 6 we conclude the discussion.

2 Foundations of Modelica

2.1 Objectives of the Modelica Language

Modelica is language for describing the dynamic behaviour of technical systems consisting of mechanical, electrical, thermal, hydraulic, pneumatical, control and other components. The behaviour of models is described with

- *ordinary differential equations (ODEs)*,

- *algebraic equations (AEs)*,
- *event handling* and *recurrence relations* (sampled control).

Object-oriented concepts are supported as a means of managing the complexity inherent to modern technical systems. Modelica can therefore be called an equation-based object-oriented (EOO) language. That term, coined by Broman [Bro10], nicely subsumes the central, distinguishing language characteristics.

Equation-based: Behavior is described declaratively using mathematical equations.

Object-oriented: Objects encapsulate behavior and constitute building blocks that are used to build more complex objects.

The most recent standard version is the Modelica Language Specification (MLS) 3.3 [Mod14]. An important extension in MLS 3.3 if the addition of support for improved discrete-time modelling for control algorithms [Mod14, Chapter 16 and 17] inspired by synchronous languages semantics [BEH+03].

It is worth noting that directly describing models using *partial differential equations (PDEs)* is currently beyond the scope of the Modelica language[1]. However, it is possible to use results of tools that support PDEs or discretise simple PDEs manually.

2.2 Semantic Foundation

Quoting from [Mod14, Section 1.2]:

> The semantics of the Modelica language is specified by means of a set of rules for translating any class described in the Modelica language to a flat Modelica structure. A class must have additional properties in order that its flat Modelica structure can be further transformed into a set of differential, algebraic and discrete equations (= flat hybrid DAE). Such classes are called simulation models.

Figure 1 illustrates the basic idea. It suggests itself to discern the depicted two stages when giving semantics to Modelica models.

In practical Modelica compiler implementations, the translation to a flat Modelica structure is typically mapped to a "*front-end*" phase and the translation to an executable simulation to a "*back-end*" phase of the translation process. Figure 2 depicts a such a typical translation process that is classified into several stages.

Sometimes a dedicated "*middle-end*" phase is defined; it is responsible for the symbolic equation transformations performed during sorting and optimising the hybrid differential algebraic equation (hybrid DAE) into a representation that can be efficiently solved by a numerical solver. In that case the "back-end" phase would only be responsible for code generation (typically C code) from the optimized sorted equations.

The following characterisation of the different stages is taken from [Thi15, p. 39]:

[1] There has been research in that direction, *e.g.*, [Sal06, LZZ08], but so far no common agreement exists whether the Modelica language should be extended to support PDEs and how a such an extension should be done.

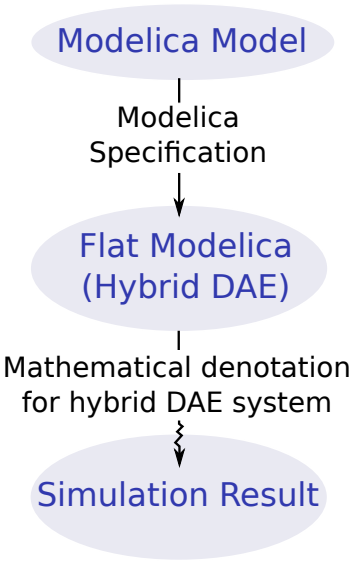


Figure 1: Modelica semantics. From model to simulation result. The squiggle arrow denotes a degree of fuzziness — a simulation result is an *approximation* to the in general inaccessible exact solution of the equation system and the specification does not prescribe a particular solution approach.

Lexical Analysis and Parsing This is standard compiler technology.

Elaboration Involves *type checking*, *collapsing the instance hierarchy* and *generation of connection equations* from connect equations. The result is a hybrid differential algebraic equation (DAE) (consisting of variable declarations, equations from equations sections, algorithm sections, and `when`-clauses for triggering discrete-time behavior).

Equation Transformation This step encompasses transforming and manipulating the equation system into a representation that can be efficiently solved by a numerical solver. Depending on the intended solver the DAE is typically reduced to an index one problem (in case of a DAE solver) or to an ODE form (in case of numerical integration methods like Euler or Runge-Kutta).

Code generation For efficiency reasons tools typically allow (or require) translation of the residual function (for an DAE) or the right-hand side of an equation system (for an ODE) to C-code that is compiled and linked together with a numerical solver into an executable file.

Simulation Execution of the (compiled) model. During execution, the simulation results are typically written into a file for later analysis.

The Modelica Language Specification (MLS) is described using the English language, its semantics is therefore to some extent subject to interpretation. It appears also notable that the intermediate hybrid DAE representation (denoted as *Flat Modelica* in Figure 2) is not formally specified in the MLS, *i.e.*, no concrete syntax description, *e.g.*, in Extended Backus–Naur Form (EBNF), is provided (despite the conceptual importance of Flat Modelica). The translation from an instantiated Modelica model to a Flat Modelica representation is called *flattening* of Modelica models (hierarchical constructs are eliminated and a "flat" set of Modelica "statements" is produced).

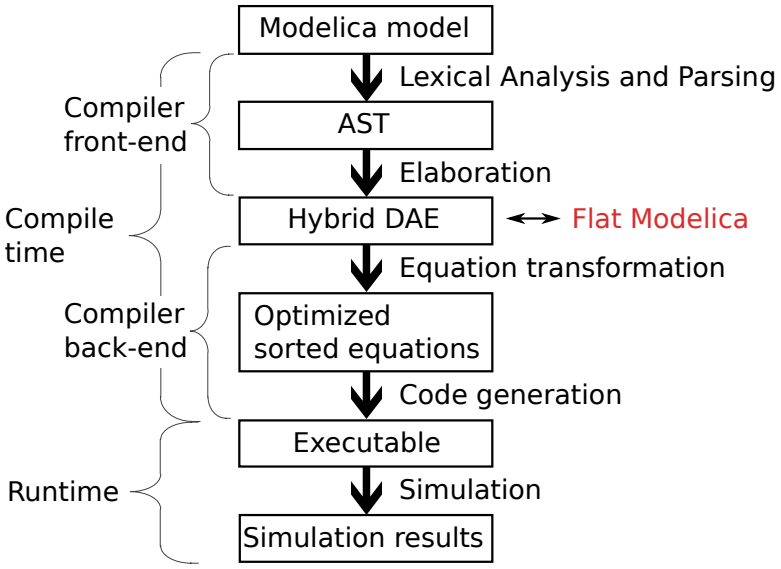


Figure 2: The typical stages of translating and executing a Modelica model.

The MLS [Mod14, Appendix C] discusses how a "flat" set of Modelica "statements" is mapped into an appropriate mathematical description form denoted as *Modelica DAE*. It is this mapping to a mathematical description that allows to associate dynamic semantics to the declarative Flat Modelica description. This will be further discussed in Section 2.4.

2.3 Formal Specification Approaches for Modelica

This section gives a brief (non-exhaustive) account of previous work related to formally specifying aspects of the Modelica language.

2.3.1 Operational Semantics

A first attempt to formalise aspects of the Modelica language was made at a time where no complete implementation was available and was performed by Kågedal and Fritzson to discover and close gaps in the early specification documents [KF98]. Kågedal and Fritzson distinguish between *static semantics* (which describes how the object-oriented structuring of models and their equations work) and the *dynamic semantics* (which describes the simulation-time behaviour) of Modelica models. Like most later work on formalizing Modelica semantics the work by Kågedal and Fritzson [KF98] addresses the *static semantics* and does not intend to describe the equations solving process, nor the actual simulation. The formal semantics provided in their work is expressed in a high-level specification language called Relational Meta Language (RML) [Pet95], which is based on natural semantics (an operational semantics specification style). A compiler generation system allows generating a translator from a language specification written in RML. The paper explains the basic ideas behind that approach, but it does not list the complete RML source-code that was written during that early effort.

Development and usage of efficient language implementation generators has been a long-

term research effort at the Programming Environments Laboratory (PELAB)[2] at Linköping University. In [FPBA09], Peter et al. report on practical experience gained with various approaches. The biggest effort was developing a complete Modelica compiler using RML as the specification language. This implementation formed the base for the *OpenModelica* environment [FAL+05]. A remarkable observation reported in the paper is the enormous growth of the RML code over time; we reproduce the data below (where we refer to lines of RML code, including comments).

Lines OM 1998	Lines OM 2003	Lines OM 2006
8709	36050	80064

Nowadays the development of OpenModelica has swapped from RML to *MetaModelica*. MetaModelica is a language developed at PELAB that introduces language modelling features known from languages like RML into the Modelica language. One of its development motivations was to provide a language with a more gentle learning curve than RML, particularly in regard to prospective OpenModelica developers without a background in formal specification methods. MetaModelica [PF06] has evolved over the years and, more recently, a bootstrapping version has been used in the OpenModelica development [SFP14].

An ongoing project that aims to formalize a clocked discrete-time subset of the Modelica language is the CERTMOD[3] project. First results that formally describe the static-name lookup mechanism are published in [SCAP15].

2.3.2 Reference Attribute Grammars

Åkesson et al. report on leveraging modern attribute grammars mechanisms to support the *front-end* phase of a Modelica compiler [ÅEH10]. They use the attribute grammars metacompilation system JastAdd for implementing their Modelica compiler. The Modelica compiler of the open-source JModelica.org[4] platform is based on this technology.

2.3.3 Translational Semantics

A substantial extension in the MLS 3.3 was the integration of language elements motivated by synchronous languages like Lustre, Esterel, or Signal [BEH+03]. The synchronous-language paradigm has been particularly successful in the development of safety-critical control software like flight control systems. Thiele et al. [TKF15] discuss an approach that leverages the synchronous-language elements extension of Modelica to enable a model-based development process for safety-related applications. For that, they aim at mapping a Modelica subset for digital control-function development to a well understood synchronous data-flow kernel language. That mapping is established by providing a *translational semantics* from the Modelica subset to the synchronous data-flow kernel language.

[2] http://www.ida.liu.se/labs/pelab/ (Oct, 2015).
[3] http://cordis.europa.eu/project/rcn/111584_en.pdf, (Nov, 2015).
[4] http://www.jmodelica.org/ (Oct, 2015).

2.3.4 Summary

Several formal specification approaches have been used to give semantics to Modelica, however one needs to be aware of their scopes and restrictions.

The translational semantics approach (Section 2.3.3) is restricted to a discrete-time subset of the Modelica language, deemed suitable for safety-related applications. The subset is strongly restricted in order to lower tool qualification efforts for a code generator that is based on that subset (trade-off between language expressiveness and tool qualification efforts). There is no notion of differential equations within that language subset. Consequently, semantic models for continuous-modelling languages is not within the scope of that previous work.

Both the operational semantics approach based on RML/MetaModelica (Section 2.3.1) and the reference attribute-grammar approach based on JastAdd (Section 2.3.2) are targeted at describing the instantiation and flattening of Modelica models (*static semantics*). The result after that stage is basically the Flat Modelica representation (see Figure 2). In addition RML/MetaModelica is also used in the equation transformation ("middle-end") of the compiler. In both cases, the primary rationale of the formal description is generating compilers using language-implementation generators, rather than (mechanised) proving of certain properties. The operational semantics approach used in the ongoing CERTMOD[5] project aims to describe static and dynamic semantics for a discrete-time subset of Modelica.

Neither the operational semantics approach nor the reference-attribute grammar approach targets the *dynamic semantics* of hybrid systems (*i.e.* the simulation-time behavior of dynamic systems that exhibit both continuous and discrete dynamic behavior) of Modelica models. The aim of this work is to investigate the formalization of Modelica language aspects related to the dynamic semantics of hybrid system models. In particular, we will formalise a UTP denotational semantics for both dynamic modelling and event-based constructs. This can then later be used to derive a complete operational semantics for the kernel language, which we will describe further in Sections 3 and 4.

2.4 Modelica Hybrid DAE Representation

[Mod14, Appendix C] discusses the mapping of a Modelica model into an appropriate mathematical description form. This section describes the principle form of the representation without aiming to cover every detail and special case[6]. Table 3 describes the symbols used in the mathematical description.

2.4.1 Equations

Flat Modelica can be conceptually mapped to a set of equations consisting of differential, algebraic and discrete equations of the following form (see Table 3 for brief description of the used symbols):

[5] http://cordis.europa.eu/project/rcn/111584_en.pdf, (Nov, 2015).

[6] *E.g.*, the semantics of operators like `noEvent()`, or `reinit()` is not covered.

Symbol	Description
p	parameters and constants
$p^{\mathbf{B}}$	parameters and constants of type **Boolean**, $p^{\mathbf{B}} \subseteq p$
t	time
$x(t)$	vector of dynamic variables of type **Real**, *i.e.*, variables that appear differentiated at some place
$\dot{x}(t)$	differentiated vector of dynamic variables
$y(t)$	vector of other variables of type **Real** which do not fall into any other category (= algebraic variables)
$m(t_e)$	vector of discrete-time variables of type **discrete Real**, **Boolean**, **Integer**, **String**. Change only at event instants t_e
$m^{\mathbf{B}}(t_e)$	vector of discrete-time variables of type **Boolean**, $m^{\mathbf{B}}(t_e) \subseteq m(t_e)$. Change only at event instants t_e
$m_{\text{pre}}(t_e)$	values of m immediately before the current event at event instant t_e
$m^{\mathbf{B}}_{\text{pre}}(t_e)$	values of $m^{\mathbf{B}}$ immediately before the current event at event instant t_e, $m^{\mathbf{B}}_{\text{pre}}(t_e) \subseteq m_{\text{pre}}(t_e)$
$c(t_e)$	vector containing all Boolean condition expressions, *e.g.*, if-expressions
$v(t)$	vector containing all elements in the vectors $x(t)$, $\dot{x}(t)$, $y(t)$, $[t]$, $m(t_e)$, $m_{\text{pre}}(t_e)$, p

Table 3: Notation used in the Modelica hybrid DAE representation.

1. *Continuous-time behaviour.* The system behavior *between* events is described by a system of differential and algebraic equations (DAEs):

$$f\Big(x(t), \dot{x}(t), y(t), t, m(t_e), m_{\text{pre}}(t_e), p, c(t_e)\Big) = 0 \tag{1a}$$

$$g\Big(x(t), y(t), t, m(t_e), m_{\text{pre}}(t_e), p, c(t_e)\Big) = 0 \tag{1b}$$

2. *Discrete-time behaviour.* Behaviour at an event at time t_e. An event fires if any of condition $c(t_e)$ change from **false** to **true**. The vector-value function f_m specifies the right-hand side (RHS) expression to the discrete variables $m(t_e)$. The argument $c(t_e)$ is made explicit for convenience (alternatively it could have been incorporated directly into f_m). The vector $c(t_e)$ is defined by the vector-value function f_e which contains all `Boolean` condition expressions evaluated at the most recent event t_e.

$$m(t_e) := f_m\Big(x(t_e), \dot{x}(t_e), y(t_e), m_{\text{pre}}(t_e), p, c(t_e)\Big) \tag{2}$$

$$c(t_e) := f_e\Big(m^{\mathbf{B}}(t_e), m^{\mathbf{B}}_{\text{pre}}(t_e), p^{\mathbf{B}}, rel(v(t_e))\Big) \tag{3}$$

where $rel(v(t_e)) = rel([x(t); \dot{x}(t); y(t); t; m(t_e); m_{\text{pre}}(t_e); p])$ is a Boolean-typed vector-valued function containing the relevant elementary relational expressions ("<", "<=", ">", ">=", "==", "<>") from the model, containing variables v_i, e.g., $v_1 > v_2,\ v_3 \geq 0$.

2.4.2 Simulation

Simulation means that an initial value problem (IVP) is solved. The equations define a DAE which may have discontinuities, a variable structure and/or which are controlled by a discrete-event system. Simulation is performed in the following way:

1. The DAE (1) is solved by a numerical integration method. Conditions c as well as the discrete variables m are kept constant. Therefore, (1) is a continuous function of continuous variables and the most basic requirement of numerical integrators is fulfilled.
2. During integration, all relations from (3) are monitored. If one of the relations changes its value an event is triggered, *i.e.*, the exact time instant of the change is determined and the integration is halted.
3. At an event instant, (1)–(3) is a mixed set of algebraic equations, which is solved for the Real, Boolean and Integer unknowns. New values of the discrete variables m and of new initial values for the states x are determined.
4. After an event is processed, the integration is restarted at 1.

There might have been discontinuous variable changes at an event that trigger another event. This case is described in the next section.

2.4.3 Event Iteration

At an event instant, including the initial event, the model equations are reinitialized according to the following iteration procedure:

known variables: $x,\ t,\ p$
unkown variables: $\dot{x},\ y,\ m,\ m_{\text{pre}},\ c$
// m_{pre} = value of m before event occured
loop
  solve (1)–(3) for the unknowns, with m_{pre} fixed
  if $m = m_{\text{pre}}$ **then break**
  $m_{\text{pre}} := m$
end loop

The iterative process of triggering events and solving the reinitialization problem is called *event iteration*. It is an example of a so-called fixed-point procedure, *i.e.*, the iterative process is assumed to converge to a fixed point.

2.4.4 Remarks on DAEs

DAEs are distinct from ODEs in that they are not completely solvable for the derivatives of all components of the function $x(t)$ because these may not all appear (*i.e.*, some equations are algebraic).

Consider following DAE given in the general form

$$f(x, \dot{x}, y, t) = 0 \tag{4}$$

where $x = x(t), \dot{x} = \dot{x}(t) \in \mathbb{R}^n, y = y(t) \in \mathbb{R}^m, f : G \subseteq \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R} \to \mathbb{R}^{n+m}$.

For a set of initial conditions $(x_0, \dot{x}_0, y_0, t_0) = 0$ to be consistent, it must satisfy system (4) at an initial time t_0:

$$f(x_0, \dot{x}_0, y_0, t_0) = 0. \tag{5}$$

The consistent initialization of such systems is the problem that Pantelides considered in his famous paper [Pan88].

Note that "initial conditions" here refers to the vector $(x_0, \dot{x}_0, y_0)$ rather than simply (x_0, y_0), hence elements of $\dot{x}_0$ may appear in (5). There is another complication as [Pan88] describes:

> Differentiating some or even all of the original equations produces new equations which must also be satisfied by the initial conditions. This need not necessarily constrain the vector $(x_0, \dot{x}_0, y_0)$ further: differentiation can also introduce new variables (time-derivatives of $\dot{x}$ and y) and it may well be the case that the new equations can be satisfied for all possible values of the initial conditions and appropriate choices of values for the new variables. Thus, in this case no useful information is generated by differentiation.

Pantelides then proposes an algorithm to analyze the structure of the system equations and determine the minimal subset for which differentiation may yield useful information in the sense that it imposes further constraints on the vector of initial conditions. This algorithm is not only useful for the initialization of a DAE system, but also for transforming the DAE system to a corresponding ODE system. The algorithm is further detailed below.

An important property for DAEs is the notion of the *DAE index*. There exist different definitions for DAE index in the literature. The following informal definition is from [Fri14, Section 18.2.2]. A more formal definition can be found in [AP98, Definition 9.1, p. 236].

Definition 2.1. DAE index or differential index. The *differential index* of a general DAE system is the minimum number of times that certain equations in the system need to be differentiated to reduce the system to a set of ODEs, which can then be solved by the usual ODE solvers.

The DAE index can therefore be seen as measurement for the distance between a DAE and a corresponding ODE.

An ODE system in *explicit state-space form* is a DAE system of *index 0*:

$$\dot{x} = f(x, t) \tag{6}$$

The following semi-explicit form of DAE system:

$$\dot{x} = f(x, y, t) \tag{7a}$$
$$0 = g(x, y, t) \tag{7b}$$

is of *index 1* if $\frac{\partial g(x,y,t)}{\partial y}$ is non-singular, because then one differentiation of (7b) yields $\dot{y}$ in principle.

DAEs with an index > 1 are called *higher-index DAEs*.

Bachmann et al. [BAF06, Section 4.4] describe the typical approach in which higher-index DAEs are solved by a Modelica tool:

1. Use Pantelides algorithm to determine how many times each equation has to be differentiated to reduce the *index* to one or zero.
2. Perform *index reduction* of the DAE by analytic symbolic differentiation of certain equations and by applying the method of dummy derivatives [MS93]. The method of dummy derivatives for index reduction augments the DAE system with differentiated versions of equations, and replaces some of the differentiated variables with new algebraic variables called dummy derivatives [Fri14, Section 18.2.4.1].
3. Select the core state variables to be used for solving the reduced problem. These can either be selected statically during compilation, or in some cases selected dynamically during simulation.
4. Use a numeric ODE solver to solve the reduced problem.

We can potentially invoke a similar approach in our semantics by only considering the underlying ODEs themselves, and using the index reduction to convert from the Modelica-level DAE. Nevertheless, there may be benefit for theorem proving in directly considering DAEs, and so we will consider both possibilities.

2.5 Super-Dense Time

The Functional Mock-up Interface (FMI) 2.0 standard for model exchange discusses a mathematical description for ODEs in state space form with event handling denoted as a "*hybrid ODE*" [FMI14, Section 3.1]. Modelica tools contain symbolic algorithms for DAE index reduction. These algorithms allow reduction of the DAE index in order to solve DAEs using numerical reliable methods and therefore also allow transform of the DAE formulation to an ODE formulation (transforming the DAE to an ODE for simulation is indeed the preferred method for some Modelica tools).

Studying the FMI description for hybrid ODEs is also interesting in the context of providing a dynamic semantics to Modelica models for the following reasons:

1. DAE index reduction methods allow transformation of a Modelica hybrid DAE to an FMI hybrid ODE.
2. It allows relation of the mathematical description used in the FMI standard [FMI14, Section 3.1] with the mathematical description from the Modelica standard [Mod14, Appendix C].

The mathematical description of FMI's Model Exchange interface [FMI14, Section 3.1] employs the concept of *super-dense time* for giving semantics to hybrid ODEs.
Definition 2.2. Super-dense time The independent variable time $t \in \mathbb{T}$ is a tuple $t = (t_R, T_I)$ where $t_R \in \mathbb{R}, t_I \in \mathbb{N} = \{0, 1, 2, \ldots\}$, see *e.g.*, [LZ07].

Super-dense time provides a suitable formalism to reason about the dynamical evolution of hybrid system variables. Table 4 describes the ordering defined on super-dense time. Figure 3 depicts the concept graphically.

Operation	Mathematical meaning	Description
$t_1 < t_2$	$(t_{R1}, t_{I1}) < (t_{R2}, t_{I2}) \Leftrightarrow (t_{R1} < t_{R2}) \vee ((t_{R1} = t_{R2}) \wedge (t_{I1} < t_{I2}))$	t_1 is before t_2
$t_1 = t_2$	$(t_{R1}, t_{I1}) = (t_{R2}, t_{I2}) \Leftrightarrow (t_{R1} = t_{R2}) \wedge (t_{I1} = t_{I2})$	t_1 is identical to t_2
t^+	$(t_R, t_I)^+ \Leftrightarrow (\lim_{\varepsilon \to 0}(t_R + \varepsilon), t_{Imax})$	right limit at t. t_{Imax} is the largest occurring Integer index of super dense time
^-t	$^-(t_R, t_I) \Leftrightarrow (\lim_{\varepsilon \to 0}(t_R - \varepsilon), 0)$	left limit at t
$^\bullet t$	$^\bullet(t_R, t_I) \Leftrightarrow \begin{cases} ^-t & \text{if } t_I = 0 \\ (t_R, t_I - 1) & \text{if } t_I > 0 \end{cases}$	previous time instant (= either left limit or previous event instant)
v^+	$v(t^+)$	value at the right limit of t
^-v	$v(^-t)$	value at the left limit of t
$^\bullet v$	$v(^\bullet t)$	previous value (= either left limit or value from the previous event)

Table 4: Ordering defined on super-dense time $\mathbb{T}$.

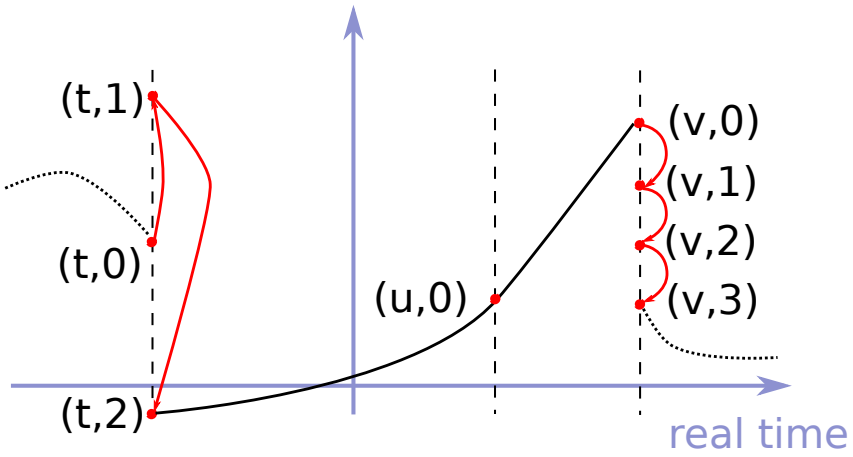


Figure 3: Super-dense time modelling $\mathbb{T} = \mathbb{R} \times \mathbb{N}$.

In particular, note that super-dense time allows signals to have an ordered sequence of values at the same time instant. This is important since a time model based on the set of real numbers ($t \in \mathbb{R}$) is not semantically rich enough to capture an ordered sequence of signal values during an event iteration as described in Section 2.4.3.

2.6 Examples

This section points out aspects of Modelica by providing a collection of concrete examples. These serve to illustrate the key semantic concepts from Modelica, and how the current simulation tool implements them.

2.6.1 Hello World (ODEs)

The "hello world" program of Modelica is a simple ODE. Listing 1 describes the initial value problem (IVP)

$$\dot{x} = -x, \tag{8}$$
$$\text{with initial condition (IC): } x(t_0) = 1.$$

Listing 1: HelloWorld example.

```
model HelloWorld "Simple ODE"
  parameter Real c = -1;
  Real x(start=1, fixed=true);
equation
  der(x) = c*x;
end HelloWorld;
```

The model is simulated in OpenModelica using the settings below (for the remaining settings, default settings are used):

startTime	0
stopTime	10
numberOfIntervals	10

Variable `x` of the `HelloWorld` example is plotted in Figure 4. The dot within the graph

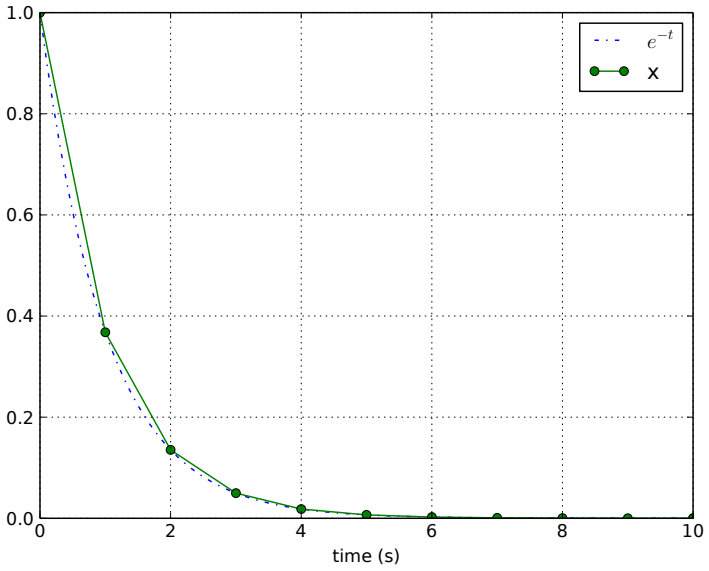


Figure 4: Plot for the "Hello World" example (Listing 1).

of `x` depicts an entry for the variable at this point within the generated result file. In addition the symbolic solution of the IVP (8), $x(t) = e^{-t}$, is plotted.

2.6.2 Cartesian Pendulum (DAEs)

A frequently presented example of a DAE system results from modelling the motion of a pendulum in Cartesian coordinates.

Figure 5 shows a pendulum in Cartesian coordinates (x, y) with the center in $(0, 0)$ and a length l. Of course, it is well known that the equations simplify to the ODE $l\ddot{\theta} = -g \sin\theta$

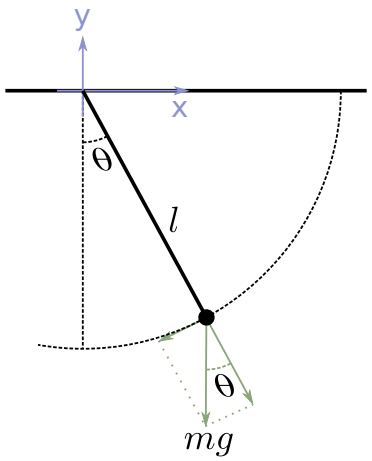


Figure 5: Pendulum with Cartesian coordinates (x, y).

if using a polar coordinate system instead of a Cartesian coordinate system. However, this is not the point of this example (but of course results based on this ODE can be used to validate the DAE solution based on using Cartesian coordinates).

The Lagrangian $L = T - V$ can be computed from the kinetic energy T and the potential energy V.

$$L = \frac{1}{2}(\dot{x}^2 + \dot{y}^2) - mgy.$$

The motion is restricted on to a circle with radius l that gives rise to the equality constraint

$$g = x^2 + y^2 - l^2 = 0.$$

Using the method of Lagrange multipliers, the equality constraint can be included in the new Lagrangian

$$L' = L + \lambda g = \frac{1}{2}(\dot{x}^2 + \dot{y}^2) - mgy + \lambda(x^2 + y^2 - l^2)$$

where λ is a Lagrange multiplier[7]. The Euler-Lagrange equation, then, is given by

$$\frac{d}{dt}\frac{\partial L'}{\partial(\dot{x}, \dot{y}, \dot{\lambda})} - \frac{\partial L'}{\partial(x, y, \lambda)} = 0.$$

Evaluation of the Euler-Lagrange equation, substituting $\dot{x} = u$, $\dot{y} = v$ and assuming mass $m = 2$, results in the DAE system

$$\dot{x} = u, \qquad \dot{y} = v, \tag{9a}$$

$$\dot{u} = \lambda x, \qquad \dot{v} = \lambda y - g, \tag{9b}$$

$$x^2 + y^2 = l^2. \tag{9c}$$

[7] See, *e.g.*, section "Lagrange multipliers and constraints" at https://en.wikipedia.org/wiki/Lagrangian_mechanics (Nov, 2015).

The momentum variables u and v should be governed by the law of conservation of energy and their direction should point along the circle. Neither condition is explicit in the equations above.

Since this is an initial value problem (IVP), a compatible set of initial conditions needs to be provided. One possibility is to use $(x(t_0) = x_0, u(t_0) = u_0)$, where x_0 is constrained to be between $-l \leq x_0 \leq l$. In addition, a sign is needed for the value of $y(t_0)$, since there are the two possibilities $y(t_0) = \pm\sqrt{l^2 - x^2}$.

The "CartesianPendulum" Modelica model is given in Listing 2.

Listing 2: Cartesian pendulum example.

```
1  model CartesianPendulum "modelling the motion of a pendulum
       in Cartesian coordinates"
2    parameter Real l = 1;
3    parameter Real g = 9.8;
4    parameter Real x_start = 0.5;
5    Real x(start=x_start, fixed=true);
6    Real u(start=0, fixed=true);
7    Real y(start= -sqrt(l^2 - x_start^2));
8    Real v,lambda;
9  equation
10   der(x) = u;
11   der(u) = lambda*x;
12   der(y) = v;
13   der(v) = lambda*y - g;
14   x^2 + y^2 = l^2;
15 end CartesianPendulum;
```

Notice that that start values (initial conditions) are specified for variables `x`, `u`, and `y` (lines 5 – 7). The start values for `x` and `u` are specified as `fixed`, but not so the start value for `y`. If no value for `fixed` is provided, the default value `fixed=false` is used for variables (for constants and parameters the default value for `fixed` is `true`). The meanings for variable attributes `start` and `fixed` are as follows:

Attribute	Meaning
`start`	Initial value of variable
`fixed`	= `true`: Value of `start` is a fixed initial condition = `false`: Value of `start` is a "guess" value (*i.e.*, the value can be changed by the simulator in order to find a consistent set of start values)

Hence, line 7 does not specify a fixed initial condition for `y` (if it did, the initialization problem would be overspecified), instead it proposes a "guess" value for `y` in order to guide the initialization algorithm to find the solution of y in the negative realm, *i.e.*, $y = -\sqrt{l^2 - x^2}$ (the situation depicted in Figure 5) rather than possibly selecting $y = \sqrt{l^2 - x^2}$ (which is also a valid solution that corresponds to the pendulum being in a "standing" position).

The model in Listing 2 can be simulated in OpenModelica. Simulation results for variables `x` and `y` are plotted in Figure 6.

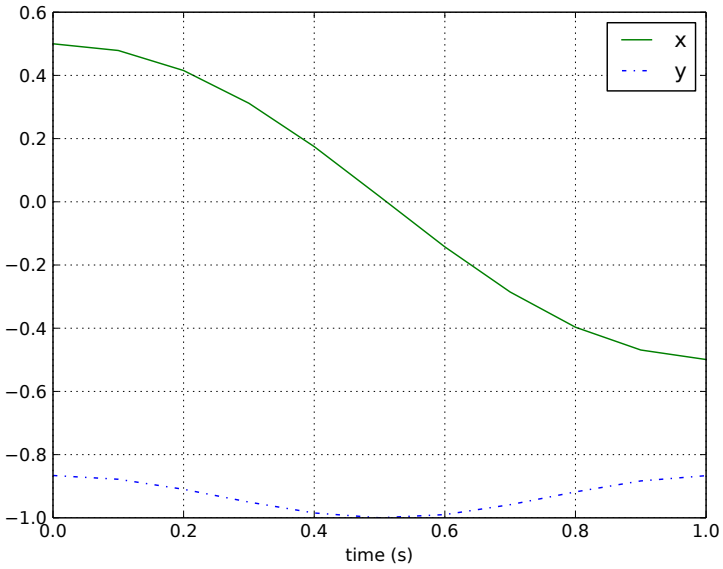


Figure 6: Plot for the Cartesian pendulum example (Listing 2).

Remark. In order to transform DAE system (9) to an ODE system (see Section 2.4.4), equation (9c) needs to be differentiated three times. First time derivative:

$$\begin{aligned} & \dot{x}x + \dot{y}y = 0 \\ \Rightarrow \quad & ux + vy = 0. \end{aligned}$$

Second time derivative:

$$\begin{aligned} & \dot{u}x + \dot{v}y + u\dot{x} + v\dot{y} = 0 \\ \Rightarrow \quad & \lambda(x^2 + y^2) - gy + u^2 + v^2 = 0 \\ \Rightarrow \quad & l^2\lambda - gy + u^2 + v^2 = 0 \end{aligned}$$

Third time derivative:

$$\begin{aligned} & l^2\dot{\lambda} - g\dot{y} + 2u\dot{u} + 2v\dot{v} = 0 \\ \Rightarrow \quad & l^2\dot{\lambda} - gv + 2\lambda ux + 2\lambda vy - 2gv = 0 \\ \Rightarrow \quad & l^2\dot{\lambda} - 3gv = 0. \end{aligned}$$

Hence, the DAE system (9) is a higher-index DAE with a differential index of 3, which is typical for constrained mechanical systems.

2.6.3 If-expressions (relation triggered events)

The aim of this section is to introduce *relation triggered events*. Relation triggered events are explained by using a discontinuous function of the form

$$y(x) = \begin{cases} a_1 x + b_1 & \text{if } x < x_1 \\ a_2 x + b_2 & \text{if } x_1 \leq x < x_2 \\ a_3 x + b_3 & \text{else} \end{cases} \tag{10}$$

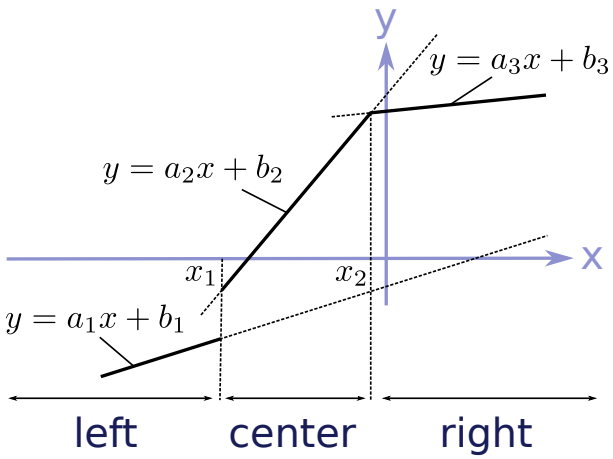


Figure 7: Schematic diagram of a discontinuous function of the form (10).

depicted in Figure 7; it is inspired by [Cel13, lecture 22].

A Modelica model using **if**-expressions to describe two equations that resemble (10) is shown below (however, notice that the model declares *equations*, not functions!).

Listing 3: If-expressions example.

```
1  model IfExpressions
2    parameter Real a1 = 1, a2 = 3, a3 = 0.5;
3    parameter Real x1 = -3, x2 = -1;
4    parameter Real b1 = -1;
5    parameter Real b2 = a1*x1 + b1 - a2*x1 + 2;
6    parameter Real b3 = a2*x2 + b2 - a3*x2;
7    Real x(start=-6), y, z;
8  equation
9    der(x) = 1;
10   y = if x < x1 then a1*x + b1
11       else if x < x2 then a2*x + b2
12       else a3*x + b3;
13   y = if z < x1 then a1*z + b1
14       else if z < x2 then a2*z + b2
15       else a3*z + b3;
16 end IfExpressions;
```

The model is simulated in OpenModelica using the settings below (for the remaining settings, default settings are used):

startTime	0
stopTime	10
numberOfIntervals	3

The variables `y` and `z` are plotted in Figure 8. The equation in line 13–15 is solved for `z` and not for `y`. Hence, if the equation in line 10–12 is considered as a function $f : X \to Y$, then the equation in line 13–15 can be considered as the inverse function $f^{-1} : Y \to X$.

Dots within the graph depict a data entry for the respective variable within the generated

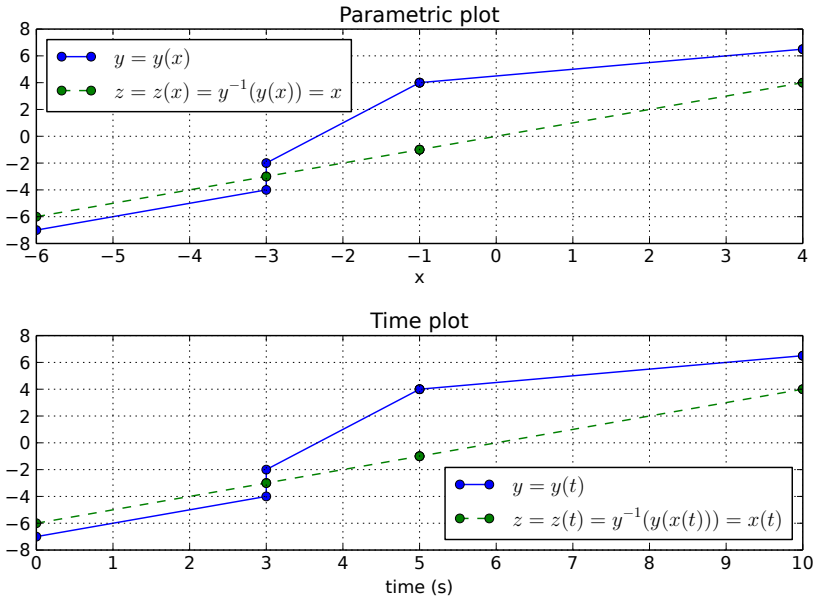


Figure 8: Plot for if-expression example.

result file. Note that the plot has entries at the switching points. The solver detects if a relation changes its value and determines the time instant within some prescribed accuracy. After determining the time instant, continuous-time integration proceeds until that time instant, integration is halted and an event is triggered, the if-clause is changed at the event and integration is restarted.

2.6.4 Hysteresis (Discrete State Variables)

The hysteresis relation depicted in Figure 9 can be modelled by combining `Real` and `Boolean` equations with *discrete state variables*.

Listing 4: Hysteresis example.

```
model Hysteresis
  Real y,u;
  Boolean high(start=true, fixed=true);
equation
  u = 1.5*cos(time);
  high = not pre(high) and u >= 1 or
              pre(high) and u > -1;
  y = if high then 0.5 else -0.5;
end Hysteresis;
```

The Boolean variable `high` is a discrete state variable and therefore needs an initial value. The Expression `pre(high)` returns the value of `high` immediately before the current event at event instant t_e. In the hybrid DAE representation presented in Section 2.4.1 that corresponds to `pre(high)` $\in m_{\text{pre}}(t_e)$. Using the super-dense time model (see Table 4), expression `pre(v)` corresponds to ${}^{\bullet}v$. It is possible to formulate the equation for `high`

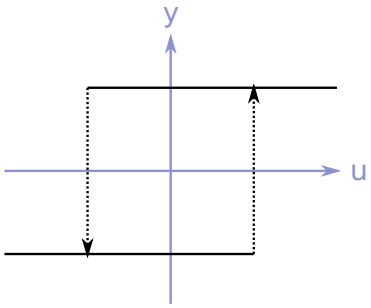


Figure 9: Hysteresis relation.

in the more compact, but possibly harder to understand form "`high = u >= 1 or pre (high)and u > -1;`".

The model is simulated in OpenModelica using the settings below (for the remaining settings, default settings are used):

startTime	0
stopTime	10
numberOfIntervals	10

Variables `u`, `y`, and `high` are plotted in Figure 10. Dots within the graph depict a data entry for the respective variable within the generated result file. For the `Boolean` variable `high` the value "1" depicts `true` while "0" depicts `false`. The plot has entries at the event points that show that the solver detected the switching point within some prescribed accuracy.

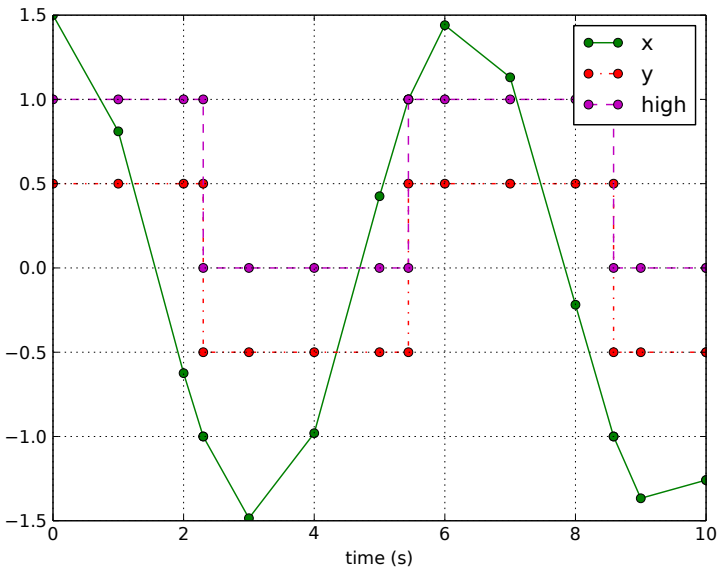


Figure 10: Plot for Hysteresis example.

2.6.5 When-equations (Instantaneous Equations)

The aim of this section is to introduce *instantaneous equations* and their activation at *events*.

Relations trigger events (see Section 2.6.3) by implicitly introducing event conditions $c(t)$ (compare equation (3)) that are monitored by the solver during continuous-time integration. In the Modelica language, Boolean variables are used to signal events (hence, there is no dedicated data type for events). Figure 11 describes various event-related operators that are available in the language. It is possible to associate equations to an

`edge(b)`	Returns `true` if Boolean variable `b` changes from `false` to `true` ≡ "`(b and not pre(b))`".
`change(b)`	Returns `true` if a non-`Real` variable `v` changes its value ≡ "`(v<>pre(v))`".
`sample(t0,td)`	Returns `true` at times $t_0 + i \cdot t_d$, where $i = 0, 1, 2, \ldots$.
`initial()`	Returns `true` during the initialization phase and `false` otherwise.

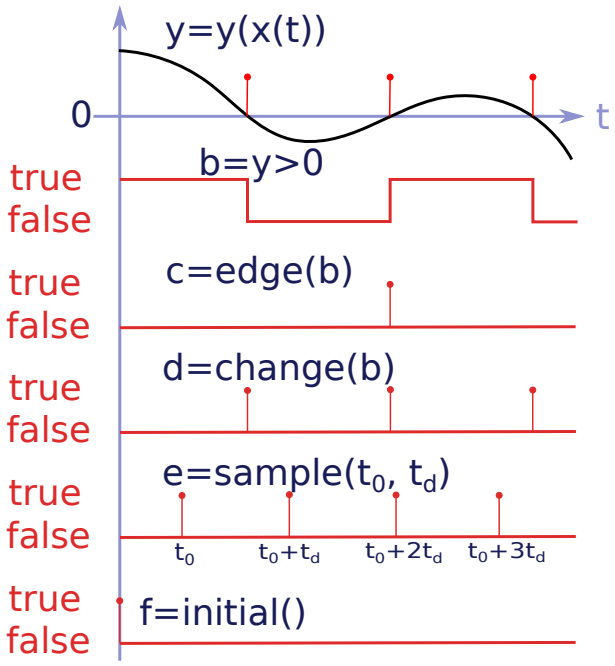


Figure 11: Important event related operators (see [Mod14, Section 3.7.3])

event that are only active during the event (i.e., at a time interval of zero duration) and are otherwise deactived by using **when**-equations. These equations are called *instantaneous equations*.

Listing 5 shows the usage of **when**-equations and the event operators introduced in Figure 11.

Listing 5: When-equations and event related operators example.

```
model WhenEquations
  "When-equations and event related operators"
  Boolean c(start=false,fixed=true), ev1, ev2;
  Integer di1(start=0,fixed=true), di2(start=0,fixed=true),
          di3(start=0,fixed=true), di4(start=0,fixed=true);
  Real x;
equation
  x = sin(time);
  c = x > 0;
  when {c} then
    di1 = pre(di1) + 1;
```

```
12    end when;
13    // Conceptual mapping of when equations to if expressions
14    ev1 = edge(c) and not initial();
15    di2 = if ev1 then pre(di2) + 1 else pre(di2);
16
17    // Other event related operators
18    ev2 = change(c);
19    when {initial(), ev2} then
20      di3 = pre(di3) + 1;
21    end when;
22    when {sample(7.0, 1.0)} then
23      di4 = pre(di4) + 1;
24    end when;
25  end WhenEquations;
```

The model is simulated in OpenModelica using the settings below (for the remaining settings, default settings are used):

startTime	0
stopTime	10
numberOfIntervals	5

The Variables `di1`, `di2`, `di3`, `di4` and the zero-crossing function $x = \sin(t)$ are plotted in Figure 12. The triangle markers within the graph depict a data entry for the respective variable within the generated result file. A when-equation is activated if there is an "edge" in one of its Boolean conditions **when** $\{b_1, b_2, \ldots, b_n\}$ **then** (line 10) or by explicit events like in lines 19 and 22. Furthermore, when-equations are *not active during initialization*. However, by using the `initial()` operator, when-equations can be activated explicitly during initialization (see line 19).

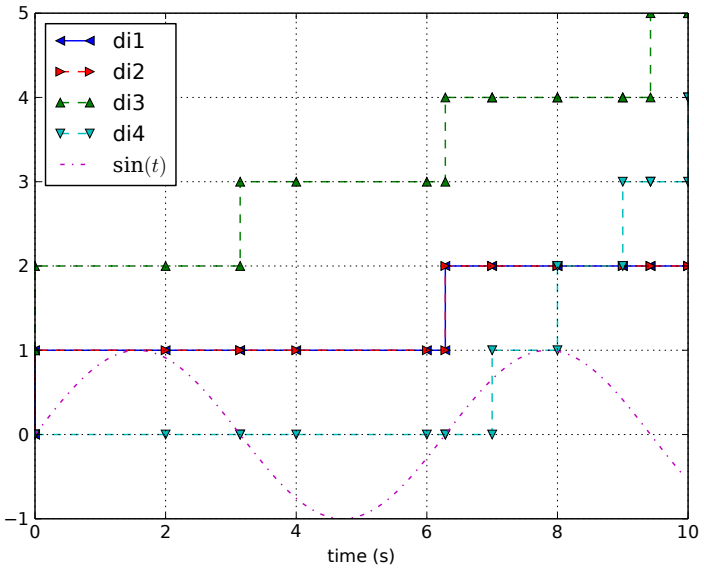


Figure 12: Plot for when-equations and event related operators example.

Remark. [Mod14, Section 8.3.5.1] defines a conceptual mapping from when-equations to if-expressions. The idea is indicated in Listing 5 where `di1` (lines 10–12) has equal observable behaviour as `di2` (lines 14–15) (see Figure 12).

2.6.6 Event Iteration

A model featuring *event iteration* as described in Section 2.4.3 is given if Listing 6.

Listing 6: Event iteration example.

```
1  model EventIteration "Event iterations and initialization"
2    Boolean iev1(start=false,fixed=true),
3            ev1(start=false,fixed=true),
4            ev2(start=false,fixed=true),
5            ev3(start=false,fixed=true),
6            c(start=false,fixed=true);
7    Integer v1(start=0,fixed=true), v2(start=0,fixed=true),
8            v3(start=0,fixed=true);
9    Real x, du, y;
10 equation
11   iev1 = initial();
12   ev1 = pre(iev1);
13   v1 = if iev1 and ev1 then pre(v1) + 1 else pre(v1);
14   v2 = if iev1 or ev1 then pre(v2) + 1 else pre(v2);
15
16   x = sin(time);
17   c = x > 0;
18   ev2 = edge(c);
19   ev3 = pre(ev2);
20   // Not valid to replace initial() by iev1!
21   when {initial(), ev1, ev2, ev3} then
22     Modelica.Utilities.Streams.print("v3: " + String(v3));
23     v3 = pre(v3) + 1;
24     du = v3 + y;
25   end when;
26   der(y) = 0.1*v3;
27 initial equation
28   y = v2 + v3;
29 end EventIteration;
```

The model is simulated in OpenModelica using the settings below (for the remaining settings, default settings are used):

startTime	0
stopTime	10
numberOfIntervals	5

The Variables `v1`, `v2`, `v3`, `y` and the zero-crossing function $x = \sin(t)$ are plotted in Figure 13. The triangle markers within the graph depict a data entry for the respective variable within the generated result file.

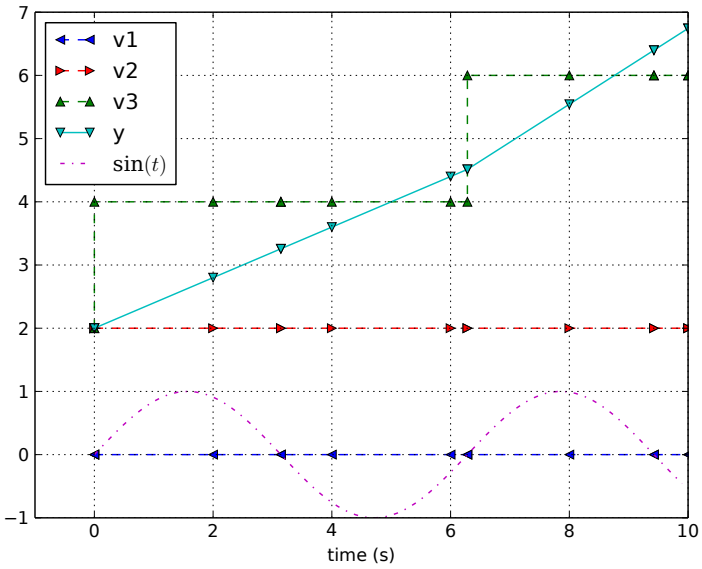


Figure 13: Plot for the event iteration example of Listing 6.

Listing 6 demonstrates several aspects of events and initialization in the Modelica language that can be observed in the plots of Figure 13:

- The `initial()` operator allows triggering of an event at initialization time. That event can be used to enable equations that are active during initialization.
- Events can be propagated by Boolean variables. Lines 13 and 14 use Boolean operators on Boolean event variables that are used as conditionals for if-expressions.
- Instantaneous equations that are activated using when-equations are not active during initialization, unless explicitly activated by using the `initial()` operator (line 21). Due to [Mod14, Section 8.6] it is not legal to replace `initial()` by `iev1` here.
- Line 22 shows a function call for writing simulation values to the standard output. Modelica functions are supposed to be "pure" mathematical functions, in the sense that they are side-effect free with respect to the internal Modelica simulation state [Mod14, Section 12.3]. Modelica functions that do not fulfil that property are supposed to be declared as `impure` functions. Since writing to standard output has no effect on the internal simulation state, the function used in this example is a "pure" function.
- The `initial equation` sections (line 27) allow stating purely algebraic constraints between variables that must hold during the initial time instant. Notice that instead of activating equations using `when initial()then` it is equally possible to activate them by putting them into an initial algorithm section.
- Further, we note that there is no event iteration during initialization. The event `ev1` (line 12) is active after initialization, hence `y = 2` after initialization.
- Assigning a fixed start value to a discrete variable `x(start=x_start,fixed=true)` is equal to declaring an initial equation `pre(x)=x_start;`.

- Assigning a fixed start value to a non-discrete variable `x(start=x_start,fixed=true)` is equal to declaring an initial equation `x=x_start;`.
- For further information on the initialization of Modelica models see [Mod14, p. 95].

2.6.7 Bouncing Ball (Reinitialization)

A famous model for a hybrid systems is the bouncing ball. A possible Modelica implementation for a ball with mass 1 kg that falls from an initial height of $p = 2\,\text{m}$ and an impact coefficient of 0.8 is given in Listing 7. When the ball hits the ground it changes its velocity discontinuously and bounces back. An UTP version of the bouncing ball model is provided in Example 4.1 on page 41.

Listing 7: Bouncing ball example.

```
1 model BouncingBall
2   Real p(start=2,fixed=true), v(start=0,fixed=true);
3 equation
4   der(v) = -9.81;
5   der(p) = v;
6   when p <= 0 then
7     reinit(v, -0.8*pre(v));
8   end when;
9 end BouncingBall;
```

The operator `reinit(x, expr)` (line 7) allows a discontinuous *reinitialization* of a continuous-time state variable `x` with a value computed from an expression `expr`. The operator can only be applied in the body of a when-clause.

The model is simulated in OpenModelica using the settings below (for the remaining settings, default settings are used):

startTime	0
stopTime	8
numberOfIntervals	80

The Variable `p` is plotted in Figure 14 for two time intervals. In the first interval $t = [0, 4]\,\text{s}$ the ball behaves as expected. However, at some point in the second domain the ball suddenly falls through the ground.

The reason for this is that `p` is smaller than zero when the event occurs and at some point during simulation the velocity `-0.8*v` is not large enough so that the ball will not fly above p=0. Consequently, no new event is triggered and the ball accelerates in the direction of gravity. This apparently simple model is in fact quite interesting since it exhibits *Zeno* behavior. The Zeno behaviour can be informally described as the system making an infinite number of jumps in a finite amount of time.

From a physical point of view one can argue that the model above is *wrong*, because the constraint of the ground is not described properly. However, a detailed ground model might slow down simulation dramatically and at least the model above can be changed easily so that it shows reasonable behavior by stating a condition that switches to another equation if the ball falls through the ground (see Listing 8).

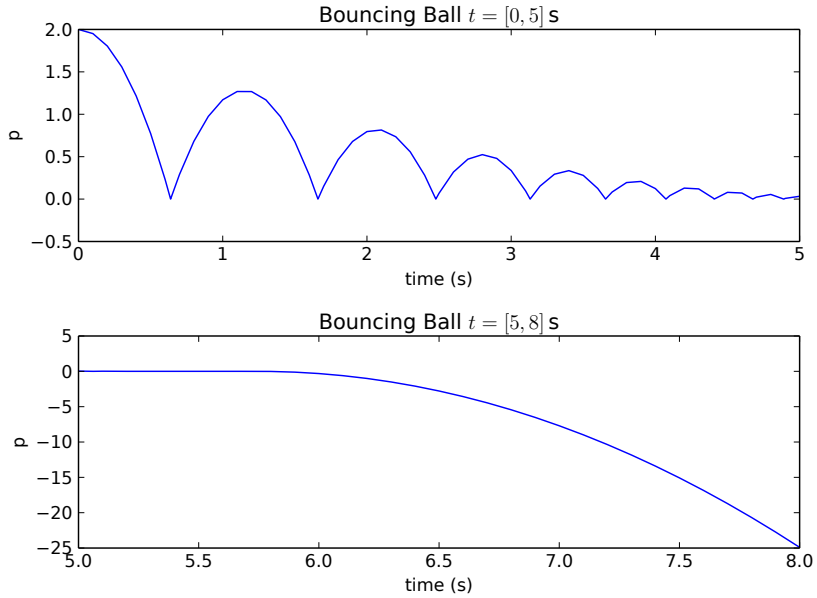


Figure 14: Plot for the bouncing ball example of Listing 7.

Listing 8: Improved bouncing ball example.

```
model ReasonableBouncingBall
  Real p(start=2,fixed=true), v(start=0,fixed=true);
  Boolean flying;
equation
  der(v) = if flying then -9.81 else 0;
  flying = not (p <= 0 and v <= 0);
  der(p) = v;
  when p <= 0 then
    reinit(v, -0.8*pre(v));
  end when;
end ReasonableBouncingBall;
```

2.6.8 Prioritizing event actions and event synchronization

A model featuring *prioritized event actions* and (un)*synchronized events* is shown in Listing 9.

Listing 9: Example for prioritized event actions and (un)synchronized events.

```
model WhenPriority "Event priority and event
   synchronization"
 import S = Modelica.Utilities.Streams;
 Integer x(start=0,fixed=true), y(start=-1,fixed=true),
         z(start=-2,fixed=true);
equation
  when pre(x) == 2 then
    S.print("A x: "+String(x));
    x = pre(x) + 3;
```

```
9     elsewhen pre(x) > 4 then
10      S.print("B x: "+String(x));
11      x = pre(x) + 1;
12    elsewhen pre(x) > 3 then
13      S.print("C x: "+String(x));
14      x = pre(x) + 2;
15    elsewhen time > 2 then
16      S.print("D x: "+String(x));
17      x = pre(x) + 2;
18    end when;
19    // No guarantee that event is triggered at
20    // exactly the same time instant as above
21    when time > 2 then
22      S.print("E y: "+String(y));
23      y = pre(x) + 1;
24    end when;
25    // "Synchronized" event,
26    // propagated through data-flow analysis of x
27    when x == 2 then
28      S.print("F z: "+String(z));
29      z = x + 1;
30    end when;
31    // It is an error to activate the following line
32    // when time > 3 then x = 0; end when;
33  end WhenPriority;
```

Using **when/elsewhen** constructs allows prioritizing event actions, where the declaration order of the different parts determines the priority (last **elsewhen** has lowest priority). This order ensures that the single-assignment rule is always fulfilled (if two events are triggered at the same time, the equations in the upper part take priority). Notice that it is an error if the same left-hand side (LHS) variable is assigned in two different when-equations, *e.g.*, if line 32 was activated that would result in an erroneous model (since in general it cannot be excluded that two different when-clauses can get activated at the same time, and thus violate the single-assignment rule).

The model is simulated in OpenModelica using the settings below (for the remaining settings, default settings are used):

startTime	0
stopTime	5
numberOfIntervals	5

The print statements allow tracing the order in which the when-clauses are activated:

```
D x: 2
E y: 1
F z: 3
A x: 5
B x: 6
```

It is notable that independent events are not guaranteed to be triggered at exactly the same time [Mod14, Section 8.5]. Hence, it is not guaranteed from the specification that the equations within the when-clauses in line 15 and 21 are synchronized (hence, there is no guarantee that they will be concurrently satisfied). To achieve guaranteed synchronization between when-clauses, it has to be explicitly modeled by propagating the event through data-flow dependencies and exploiting the *synchronous principle* that computation at an event takes no time (see [Fri14, Section 13.2.3 and 13.2.6.6]). This has been used to synchronize the when-clauses in line 15 and 27.

The Variables `x`, `y`, and `z` are plotted in Figure 15. The triangle markers within the graph depict a data entry for the respective variable within the generated result file.

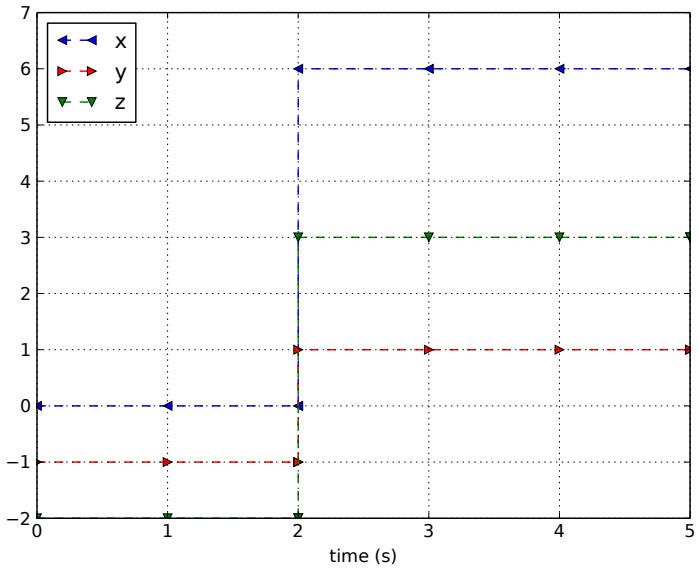


Figure 15: Plot for the example of prioritizing events from Listing 9.

2.6.9 Other notable examples

The purpose of the presented examples is to depict important aspects of the dynamic semantics of Modelica models. The examples are not supposed to cover the complete dynamic semantics of the Modelica language.

The following list provides a (non-exhaustive) selection of other notable examples from literature related to the semantics of Modelica models or hybrid systems in general:

- Fritzson's comprehensive Modelica book [Fri14] contains instructive examples covering a wide range of the Modelica language. In the context of this work, particularly Chapter 13 "Discrete Events and Hybrid and Embedded System Modeling" of this book is of great relevance.
- Otter et al. [OEM99] discuss hybrid modelling in Modelica based on several examples like sampled data systems (digital control applications), ideal electrical switches and mechanical friction.

- Recent work from Bliudze and Furic leverages ideas from [BBCP12] for giving a Zeno-free operational semantics for hybrid systems and coping with composition issues stemming from the use of idealized hybrid models using two demonstrative examples [BF14, Listing 1 "Bouncing Ball", Listing 3,4 "Double Fuse"].
- Lee and Zheng discuss an operational semantics for hybrid systems (using the concept of *super-dense time*) with instructive examples [LZ05, Section 2 "Sticking Masses", Figure 13 "bouncing ball"].
- The `reinit()` operator is known to be a problematic construct in the Modelica language that has to be used carefully (however, no better alternatives have been found, yet). Bourke et al. provide an example in [BBCP15, Slide 15]. Fritzson discusses the operator in [Fri14, Section 8.3.6].

2.6.10 Summary

The dynamic semantics of the Modelica language is complex. The main points raised in the examples are:

- Solving the IVP for DAE systems is in general much more involved than solving ODE systems. Index reduction methods play an important part in the solution (see the "Cartesian Pendulum" example in Section 2.6.2).
- Handling of discontinuous equations by relation triggered events (Section 2.6.3).
- Introducing instantaneous equations and related operators (Section 2.6.5).
- Event iterations (Section 2.6.6).
- Reinitialization of continuous-time state variables (Section 2.6.7).
- Prioritizing event actions (Section 2.6.8).
- Event synchronization (Section 2.6.8).

The examples illustrate key concepts of Modelica that can serve as a guidelines for later formalization efforts.

3 Unifying Theories of Programming

Unifying Theories of Programming [HH98] (UTP) is a mathematical framework for the specification and study of heterogeneous programming language semantics. We will use the UTP to give a denotational semantics to Modelica, as a well as several of the other languages in INTO-CPS. The UTP is based on the idea that any program or temporal model can be expressed as a logical predicate that describes how the program variables change over time. This idea of "programs-as-predicates" [Heh93] means that the duality of programs and specifications all but disappears as programs themselves are just a class of logical specification.

This powerful idea provides a strong basis for unification of heterogeneous languages and semantic models, since many different shapes of models can be given a uniform view. The

$$
\begin{array}{rcll}
P\,;\,Q & \triangleq & \exists \tilde{v}_0 \bullet (P[\tilde{v}_0/\tilde{v}'] \wedge Q[\tilde{v}_0/\tilde{v}]) & \text{for } \tilde{v} = \text{out}\alpha(P), \tilde{v}' = \text{in}\alpha(Q) \\
x :=_A v & \triangleq & x' = v \wedge y_1' = y_1 \wedge \cdots \wedge y_n' = y_n & \text{for } y_i \in \text{in}(A) \setminus \{x\} \\
P \sqcap Q & \triangleq & P \vee Q & \\
P \lhd b \rhd Q & \triangleq & (b \wedge P) \sqcap (\neg b \wedge Q) & \\
P^\star & \triangleq & \nu X \bullet (\mathit{II} \sqcap P\,;\,X) & \\
P^\omega & \triangleq & \mu X \bullet (P\,;\,X) &
\end{array}
$$

$$
\begin{array}{rcl}
\alpha(P;Q) & = & \text{in}\alpha(P) \cup \text{out}\alpha(Q) \\
\alpha(x :=_A v) & = & A \\
\alpha(P \lhd b \rhd Q) & = & \alpha(P) \cup \alpha(b) \cup \alpha(Q) \\
\alpha(P \sqcap Q) & = & \alpha(P) \cup \alpha(Q) \\
\alpha(P^\star) & = & \alpha(P^\omega) \;=\; \alpha(P)
\end{array}
$$

Table 5: UTP imperative operator predicate definitions and alphabets

UTP further allows that different semantic presentations, such as denotational, algebraic, axiomatic, and operational, can be formally linked, for example through suitable Galois connections. This ensures that consistency is maintained between semantics and that tools that implement the different semantic models can, nevertheless, be combined for multi-pronged analysis and verification of models.

The UTP has been mechanised in the Isabelle/HOL proof assistant [NWP02] in the form of *Isabelle/UTP* [FZW14]. This means that proofs of algebraic, refinement, operational, and other forms of laws can be machine checked, and also applied to verification of models through Isabelle's powerful proof automation [BBN11].

The UTP provides a language of alphabetised predicates: for a predicate P, the alphabet $\alpha(P)$ gives the variables over which the predicate ranges. The calculus provides the operators typical of first order logic, such as connectives $\wedge, \vee, \neg, \Rightarrow$ and quantification $\forall x.P, \exists x.P, [P]$, where $[P]$ represents the universal closure of P. UTP predicates are ordered by a refinement partial order $P \sqsubseteq Q$, that equates to universal closure of reverse implication $[Q \implies P]$. A particularly interesting subclass of such predicates are alphabetised relations, which describe a program in terms of its input variables $x, y \in \text{in}\alpha(P)$ and its (primed) output variables $x', y' \in \text{out}\alpha(P)$. Imperative programs can thus be given denotational semantics using the alphabetised relational calculus, whilst maintaining the predicative core.

Imperative programs can be described using relational operators such as sequential composition $P\,;\,Q$, if-then-else conditional $P \lhd b \rhd Q$, non-deterministic choice $\sqcap$, assignment $x :=_A v$ (for expression v and alphabet A), and skip II_A all of which are given predicative interpretations. For such imperative programs, the refinement operator $P \sqsubseteq Q$ corresponds to behavioural refinement, where the refined program Q is more deterministic than P. This also induces a complete lattice on programs, where ***true***, the most non-deterministic program represents the bottom of the lattice, and ***false***, the miraculous

program, is the top. Recursive and iterative constructions can then be specified using lattice and fixed point operators, such as $\bigsqcap$, $\mu X.P$, and the derived iteration operators $P^\star$ (finite iteration) and P^ω (infinite iteration).

All these operators are given a purely predicative interpretation, as described in the UTP book [HH98] and illustrated in table 5. Sequential composition of programs P, with output alphabet (outα) in variable vector $\tilde{v}'$, and Q, with input alphabet (inα) in vector $\tilde{v}$, specifies the existence of a vector of intermediate variables $\tilde{v}_0$ upon which both programs must agree. This agreement is specified by renaming, respectively, the output and input variables according to $\tilde{v}_0$, and then conjoining the results. Sequential composition thus feeds the output of the first program into the input of the second. Assignment updates the value of a variable with an expression, and leaves other variables unchanged. Non-deterministic choice and conditional are given a simple predicative interpretation. Finally the iteration operators are defined using fixed points.

More sophisticated language constructs can be expressed by further enriching the theory of alphabetised relations to create new UTP theories. A UTP theory consists of three things:

1. a set of *observational variables*;
2. a *signature*;
3. a set of *healthiness conditions*.

The observational variables record behavioural semantic information about a particular program. For example, we may have an observational variable for recording the current time called $clock : \mathbb{R}$, or we may have a variable for event traces called $tr : \mathbb{N} \to Event$ that records, at each discrete time instance, the event that occurred. These kinds of variables are distinguished from program variables in that they cannot be queried or manually altered during the running of a program. They purely exist to record logical information about a particular model. The signature then uses these operational variables to define the main operators of the target language.

The domain of observational variables can be restricted by the use of *healthiness conditions*. A healthiness condition is a kind of invariant on the behaviour of an observational variable. For example, it is intuitively the case that time only moves forward, and so a relational observation like $C \triangleq clock = 3 \wedge clock' = 1$ ought not to be possible, since time is observed moving backwards. We can restrict this kind of behaviour with an invariant $clock \leq clock'$. In the UTP such conditions are expressed as idempotent commuting functions, for example $HT(P) = P \wedge clock \leq clock'$, so that healthiness can be expressed as a fixed point: $P = HT(P)$. If we apply HT to our errant predicate C the predicate ***false*** will result. This represents a the miraculous program, which well describes the situation when time moves backwards. Clearly then C is not HT-healthy, and should be rejected from the theory signature.

UTP theories can be used to describe a domain useful for modelling particular problems – for instance, we can add further conditions to HT to provide a theory of real-time programs. UTP theories can also be composed to produce modelling domains that combine different language aspects. Put more simply, UTP theories provide the building blocks for a heterogeneous language's denotational semantics [FMW+14]. Such a denotational semantics provides the "gold standard" for the meaning of language constructs and can

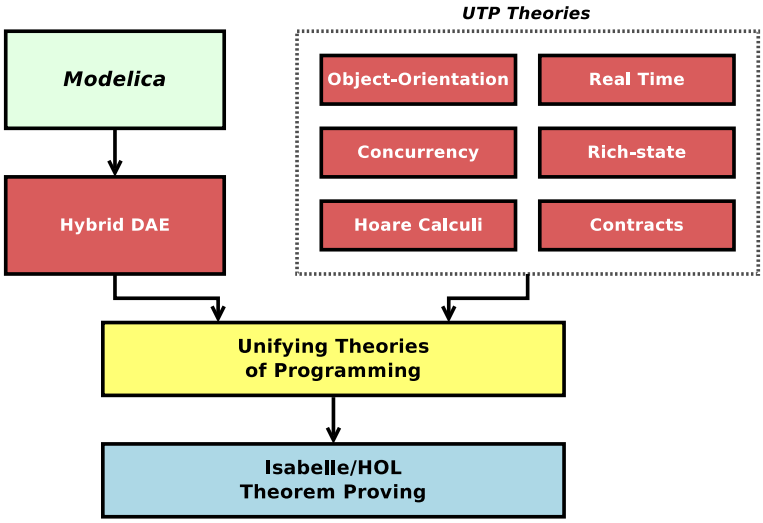


Figure 16: Modelica in the Unifying Theories of Programming

then be used to derive other presentations, such as operational. Thus for Modelica, we focus on a denotational semantics.

Disjoint heterogeneous domains can also be linked through *Galois connections* that describe the best approximations of a predicate in one theory in another. For example, in Modelica we could think about a Galois connection between the continuous time world and the discrete time world that applies a suitable sampling frequency. This would then allow us to apply verification techniques for discrete time processes present in model checking technologies like FDR3[8]. Thus the UTP can thought of as a *lingua franca* in which heterogeneous language semantics can be compared, linked, and ultimately unified.

In previous work we have used this approach to give a semantics to the *Circus* language [OCW07], a formal modelling language that elegantly combines support for rich-state modelling in the style of Z [WD96], with concurrency and communication in the style of CSP [Hoa85]. *Circus* has also been extended to encompass further programming paradigms including object-orientation [CSW05, SCS06] and real-time [WWC13, Woo14], all of which are underpinned by UTP theories. We will thus develop a UTP theory of continuous time and differential algebraic equations as a necessary building block to providing the semantics of Modelica, that we will combine with other theories already part of the UTP as illustrated in Figure 16. These theories will be combined with theories from parallel tasks to define our overall *lingua franca* language, INTO-CSP.

4 UTP Theory of Differential Equations

In this section we describe how we give an account of hybrid Differential Algebraic Equations (DAEs) in the Unifying Theories of Programming (UTP). As such, this is a preliminary theory and will continue to evolve through the life of INTO-CPS, particularly as in terms of the model of continuous time. UTP is, thus far, mainly concerned with description of behaviour of discrete systems by binary relations, and so we will need to

[8]FDR3 – The CSP refinement checker. https://www.cs.ox.ac.uk/projects/fdr/

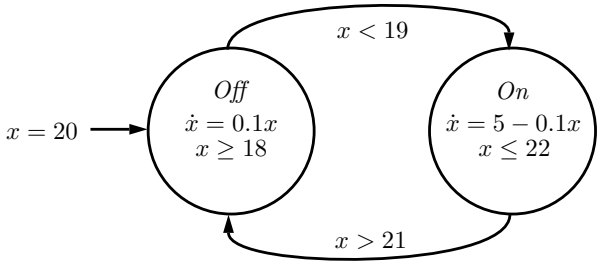


Figure 17: A simple thermostat hybrid automaton

$$P, Q ::= \ \mathbf{stop} \mid \mathbf{skip} \mid v := e \mid ch?x \mid ch!e \mid \langle F(\dot{s}, s) = 0 \,\&\, B \rangle \mid$$
$$P \,;\, Q \mid B \to P \mid P \trianglerighteq_d Q \mid P \trianglerighteq [\!]_{i \in I} (io_i \to Q_i) \mid P^\star$$

Table 6: Grammar of $\mathcal{HCSP}$ processes

introduce some additional notions and constructs to provide for continuous modelling. Our work is influenced by a number of keys works in the area of semantics for hybrid systems which we now briefly survey.

4.1 Formal approaches to hybrid systems

Our framework takes input from several main sources that we briefly now describe. The majority of the work on hybrid systems takes inspiration from Hybrid Automata [Hen96], an extension of finite state automata that also allow the specification of continuous behaviour. A hybrid automaton consists of a finite set of states that are each labelled by a system of ODEs, a state invariant, and initial conditions. An example of such a hybrid automaton is shown in figure 17 representing a simple thermostat, with variable x measuring the current temperature. The states (or "modes") are connected by transitions that are labelled with jump conditions and (optionally) events.

Whilst in a state the continuous variables evolve according to the system of ODEs and the given invariant; this is known as a *flow* as the variables values continuous flow. In the example there are two states *On* and *Off*, that are both annotated with an ODE and an invariant. When one of the jump conditions of an outgoing edge is satisfied the event is instantaneously executed, potentially resulting in a discontinuity, and the targeted hybrid state is activated. For the thermostat, when the temperature drops below 19 in state *Off* the transition can be triggered, causing a change of mode to *On*. Thus a hybrid automata is characterised by behaviour that both flows along a given trajectory (continuous behaviour) and also jumps (discrete behaviour). Hybrid automata are usually given a denotational semantics in terms of piecewise continuous functions [Hen96, EP07]. A piecewise continuous function is continuous except for in a finite number of places where a discontinuous jump can occur. These jumps represent the instantaneous update of state variables.

Verification of hybrid systems was made possible through the seminal work of Platzer [Pla08, Pla10b]. This work develops a logic called *differential dynamic logic* ($\mathsf{d}\mathcal{L}$), an extension

of dynamic logic that allows one to specify invariants over both discrete and continuous variables. Hybrid systems are modelled using a language of *hybrid programs*, that combines the usual operators of an imperative language with continuous behaviour specified by differential equations. Hybrid programs are equipped with a relational semantics, and a proof calculus for $\mathsf{d}\mathcal{L}$ is created that allows reasoning about hybrid programs. A critical design choice made in the creation of the $\mathsf{d}\mathcal{L}$ semantics seems to be the use of a semantic model that does not explicitly represent trajectories of continuous variables [Pla07]. In $\mathsf{d}\mathcal{L}$ a program simply denoted by a a set of input-output variable pairs, as is usual for relational program semantics. This relatively coarse semantics effectively circumvents problems associated with super-dense time models, for example by allowing variable assignment to be instantaneous, though this may make it inadequate for specifying real-time properties.

A further extension to $\mathsf{d}\mathcal{L}$ is also created called *differential-algebraic dynamic logic* [Pla10a] (DAL), that allows one to reason about differential equations, without the need for explicit solutions, through the use of *differential invariants*. An implementation of DAL called *KeYmaera* [Pla10b] allows the automated verification of systems modelled as hybrid programs. We base our own notion of hybrid programs around Platzer's work. A difference though is our focus on UTP denotational semantics, the use of pre-emption to drive the interaction between discrete and continuous, and our eventual desire to integrate concurrency into our model.

Hybrid CSP [He94, ZJR96] ($\mathcal{HCSP}$) is an extension of Hoare's process calculus CSP [Hoa85] that adds support for continuous variables as described by differential equations, in a similar manner to hybrid automata. This work enables algebraic modelling of hybrid concurrent systems in contrast to Platzer's work [Pla10b] which allows only the specification of sequential hybrid systems. The initial paper on $\mathcal{HCSP}$ [He94] extends CSP with continuous variables whose behaviour is described by differential equations of the form $\mathcal{F}(\dot{s}, s) = 0$. Interaction between the discrete and continuous in $\mathcal{HCSP}$ can take the form of pre-emption conditions on continuous variables, timeouts, and interruption of a continuous evolution through CSP events. This new language is equipped with a denotational semantics that is presented in a predicative style similar to the UTP [HH98].

Further work on $\mathcal{HCSP}$ [ZJR96] enriched the language to allow explicit interaction between discrete and continuous variables. This is achieved through a novel denotational semantics in terms of the extended duration calculus [ZRH93] which treats variables as piecewise continuous functions. This allows a more precise semantics for operators like pre-emption that are defined in terms of suitable variable limits. A Hoare logic for this calculus was developed in [LLQ+10] through the adoption of Platzer's differential invariants, along with an operational semantics. This later work uses a simplified version of $\mathcal{HCSP}$ [ZJR96], a grammar of which is shown in Table 6.

An associated theorem prover for $\mathcal{HCSP}$ called HHL Prover [ZZWF15] based in Isabelle/HOL (extended with axioms) has also been developed and applied to verification of Simulink diagrams through a mapping from Simulink/Stateflow into $\mathcal{HCSP}$ [ZYZ+14]. The suite of tools for $\mathcal{HCSP}$ (called "MARS") can be obtained from the website of Naijun Zhan[9]. More recently, the original author [He94] has undertaken to explore the fundamentals of hybrid system modelling purely relational setting, again similar to the UTP. This work has produced a prototype language called the *Hybrid Relational Modelling*

[9]Naijun Zhan's Homepage. http://lcs.ios.ac.cn/~znj/

Language [He15] (HRML), which draws on $\mathcal{HCSP}$ though emphasises signals rather than CSP's events as the main communication abstraction. Our work is heavily influenced by $\mathcal{HCSP}$, though we initially focus on the sequential aspects of hybrid systems, and thus concentrate on just the a few of the $\mathcal{HCSP}$ operators.

We also take substantial input from the Duration Calculus [ZHR91] in this work, a calculus designed to describe logical invariants over continuous domains for the purposes of verifying real-time systems. Duration calculus is an interval temporal logic [Mos85], meaning that specifications range over a particular non-empty period of time. For example, we can write the interval $\lceil x^2 > 7 \rceil$ which specifies all possible intervals of time over which $x^2 > 7$ holds. This could for instance be used to specify an invariant over the execution of a program. The chop operator $P \bullet Q$ specifies that an interval may be broken into two subsequent intervals, over which P and then Q hold respectively.

Duration calculus can also be used to give an account to typical modalities of temporal logics like LTL [Pne77], and is thus well suited to specification. Duration calculus has been extended several times, notably to provide a semantics for hybrid real-time systems modelling [ZRH93] that is then used to give the semantics to $\mathcal{HCSP}$ [ZJR96]. It is further extended to support super-dense computation [ZH96, ZGZ99], so that instantaneous discrete changes can be mapped to the neighbourhood of a given time point. This, for instance, allows the retention of identities like

$$x := x + 1 \,;\, x := x + 2 \quad = \quad x := x + 3$$

from the laws of programming [HH98], as they demonstrate through giving a semantics to a simplified $\mathcal{HCSP}$ like language [ZH96]. Nevertheless, assignment must take non-zero time in this model and it is thus not clear if the model is truly super-dense.

4.2 Continuous time theory domain

We now proceed to describe our theory of differential equations, in the form of a calculus for modelling sequential hybrid processes. Our initial theory possesses a rich semantic model for continuous time, which may need to be refined in the future. We thus focus in our initial work on the signature of this theory, leaving the definition of healthiness conditions to future work. DAEs, as we have said, are used to describe continuously evolving dynamic behaviour of a system. Thus, in the context of the UTP, we must first introduce a theory of continuous time processes to provide this kind of theoretical framework. A key part of this is the embedding of *trajectories* into alphabetised predicates that show how continuous variables evolve over a given interval. These intervals will be used to divide up the trace of a system into piecewise continuous regions.

Our theory of continuous time introduces observational variables $time, time' : \mathbb{R}$ that define the start and end time of the current computation interval. We also introduce the expression ℓ to denote the duration of the current interval, i.e. $\ell \triangleq time' - time$. UTP divides the alphabet of a predicate P into input variables $x, y, z \in \mathrm{in}\alpha(P)$ and output variables $x', y', z' \in \mathrm{out}\alpha(P)$. Inspired by [He15] we add a further subdivision $\underline{x}, \underline{y}, \underline{z} \in \mathrm{con}\alpha(P)$, the set of continuous variables that is orthogonal to discrete variables, that is $\mathrm{con}\alpha(P) \cap (\mathrm{in}\alpha(P) \cup \mathrm{in}\alpha(Q)) = \emptyset$. The elements of $\mathrm{con}\alpha(P)$ are the variables to be used in differential equations and other continuous constructs. We assume that

all variables consist of a name, type, and decoration. For example the name in the variables x, x', and $\underline{x}$ is the same – x – but the decorations differ. We introduce the distinguished continuous variable $\underline{t}$ that denotes the current instant in an algebraic or differential equation. An alphabetised predicate P whose alphabet can be so partitioned, i.e. $\alpha(P) = \text{in}\alpha(P) \cup \text{out}\alpha(P) \cup \text{con}\alpha(P)$, is called a *hybrid relation.*

We also propose two healthiness conditions for hybrid relations:

Definition 4.1. Hybrid relation healthiness conditions

$$\mathsf{HCT1}(P) \triangleq P \wedge time \leq time' \tag{11}$$

$$\mathsf{HCT2}(P) \triangleq P \wedge \left(\bigwedge_{\underline{v} \in \text{con}\alpha(P)} \bullet v = \underline{v}(time) \wedge v' = \underline{v}(time') \right) \tag{12}$$

HCT1 states that time may only ever go forward, as should be the case, and thus an interval of time is always defined for any relation. HCT2 states the intuition explained above – that every continuous variable has two corresponding discrete variables measuring the variable before and after the interval. The latter healthiness condition is satisfied only by constructs that produce continuous evolution (such as DAEs).

Continuous variables come in two varieties which will allow us to talk respectively about both a particular instant and over the whole time continuum:

- *flat variables* – these are regular variables of any type;
- *lifted variables* – these are variables of type $\mathbb{R} \to T$ and define variables values in the trajectory.

In our calculus we assume that each lifted continuous variable $\underline{x} : \mathbb{R} \to T$ is accompanied by before and after variables with the same name – $x, x' : T$ – that give the values at the beginning and end of the current interval. This allows us to use the standard operators of relational calculus for manipulating continuous variables by preserving discrete copies. We introduce the following @ operator borrowed from [Fid99] that lifts a predicate in only flat variables to one in lifted variables:

Definition 4.2. Continuous variable lifting

$$P \mathbin{@} \tau \triangleq P \dagger \{\underline{x} \mapsto \underline{x}(\tau) \mid \underline{x} \in \text{con}\alpha(P) \setminus \{\underline{t}\}\}$$

The dagger (†) operator is a nominal substitution operator. It applies the given function, which maps variables to expressions, as a substitution to the given predicate. We construct a substitution that maps every flat continuous variable (other than the distinguished time variable $\underline{t}$) to a corresponding variable lifted over the time domain. The effect of this is to state that the predicate holds for values of continuous variables at a particular instant τ, a variable which is potentially free in P. Each flat continuous variable $\underline{x} : T$ is thus transformed to have time-dependent function $\underline{x} : \mathbb{R} \to T$ type.

4.3 Hybrid relational calculus

We next begin to define the operators of our hybrid calculus, which is effectively the imperative subset of $\mathcal{HCSP}$ [ZJR96] (which is itself similar to Platzer's hybrid programming language [Pla10b]), but extended with a version of the interval operator [ZHR91]

$$P, Q ::= \; P \,;\, Q \mid P \lhd b \rhd Q \mid x := e \mid P^\star \mid P^\omega \mid$$
$$\lceil P \rceil \mid \langle\, F_n \,|\, b \,\rangle \mid P \,[\, b \,]\, Q \mid P \trianglerighteq_d Q$$

Table 7: Signature of hybrid programs

that provides with a continuous specification statement. The signature of our theory is given in 7. It consists of the standard operators of the alphabetised relational calculus together with operators to specify intervals $\lceil P \rceil$, differential algebraic equations $\langle\, F_n \,|\, b \,\rangle$, pre-emption $P \,[\, b \,]\, Q$, and $timeout P \trianglerighteq_d Q$.

Using this calculus, we can describe a simple example, the bouncing ball as modelled in Modelica in section 2.6.7:

Example 4.1. *Bouncing ball in hybrid relational calculus*

$$p, v := 2, 0 \;;\; \Big(\Big\langle\, \underline{\dot{p}} = \underline{v}; \underline{\dot{v}} = -9.81 \,\Big\rangle \Big[\, \underline{p} \leq 0 \,\Big]\, v := -v * .8\Big)^\omega$$

The hybrid program has two continuous variables: $\underline{p}$ representing the position (in meters) of the ball above the ground, and $\underline{v}$ describing the velocity of the ball. Initially we set these two variable to 2 and 0 respectively, and then initiate the system of ODEs. This states that the derivative of $\underline{\dot{p}}$ is $\underline{v}$, and the derivative of $\underline{v}$ is -9.81, i.e. the earth's gravitational acceleration. The system is allowed to evolve until $\underline{p} \leq 0$, i.e. the ball impacts the ground. At this point a discrete command is executed that assigns $-v * .8$ to v; the velocity is reversed with a dampening factor. The system then infinitely iterates, allowing the system dynamics to continue evolving but with new initial values. Moreover, as in the Modelica example this bouncing ball exhibits Zeno effects.

4.4 Denotational semantics

We note that many of the standard operators of the alphabetised relational calculus (imperative programs) retain their standard definitions in this setting as illustrated previously in table 5, but over the expanded alphabet. Indeed, it is easy to see that an alphabetised relation is simply a hybrid relation with the degenerate alphabet $con\alpha(P) = \emptyset$. For continuous variables sequential composition behaves like conjunction. In particular, if we have $P \,;\, Q$, with P and Q representing evolutions over disjoint intervals then their sequential composition combines the corresponding trajectories when they agree on variable valuations. Put another way, the final condition P also defines the initial condition for Q.

Similarly, other operators like the Kleene star and omega iteration operators $P^\star$ and P^ω, being defined solely in terms of sequential composition, disjunction (internal choice), $\sqcap$, and fixed point operators, also remain valid in this context. This observation means that we already have the core operators of an imperative programming language at our disposal. We will prove that these core operators satisfy our two healthiness conditions in Isabelle in section 5.

$$\lceil true \rceil = \ell > 0 \quad (13)$$
$$\lceil false \rceil = \textbf{\textit{false}} \quad (14)$$
$$\lceil P \wedge Q \rceil = \lceil P \rceil \wedge \lceil Q \rceil \quad (15)$$
$$\lceil P \vee Q \rceil \sqsubseteq \lceil P \rceil \vee \lceil Q \rceil \quad (16)$$
$$\lceil P \rceil \sqsubseteq \lceil P \rceil \,;\, \lceil P \rceil \quad (17)$$

Table 8: Algebraic laws of durations

We will now give the denotational semantics for each of the additional operators of our calculus. First we define the interval operator from duration calculus [ZHR91]:

Definition 4.3. Interval operator

$$\begin{aligned} \lceil P \rceil \triangleq\ & \ell > 0 \Rightarrow \\ & (\forall \underline{t} \in [time, time') \bullet P \,@\, \underline{t} \\ & \wedge \bigwedge_{\underline{v} \in \mathrm{con}\alpha(P)} v = \underline{v}(time) \wedge v' = \underline{v}(time')) \end{aligned}$$

The interval operator states that P holds at every instant over all non-empty right-open intervals from *time* to *time'*. It can be considered analogous to a continuous specification statement. All continuous variables are paired with a discrete variable at the start and end of the interval, as required by ***HCT2***. Intervals satisfy a number of standard laws of duration calculus illustrated in table 8, which we will also prove in section 5.

We next introduce the following operator, adapted from $\mathcal{HCSP}$ [ZJR96, LLQ+10], to describe the evolution of a system of DAEs.

Definition 4.4. Differential Algebraic Equation system

$$\begin{aligned} \langle\, \dot{\underline{v}}_1 = f_1; \cdots; \dot{\underline{v}}_n = f_n \mid B \,\rangle \triangleq\ & \lceil (\forall 1 \leq i \leq n \bullet \dot{\underline{v}}_i(\underline{t}) = f_i(\underline{t}, \underline{v}_1(\underline{t}), \cdots, \underline{v}_n(\underline{t}))) \wedge B \rceil \\ & \wedge \mathit{II}_{A \setminus \{v_1 \cdots v_n\}} \end{aligned}$$

A DAE ($\langle\, F_n \mid B \,\rangle$) consists of a set of n ODEs characterised by functions $f_i : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}$ each of which defines the derivative of variable $\underline{v}_i$ in terms of the independent time variable $\underline{t}$ and the n dependent variables. It also contains an algebraic constraint B that must be invariant for any solution and does not refer to derivatives. When B is predicate *true* we will simply abbreviate to $\langle\, F_n \,\rangle$. We can, for example, use the DAE operator to describe the behaviour of the Cartesian pendulum from section 2.6.2 (assuming the prior definition of appropriate constants):

$$\left\langle\, \dot{x} = u; \dot{u} = \lambda \cdot x; \dot{y} = v; \dot{v} = \lambda \cdot y - g \;\middle|\; x^2 + y^2 = l^2 \,\right\rangle$$

The behaviour of a DAE is given in terms of the interval operator which contains an invariant showing how the continuous variables evolve with respect to the present time instant $\underline{t}$. At the initial time (*time*) each continuous variable $\underline{v}_i$ of the system is assigned the value of the corresponding discrete input variable v_i, which provides the DAE with

an initial value problem. The system is then allowed to evolve from this point in the interval between *time* and *time*′ according to the DAEs. At the end of the interval, the corresponding output discrete variables are assigned. During the evolution all discrete variables and unconstrained continuous variables are held constant, hence the conjoined $\text{I\!I}$.

We also define the following pre-emption operator, whose definition is also taken from $\mathcal{HCSP}$.

Definition 4.5. Pre-emption operator

$$P\,[\,B\,]\,Q \triangleq (Q \lhd B \,@\, time \rhd (P \wedge \lceil \neg B \rceil)) \vee ((\lceil \neg B \rceil \wedge B \,@\, time' \wedge P)\,;\,Q)$$

Intuitively, P is a continuous process that evolves until predicate B is satisfied, at which point Q is activated. This operator can, for example, be used to capture the idea of events from Modelica. The semantics is defined as a disjunction of two predicates. The first predicate states that if B holds in the initial state of *time* then Q is activated immediately. Otherwise, P is activated and can evolve while ever B remains false (potentially indefinitely). The second predicate states that $\neg B$ holds on the interval $[time, time')$ until instant *time*′ when B switches to a true valuation, during which P is executing. Following this, P is terminated and Q is activated.

Assignment to variables can introduce discontinuities when applied to the discrete partners of continuous variables. When several assignments take place this can lead to a sequence of variable valuations at a single time instant. For example, a naive implementation of an instantaneous $x := v$ (for continuous variable x) might be:

$$x := v \triangleq x' = v \wedge \underline{x}(time') = v \wedge time' = time$$

That is, we assign the value to both the continuous and discrete variable at *time*′, whilst ensuring that no time is expended. However, it is clear to see that $x := 1\ ;\ x := 2$ will yield a contradiction, since x can have two values at *time*′ $(1 = 2)$ which is a contradiction, and thus the miraculous process ***false*** will result. Several implementations of assignment are possible that overcome this issue depending on the underlying model. We could, for example, extend our variable type to be explicitly super-dense: $x : \mathbb{R} \times \mathbb{N} \to T$, so that at each instant x can possess several values. However, we wish to retain a simpler model for now, and so we instead we define a simple form of timed assignment, which is also adopted from $\mathcal{HCSP}$, that must expend a non-zero amount of time.

Definition 4.6. Timed assignment

$$(x :=_A v)_{\leq d} \triangleq 0 < \ell \leq d \wedge x' = v \wedge (\forall y \in \text{in}(A) \setminus \{x\} \bullet y' = y)$$

This operator assigns the expression v to the variable x in no more than d time units. Finally we also define the following timeout operator:

Definition 4.7. Timeout operator

$$P \trianglerighteq_d Q \triangleq (P \wedge (\ell \leq d)) \vee ((P \wedge (\ell \leq d))\,;\,Q)$$

This operator behaves like P until d time units have elapsed, at which point Q is activated.

5 Mechanisation

In this section we describe a partial implementation of the theory from Section 4 in the Isabelle/HOL theorem prover [NWP02] presented using the latter's automated document preparation system. The contents of this section has therefore passed Isabelle's automatic checks and the proofs are valid. We will implement most of the operators of our UTP theory, though their presentation differs slightly for technical reasons. Our implementation of the hybrid calculus builds on Isabelle's existing libraries for real analysis and thus we first provide a brief survey of these.

5.1 Mechanised Real Analysis

Real analysis has been implemented in various theorem provers, including Coq, Isabelle, Mizar, and PVS. A recent survey of these mechanisations [BLM15] provides a detailed comparison of the state-of-the-art. All these implementations provide exact real arithmetic and do not rely on approximations. Moreover, the majority of them (Isabelle included) are introduced contructively rather than by axiomatisation. We chose to use Isabelle as it is most familiar to us from our work with the UTP, it provides powerful automated reasoning facilities, and its implementation of analysis is at least comparable with other implementations. Real numbers were first constructed in Isabelle using Dedekind cuts [Fle00], that is via a partitioning of the rational numbers ($\mathbb{Q}$) into two sets – those strictly less than and greater the given real-number. A more recent implementation in the Isabelle standard library uses Cauchy sequences, whereby a real number n is denoted as an infinite sequence of rational numbers converging towards n. Isabelle also provides the basis for non-standard analysis in an implementation of the hyperreals [Fle00]; that is $\mathbb{R}$ extended with infinitesimals. For now, we focus on the standard analysis libraries in Isabelle since our semantics requires only standard reals.

Calculus and multivariate analysis was first mechanised in an impressive HOL development by John Harrison [Har98, Har05], and then later adapted to Isabelle [Fle00]. This implementation of analysis permits differentiation over real vector spaces through the provision of a function $(f \textit{ has-vector-derivative } f')F$ that defines when a particular function f' is a solution to a system of differential equations f over a given range F. For example, we can express and prove the following derivative:

```
lemma power2-deriv:
  fixes t :: real
  shows ((λ t. t²) has-vector-derivative (2 * t)) (at t within {0 ..< n})
proof −
  have ((λt. t * t) has-vector-derivative (t * 1 + 1 * t)) (at t within {0 ..< n})
    by (rule derivative-intros)+
  thus ?thesis
    by (simp add: power2-eq-square)
qed
```

This shows that the equation t^2 (for independent variable t) has solution $2t$, within the range $[0..n)$, a fact of elementary calculus. Building on Harrison's work, Immler later mechanised an Isabelle library for analysis of ODEs [IH12, Imm14], including the specification of initial value problems and an implementation of Euler's method for numerical

integration. The latter is implemented through a principled conversion of the constructed real numbers to an Isabelle implementation of floating point numbers. All this taken together provides an excellent foundation for reasoning about continuous models in Isabelle.

5.2 Theory of hybrid relations

Our implementation presented in this section is a shallow embedding of our UTP theory in Isabelle which acts as a proof of concept of the theory in Section 4, and could be further developed into a theorem prover for hybrid systems. Our encoding is based on Isabelle's implementation of binary relations, for which a large library of proven algebraic laws already exists. However, this means that we need to embed the (non-relational) continuous trajectories into this model. We will do this by constraining relations so that the variable storing the trajectory remains constant throughout the relation. We first introduce types to represent hybrid states and programs.

record $('a, 'c)$ *hyst* =
  time :: *real*
  disc :: $'a \times 'c$
  cont :: *real* $\Rightarrow$ $'c$

A hybrid state $('a, 'c)$ *hyst* is a type with type two parameters: $'a$ that describes the discrete state of a program, represented in inα and outα, and $'c$ that describes the continuous state of a program, represented in conα. Unlike in our UTP theory of Section 4 we have to split the variables in this way since the shallow embedding does not allow variables as first class citizens. The discrete state can, for example, be an Isabelle record constituting the discrete program variables, and the continuous state will usually be a suitable n-ary vector of real numbers, $\mathbb{R}^n$. The hybrid state record has three fields representing respectively, a real-valued time bound, a snapshot of the discrete and continuous state (represented as a product), and a trajectory over the continuous state (represented as a real function).

We then use this type to encode our two healthiness conditions:

abbreviation *isHCT1* $\equiv$ $(\lambda\ (s,\ s').\ time\ s \leq time\ s')$
abbreviation *isHCT2* $\equiv$ $(\lambda\ (s,\ s').\ snd\ (disc\ s) = cont\ s\ (time\ s)$
  $\wedge\ snd\ (disc\ s') = cont\ s\ (time\ s'))$

For simplicity of presentation we encode the healthiness conditions not as idempotent functions but as predicates. The healthiness conditions are defined with respect to the before state and after states, s and s'. ***HCT1*** states that time cannot move in reverse as the state transitions advance, and follows the same structure as in section 4. ***HCT2*** states that the discrete variables must match the continuous variables at the start and end of the given interval. We encode this by stating that the second component of the discrete state snapshot (i.e. the continuous variables) should be the same as the trajectory values at the beginning and end of the interval. We now use our hybrid state type and healthiness conditions to encode hybrid relations.

typedef $('a, 'c)$ *hrel* =
  $\{(\ R :: ('a, 'c)\ hyst\ rel).\ \forall\ (s,\ s') \in R.\ cont\ s = cont\ s' \wedge isHCT1(s,s') \wedge isHCT2(s,s')\}$
  by (*auto*)

The hybrid relation type (*'a*, *'c*) *hrel* is a subset of the binary relations (*'t rel*) over the hybrid state type, such that the trajectory function is unchanged and the two healthiness conditions are respected. We define the type using the **typedef** command to be the subset of the binary relations over the hybrid state type ((*'a*, *'c*) *hyst rel*) that also respect the constraints. These constraints are defined by quantifying over all members (s, s') of the given relation R and imposing three constraint formulae. Specifically, we require that the continuum function remains unchanged $cont\ s = cont\ s'$, effectively making it non-relational, and that the two healthiness conditions are satisfied.

5.3 Operators

With our type of hybrid relations defined, we begin to define the operators of our language. Most of the operators are defined as "lifted" definitions using Huffman's lifting package [HK13]. A lifted definition defines a new constant in terms of the base type of the underlying type definition (in this case binary relations), and requires that any additional constraints imposed are respected by the value. The latter is supported by a proof that must accompany the definition. Thus the definitions in this section also encapsulate certification of their closure under our healthiness conditions.

We first encode the interval operator $\lceil P \rceil$ in this way using the **lift-definition** command:

lift-definition *hInt* :: (*real* × *'c* ⇒ *bool*) ⇒ (*'a*, *'c*) *hrel* ($\lceil\text{-}\rceil_H$) **is**
λ *P*. {(*s*, *s'*). *time s* < *time s'*
∧ (∀ *t* ∈ {*time s* ..< *time s'*}. *P* (*t*, *cont s t*))
∧ *cont s* = *cont s'*
∧ *isHCT2*(*s*,*s'*)}
by *auto*

This lifted definition take a type for the new construct, and a definition ("**is**") in terms of the base type, which in this case is (*'a*, *'c*) *hyst rel*. Since the interval operator has a function type, we must introduce a λ-abstraction that introduces the predicate parameter P. Since relations are just sets of pairs, we use the set comprehension notation to quantify each element pair (s, s') and then impose four constraints on these elements that characterise the semantics of an interval. Respectively they impose that the interval be non-empty, that at each point t in the interval the predicate P is satisfied, that the trajectory remains constant and that ***HCT2*** is satisfied. ***HCT1*** is automatically satisfied by the first constraint. Definitions can also specify pretty-printer syntax, and in this case we allow intervals to be written as $\lceil P \rceil_H$.

The proof of closure under of our type constraints for *hInt* is simply obtained by the application of the **auto** tactic. The interval function *hInt* thus takes a predicate over the current time (*real*) and continuous state *'c*, and yields a continuous relation with all intervals over which the predicate is invariant. It also imposes ***HCT2*** ensuring that the corresponding discrete variables mirror the continuum at the beginning and end of the interval.

We also implement a special construct to constrain the domain of a duration:

lift-definition *hDur* :: (*real* ⇒ *bool*) ⇒ (*'a*, *'c*) *hrel* ($\mathcal{L}[\text{-}]_H$) **is**
λ *P*. {(*s*, *s'*). *P* (*time s'* − *time s*)

$\wedge$ *cont* s = *cont* s'
$\wedge$ *isHCT1*(s, s') $\wedge$ *isHCT2*$(s, s')\}$
by *auto*

This construct takes a predicate over a real value time instant and produces all intervals whose duration satisfies the constraint. We use this construct to encode formula like $\ell > 0$ the like of which is often used in duration calculus. Next, we define some simple logical operators for hybrid relations.

lift-definition *hTrue* :: $('a, 'c)$ *hrel* $(true_H)$ **is**
$\{(s, s').$ *cont* s = *cont* $s' \wedge$ *isHCT1*$(s,s') \wedge$ *isHCT2*$(s,s')\}$
by *auto*

The $true_H$ $(true_H)$ operator is the bottom of the refinement lattice. It represents the most non-deterministic hybrid relation, requiring only that the continuum remains constant and the two healthiness conditions are satisfied.

lift-definition *hFalse* :: $('a, 'c)$ *hrel* $(false_H)$ **is** $\{\}$
by *auto*

The $false_H$ $(false_H)$ operator is the top of the refinement lattice and has no possible observations; it is obtained by lifting the empty relation.

lift-definition *hDisj* :: $('a, 'b)$ *hrel* $\Rightarrow$ $('a, 'b)$ *hrel* $\Rightarrow$ $('a, 'b)$ *hrel* (**infixl** $\vee_H$ *60*)
is *op* $\cup$ **by** *auto*

lift-definition *hConj* :: $('a, 'b)$ *hrel* $\Rightarrow$ $('a, 'b)$ *hrel* $\Rightarrow$ $('a, 'b)$ *hrel* (**infixl** $\wedge_H$ *60*)
is *op* $\cap$ **by** *auto*

We also define the disjunction $(P \vee_H Q)$ and conjunction $(P \wedge_H Q)$ operators that are obtained simply by lifting union and intersection on relations.

abbreviation *hChoice* :: $('a, 'b)$ *hrel* $\Rightarrow$ $('a, 'b)$ *hrel* $\Rightarrow$ $('a, 'b)$ *hrel* (**infixl** $\sqcap_H$ *60*)
where $P \sqcap_H Q \equiv P \vee_H Q$

lift-definition *hCond* ::
$('a, 'c)$ *hrel* $\Rightarrow$ $('a, 'c)$ *hrel* $\Rightarrow$ $('a, 'c)$ *hrel* $\Rightarrow$ $('a, 'c)$ *hrel* (**infixl** $\lhd$ - $\rhd$ *65*)
is $\lambda\ P\ b\ Q.\ ((b \cap P) \cup (-\ b \cap Q))$
by *auto*

We obtain the non-deterministic choice operator $P \sqcap_H Q$ simply as an abbreviation to disjunction. We also define the if-then-else conditional operator, $P \lhd b \rhd Q$ which behaves as P if b is true and Q otherwise. Next we define the operators of relations.

lift-definition
hSeq :: $('a, 'c)$ *hrel* $\Rightarrow$ $('a, 'c)$ *hrel* $\Rightarrow$ $('a, 'c)$ *hrel* (**infixl** $;_H$ *85*)
is *relcomp* **using** *order.trans* **by** *fastforce*

The sequential composition operator $op\ ;_H (P\ ;_H Q)$ is obtained by lifting the standard relational composition function $op\ O$. The proof is little more involved as we need to make use of order transitivity to show that $time \leq time'$ in terms of the two sequential components. The proof is still, however, automated using the **fastforce** tactic.

lift-definition *hSkip* :: $('a, 'c)$ *hrel* (II_H) **is**

$\{(s,\ s').\ cont\ s = cont\ s' \wedge disc\ s = disc\ s' \wedge time\ s = time\ s' \wedge isHCT2(s,s')\}$
by *auto*

The II_H (II_H) operator is the unit of sequential composition. Unlike sequential composition we cannot lift the relational identity Id as this does not respect ***HCT2***. Instead, we identify each of the components of the skip and impose ***HCT2*** manually.

lift-definition *hAssign* :: $('a \times 'c \Rightarrow 'a \times 'c) \Rightarrow ('a,\ 'c)\ hrel$ **is**
$\lambda\ f.\ \{(s,\ s').\ disc\ s' = f\ (disc\ s) \wedge time\ s' > time\ s \wedge cont\ s' = cont\ s \wedge isHCT2(s,\ s')\}$
by *auto*

The *hAssign* operator is implemented using an update function over the discrete and continuous state. The discrete state afterwards is set to the function applied to the discrete state before. Assignment in this setting is not instantaneous – it must expend a non-zero amount of time. Next we define the operator for differential equations.

lift-definition *hODE* ::
$(real \times 'c :: real\text{-}normed\text{-}vector \Rightarrow 'c) \Rightarrow ('a,\ 'c)\ hrel\ (\langle\text{-}\rangle_H)$
is $\lambda\ f'.\ \{(s,\ s').\ (\forall\ \tau \in \{time\ s\ ..<\ time\ s'\}.$
$((cont\ s)\ has\text{-}vector\text{-}derivative\ f'\ (\tau,\ cont\ s\ \tau))$
$(at\ \tau\ within\ \{time\ s\ ..<\ time\ s'\}))$
$\wedge\ cont\ s = cont\ s' \wedge isHCT1(s,s') \wedge isHCT2(s,s')\}$
by *auto*

In our current implementation we are limited to expression of ODEs only. This is because algebraic variables, which can change continuously but do not have derivatives, would require a further type parameter, which we wish to avoid for now.

The *hODE* operator takes a function representing an ordinary differential equation and produces the relation that accordingly evolves over all possible intervals. It is implemented using the *has-vector-derivative* function from Isabelle's implementation of analysis [Fle00]. The definition states that, given an ODE function f', at each instant τ within the time interval, the solution to f' at τ for said variable valuations is given by the function $cont\ s$, that is, the state trajectory, over the interval $[time, time')$. Such a function thus defines a trajectory or flow over a given domain, as in hybrid automata [Hen96]. As before we also require that the continuum function remains constant, the time interval is well-defined, and the discrete variable copies match their continuous counterparts.

Finally, we define the preemption operator.

lift-definition *hInit* :: $('c \Rightarrow bool) \Rightarrow ('a,\ 'c)\ hrel$ **is**
$\lambda\ P.\ \{(s,\ s').\ P\ (cont\ s\ (time\ s)) \wedge cont\ s = cont\ s' \wedge isHCT1(s,\ s') \wedge isHCT2(s,\ s')\}$
by *auto*

lift-definition *hFinal* :: $('c \Rightarrow bool) \Rightarrow ('a,\ 'c)\ hrel$ **is**
$\lambda\ P.\ \{(s,\ s').\ P\ (cont\ s\ (time\ s')) \wedge cont\ s = cont\ s' \wedge isHCT1(s,\ s') \wedge isHCT2(s,\ s')\}$
by *auto*

definition *hPreempt* ::
$('a,\ 'c)\ hrel \Rightarrow ('c \Rightarrow bool) \Rightarrow ('a,\ 'c)\ hrel \Rightarrow ('a,\ 'c)\ hrel$
(**infixr** $[\text{-}]_H$ *90*) **where**
$P\ [B]_H\ Q = (Q \lhd hInit\ B \rhd (\lceil(\lambda\ (\tau,\ v).\ \neg\ B\ v)\rceil_H \wedge_H P))$
$\vee_H\ ((\lceil(\lambda\ (\tau,\ v).\ \neg\ B\ v)\rceil_H \wedge_H hFinal\ B \wedge_H P)\ ;_H\ Q)$

For the sake of simplicity in this implementation, we define preemption in terms of two auxiliary operators. The *hInit* and *hFinal* operators both take a predicate over the continuous state, and assert that this predicate is true at, respectively, the beginning and end of the interval. The *hPreempt* operator is then defined following the definition in Section 4. Unlike previous operators we do not need to obtain this by lifting, but define it purely in terms of previous operators.

5.4 Complete lattice of hybrid relations

With all the core operators defined, we now proceed to prove that our theory of hybrid relations forms a complete lattice, which will, amongst other things, provide us with the ability to express recursive and iterative behaviours in the form of fixed points. We first show that hybrid relations form a partially ordered set, which we do by demonstrating that the *hrel* type is an instance of the *order* type class. The order and strict order operators for hybrid relations are defined by lifting the subset and proper subset operators, respectively.

instantiation *hrel* :: (*type*, *type*) *order*
begin
  lift-definition *less-eq-hrel* :: ('*a*, '*c*) *hrel* ⇒ ('*a*, '*c*) *hrel* ⇒ *bool* **is** *op* ⊆ **.**
  lift-definition *less-hrel* :: ('*a*, '*c*) *hrel* ⇒ ('*a*, '*c*) *hrel* ⇒ *bool* **is** *op* ⊂ **.**
instance proof
  fix *x y z* :: ('*a*, '*c*) *hrel*

  — Reflexivity
  show $x \leq x$
    by (*transfer*, *auto*)

  — Transitivity
  show $x \leq y \Longrightarrow y \leq z \Longrightarrow x \leq z$
    by (*transfer*, *auto*)

  — Antisymmetry
  show $x \leq y \Longrightarrow y \leq x \Longrightarrow x = y$
    by (*transfer*, *auto*)

  — Soundness of strict order
  show $(x < y) = (x \leq y \wedge \neg\ y \leq x)$
    by (*transfer*, *auto*)
qed
end

abbreviation *hRefine* :: ('*a*, '*c*) *hrel* ⇒ ('*a*, '*c*) *hrel* ⇒ *bool* (**infixl** $\sqsubseteq_H$ *60*) **where**
hRefine x y ≡ $y \leq x$

This also allows us to define the refinement relation for hybrid relations, $P \sqsubseteq_H Q$, which is simply the inverse order relation. This, along with the interval operator, provides us with the ability to state invariants of hybrid relations through a suitable refinement statement. For example, we below prove that the formula $x = -t^2$ satisfies the invariant $x < 10$. This is achieved through a transfer, followed by application of auto to simply the proof

structure, and finally application of the Z3 SMT solver to the resulting mathematical formula.

lemma *cont-inv1*: $\lceil \lambda\ (t,x::real).\ x < 10 \rceil_H \sqsubseteq_H \lceil \lambda\ (t,x).\ x = -t^2 \rceil_H$
by (*transfer*, *auto*, *smt realpow-square-minus-le*)

We next show that hybrid relations form a complete lattice by instantiating the corresponding type class. The lattice classes in Isabelle characterise the inverse lattice to the typical refinement lattice, and so infimum and supremum are swapped.

instantiation *hrel* :: (*type*, *type*) *complete-lattice*
begin
definition *inf-hrel-def* [*simp*]: *inf P Q* = (*P* $\wedge_H$ *Q*)
definition *sup-hrel-def* [*simp*]: *sup P Q* = (*P* $\sqcap_H$ *Q*)

The infimum and supremum operators are defined to be conjunction and non-deterministic choice, respectively.

definition *bot-hrel-def* [*simp*]: *bot* = $false_H$
definition *top-hrel-def* [*simp*]: *top* = $true_H$

The bottom and top of the lattice are $false_H$, the most deterministic hybrid relation, and $true_H$, the most non-determistic hybrid relation.

lift-definition *Inf-hrel* :: ('*a*, '*c*) *hrel set* ⇒ ('*a*, '*c*) *hrel* **is**
$\lambda\ A.$ *if* $(A = \{\})$ *then* $\{(s, s').$ *cont* s = *cont* $s' \wedge$ *isHCT1*$(s, s') \wedge$ *isHCT2*$(s, s')\}$
else Inter A
by (*rename-tac set prod*, *case-tac set* = {}, *auto*)

The big infimum operator is similar to the big set intersection operator ($\bigcap A$), however we have to explicitly define that $\bigcap \emptyset$ is not the universal relation (which is not closed under our healthiness conditions), but rather corresponds to $true_H$.

lift-definition *Sup-hrel* :: ('*a*, '*c*) *hrel set* ⇒ ('*a*, '*c*) *hrel* **is** *Union*
by *auto*

Finally, the big supremum operator is simply big union ($\bigcup A$). We then prove the requisite properties of a complete lattice (though the proof is omitted).

instance proof
— proof omitted **qed** (*transfer*, *auto*)+
end

5.5 Algebraic laws of hybrid relations

In this section we will show that several of the key laws of programming are retained by our hybrid relational calculus, and also demonstrate that the laws of duration calculus hold. This section thus serves to perform some preliminary validation of our model. We first show some simple properties about non-deterministic choice.

theorem *hChoice-comm*: *P* $\sqcap_H$ *Q* = *Q* $\sqcap_H$ *P*
by (*transfer*, *auto*)

theorem *hChoice-idem*: *P* $\sqcap_H$ *P* = *P*

by (*transfer*, *auto*)

Non-deterministic choice is commutative and idempotent. This follows directly from our definition of the operator in terms of the set union.

theorem *hSeq-assoc*: $P ;_H (Q ;_H R) = (P ;_H Q) ;_H R$
by (*transfer*, *auto*)

Sequential composition is associative; this is both a law of programming and of duration calculus. This fact also follows from the fact that sequential composition is simply lifted relational composition.

theorem *hSeq-mono*:
assumes $P \sqsubseteq_H P'$ $Q \sqsubseteq_H Q'$
shows $P ;_H Q \sqsubseteq_H P' ;_H Q'$
using *assms* **by** (*transfer*, *auto*)

Sequential composition is also monotone with respect to refinement. Intuitively this means that if we can demonstrate a refinement of the whole through a refinement of the parts.

theorem *hFalse-zero* [*simp*]:
$false_H ;_H P = false_H$ $P ;_H false_H = false_H$
by (*transfer*, *auto*)+

We can also show that $false_H$ is both a left and right zero of sequential composition. If we at any point in a sequential program exhibit miraculous behaviour, then the whole program is miraculous.

theorem *hSkip-left-unit* [*simp*]: $II_H ;_H P = P$
— apply-style proof omitted
theorem *hSkip-right-unit* [*simp*]: $P ;_H II_H = P$
— apply-style proof omitted

Proof of the next two properties is a little involved so we omit them. We show that the skip operator is both a left and right unit of sequential composition. This does not follow automatically, since we have redefined skip, hence we perform a manual proof.

theorem *hChoice-dist*: $(P \sqcap_H Q) ;_H R = (P ;_H R) \sqcap_H (Q ;_H R)$
by (*transfer*, *auto*)

The next law is a distributivity law. It states that if we make a choice between P and Q and then perform R, this is really just the same as making a choice between P ; R and Q ; R. Having proved a few facts about our calculus, we now proceed to show that it satisfies some more substantial algebras, to aid in validation of soundness. We first show that our calculus forms a *quantale* [Ros90], which follows through a simple automated proof.

instantiation *hrel* :: (*type*, *type*) *unital-quantale-plus*
begin
definition *one-hrel-def* [*simp*]: $1 = II_H$
definition *times-hrel-def* [*simp*]: $P \cdot Q = P ;_H Q$
instance — proof omitted **end**

A quantale is a combination of a complete lattice and multiplicative monoid, that connects the two via laws that show how multiplication composition distributes through the supremum operator. Quantales are very useful and general algebraic structures for characterising imperative programs with iteration and non-deterministic choice. Here, we use a pre-existing implementation of quantales in Isabelle [AS12]. The quantale that we define augments the previously defined complete lattice with the multiplication operator defined to be sequential composition, and the unit of sequential composition to be skip. We then use the fact that our structure is a quantale to derive some further algebraic structures.

instantiation *hrel* :: (*type*, *type*) *dioid-one-zero*
begin
  definition *zero-hrel-def* [*simp*]: $0 = false_H$
  definition *plus-hrel-def* [*simp*]: $P + Q = P \sqcap_H Q$
instance
  by (*intro-classes*, *simp-all*) (*transfer*, *auto*)+
end

Dioids correspond to idempotent semirings and have also proved useful structures for describing imperative programs [FSW11]. In addition to the multiplication operator already defined, they also include a plus operator (and a unit) that in this setting corresponds to the non-deterministic choice. The proof of this is fully automatic. We next move on from this to consider *Kleene algebras* [Koz90].

instantiation *hrel* :: (*type*, *type*) *kleene-algebra*
begin
  definition *star-hrel* :: ('*a*, '*c*) *hrel* ⇒ ('*a*, '*c*) *hrel* **where**
  star-hrel P = *qstar P*
instance proof
  fix *x y z*
  show $x^\star \sqsubseteq_H 1 + x \cdot x^\star$
    by (*simp only*: *star-hrel-def hrel-plus-sup star-unfoldl*)
  show $y \sqsubseteq_H z + x \cdot y \longrightarrow y \sqsubseteq_H x^\star \cdot z$
    using *unital-quantale-class.star-inductl*
    by (*unfold star-hrel-def hrel-plus-sup*, *blast*)
  show $y \sqsubseteq_H z + y \cdot x \longrightarrow y \sqsubseteq_H z \cdot x^\star$
    using *unital-quantale-class.star-inductr*
    by (*unfold star-hrel-def hrel-plus-sup*, *blast*)
qed
end

Kleene algebras augment dioids with a closure operation called Kleene star ($P^\star$) that is used to axiomatise finite iteration. That is to say, $P^\star$ corresponds to all possible finite iterations of P. Recall that multiplication corresponds to sequential composition, 1 corresponds to skip, and plus corresponds to non-deterministic choice. It is necessary to prove three laws about these operators:

1. the star unfold law, that shows how an iteration either exits (i.e. skips) or else makes a copy of the body and then iterates;

2. the left-hand star induction law, that effectively shows that the star is the least

(strongest) fixed point of $x = a + x \cdot b$;

3. the right-hand star induction law, that is the dual of the above.

All these properties can be discharged from laws we've already proved about quantales. Showing that our calculus is a Kleene algebra serves to valid that we follow the usual laws of imperative programming, and also opens the door to a wealth of program verification techniques in Isabelle (see for example [ASW13]). We also show that our calculus forms an Omega algebra.

instantiation *hrel* :: (*type*, *type*) *omega-algebra*
begin
  definition *omega-hrel* :: ('*a*, '*c*) *hrel* ⇒ ('*a*, '*c*) *hrel* **where**
  omega-hrel P = *qomega P*
instance proof
  fix *x y z*
  show $x \cdot x^\omega \sqsubseteq_H x^\omega$
    by (*metis eq-iff omega-hrel-def qomega-unfold*)
  show $z + x \cdot y \sqsubseteq_H y \longrightarrow x^\omega + x^\star \cdot z \sqsubseteq_H y$
    using *qomega-coinduct*[*of y z x*]
    by (*metis* (*no-types*) *hrel-plus-sup omega-hrel-def star-hrel-def*)
qed
end

Omega algebras further extend Kleene algebras with a closure operator called Omega (P^ω), that is used to axiomatise infinite iteration. This is evidence by the first of the two laws that shows that an omega continues to unfold with no termination state reached. Omega thus corresponds to the greatest fixed point, as illustrated by the second of our two laws. Both these properties can again be discharged through the quantale laws.

5.6 Duration calculus laws

We now turn our attention to the laws of duration calculus. A collection of axioms for the interval operator is given in [ZRH93] and we will here reproduce some of the most important of these.

theorem *hInt-true*: $\lceil \lambda\ x.\ True \rceil_H = \mathcal{L}[\lambda\ t.\ t > 0]_H$
  by (*transfer*, *auto*)

This laws shows that the interval with predicate *true* specifies all possible intervals with a non-zero duration.

theorem *hInt-false*: $\lceil \lambda\ x.\ False \rceil_H = false_H$
  by (*transfer*, *auto*)

This laws shows that the interval with predicate *false* denotes the miraculous hybrid program.

theorem *hInt-conj*: $\lceil \lambda\ x.\ P(x) \wedge Q(x) \rceil_H = \lceil P \rceil_H \wedge_H \lceil Q \rceil_H$
  by (*transfer*, *auto*)

This laws shows that the conjunction of two predicates (over a common alphabet) equates to the conjunction of the two intervals. Intuitively, P and Q are both true for the whole

interval, if and only if P is true for the whole interval, and Q is also true for the whole interval.

theorem *hInt-disj*: $\lceil \lambda\ x.\ P(x) \vee Q(x) \rceil_H \sqsubseteq_H (\lceil P \rceil_H \vee_H \lceil Q \rceil_H)$
by (*transfer*, *auto*)

This law is similar to the law above, but is an inequality rather than an equality. Specifically, if P is true for the whole interval, or Q is true for the whole interval, then at each point each P is true or Q is true. However, the inverse is not necessarily the case.

theorem *hInt-seq*: $\lceil P \rceil_H \sqsubseteq_H \lceil P \rceil_H ;_H \lceil P \rceil_H$
by (*transfer*, *auto simp add*: *relcomp-unfold*)

This law states that an interval can be refined by splitting it into two intervals possessing the same property. Thus we have mechanised the main operators of our language, and proved key algebraic properties about them. In the future we will continue to refinement and extend this language and use it to reason about some example hybrid dynamical systems.

5.7 Discussion

Our present mechanisation, as mentioned, is a relatively shallow embedding of the hybrid relational calculus. The advantages and disadvantages of such a nature of implementation are well-known and are discussed in our previous paper [FZW14]. Shallow embeddings provide quick access to all the proof facilities of the object logic, in this case Isabelle/HOL. Thus they are of most use when applied to verification because of their lightweight nature. In the future we will explore whether this kind of shallow embedding is sufficient to both prove laws of programming and perform program verification, or whether a deeper embedding is needed like the one explored in [FZW14]. Nevertheless, we feel the results obtain in this section are promising, and hope we could use this as the basis to build a hybrid theorem prover building on ideas from tools like KeYmaera [Pla10b].

6 Conclusion

This deliverable summarised our current work towards the creation of a denotational semantics for Modelica based in the Unifying Theories of Programming. We first described the existing semantics for Modelica, and thus set some aspirations for the work we hope to accomplish in this task. We then presented an initial version of our UTP of differential equations and hybrid behaviour, accompanied by a preliminary mechanisation in Isabelle. The latter allowed us to also show that our calculus satisfied well known laws of programming. We also hope that by combining continuous invariants with timed reactive designs [HDM10, CW15], that we can develop a refinement technique for hybrid systems.

In the following years we will continue to develop and refine this theory, and attempt to use it to model and prove properties of some more substantial examples. Our eventual aim is that this theory will provide the basis for the Modelica semantics, that we will continue to develop in deliverables D2.2c and D2.3b. Moreover, this work will form the

basis for our *lingua franca* language *INTO-CSP*, that will allow mathematically principled integration of the different notations and languages in INTO-CPS. We will in future also begin to link this semantics to the parallel work on VDM-RT (in D2.1b) and FMI (in D2.1d). This in turn will allow us to ensure the soundness of our co-modelling approach in the other parts of INTO-CPS.

References

[ÅEH10] Johan Åkesson, Torbjörn Ekman, and Görel Hedin. Implementation of a Modelica compiler using JastAdd attribute grammars. *Science of Computer Programming*, 75(1–2):21–38, 2010. Special Issue on ETAPS 2006 and 2007 Workshops on Language Descriptions, Tools, and Applications (LDTA '06 and '07). doi:10.1016/j.scico.2009.07.003.

[AP98] Uri M. Ascher and Linda R. Petzold. *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*. SIAM, Philadelphia, 1998.

[AS12] A. Armstrong and G. Struth. Automated reasoning in higher-order regular algebra. In *13th Intl. Conf. on Relational and Algebraic Methods in Computer Science (RAMiCS)*, volume 7560 of *LNCS*. Springer, September 2012.

[ASW13] A. Armstrong, G. Struth, and T. Weber. Program analysis and verification based on kleene algebra in isabelle/hol. In *4th Intl. Conf. on Interactive Theorem Proving (ITP)*, volume 7998 of *LNCS*. Springer, 2013.

[BAF06] Bernhard Bachmann, Peter Aronsson, and Peter Fritzson. Robust initialization of differential algebraic equation. In 5th *Int. Modelica Conference*, Vienna, Austria, September 2006. URL: https://www.modelica.org/events/modelica2006/Proceedings/sessions/Session6a2.pdf.

[BBCP12] Albert Benveniste, Timothy Bourke, Benoit Caillaud, and Marc Pouzet. Non-Standard Semantics of Hybrid Systems Modelers. *Journal of Computer and System Sciences (JCSS)*, 78:877–910, May 2012. Special issue in honor of Amir Pnueli. doi:10.1016/j.jcss.2011.08.009.

[BBCP15] Timothy Bourke, Albert Benveniste, Benoit Caillaud, and Marc Pouzet. Exploiting and Extending Synchronous Languages for Hybrid Modeling. In *9th MODPROD Workshop on Model-Based Product Development*, Linköping University, Linköping, Sweden, February 2015. URL: http://www.modprod.liu.se/modprod-2015/1.620167/modprod2015-day2-talk05-TimBourke-SynchronousLanguagesHybrid.pdf.

[BBN11] J. C. Blanchette, L. Bulwahn, and T. Nipkow. Automatic proof and disproof in Isabelle/HOL. In *FroCoS*, volume 6989 of *LNCS*, pages 12–27. Springer, 2011.

[BEH+03] Albert Benveniste, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The synchronous languages 12 years later. In *Proceedings of the IEEE*, volume 91 (1), pages 64–83, 2003. doi:10.1109/JPROC.2002.805826.

[BF14] Simon Bliudze and Sébastien Furic. An Operational Semantics for Hybrid Systems Involving Behavioral Abstraction. In Hubertus Tummescheit and Karl-Erik Årzén, editors, 10th *Int. Modelica Conference*, Lund, Sweden, March 2014. doi:10.3384/ECP14096693.

[BLM15] Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. Formalization of real analysis: A survey of proof assistants and libraries. *Mathematical Structures in Computer Science*, pages 1–38, January 2015.

[Bro10] David Broman. *Meta-Languages and Semantics for Equation-Based Modeling and Simulation*. PhD thesis, Linköping University, PELAB - Programming Environment Laboratory, The Institute of Technology, 2010. URL: http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-58743.

[Cel13] François E. Cellier. Numerical Simulation of Dynamic Systems. Lecture notes, ETH Zürich, 2013. URL: https://www.inf.ethz.ch/personal/fcellier/Lect/NSDS/Ppt/nsds_ppt_engl.html.

[CSW05] Ana Cavalcanti, Augusto Sampaio, and Jim Woodcock. Unifying classes and processes. *Software and System Modeling*, 4(3):277–296, 2005.

[CW15] Samuel Canham and Jim Woodcock. Three approaches to timed external choice in UTP. In *Unifying Theories of Programming*, volume 8963, pages 1–20. Springer, 2015.

[EP07] Abbas Edalat and Dirk Pattinson. Denotational semantics of hybrid automata. *Journal of Logic and Algebraic Programming*, 73(1–2):3–21, 2007.

[FAL+05] Peter Fritzson, Peter Aronsson, Håkan Lundvall, Kaj Nyström, Adrian Pop, Levon Saldamli, and David Broman. The OpenModelica Modeling, Simulation, and Software Development Environment. *Simulation News Europe*, 44(45), December 2005.

[Fid99] C. J. Fidge. Modelling discrete behaviour in a continuous-time formalism. In K. Araki, A. Galloway, and Taguchi K., editors, *Proc. 1st Intl. Conf. on Integrated Formal Methods (IFM)*. Springer, 1999.

[Fle00] J. D. Fleuriot. On the mechanization of real analysis in Isabelle/HOL. In *13th. Intl. Conf. on Theorem Proving Higher Order Logics (TPHOLs)*, volume 1869 of *LNCS*, pages 145–161. Springer, 2000.

[FMI14] FMI development group. Functional Mock-up Interface for Model Exchange and Co-Simulation v2.0. Modelica Association Project "FMI", October 2014. Standard Specification. URL: https://www.fmi-standard.org/.

[FMW+14] S. Foster, A. Miyazawa, J. Woodcock, A. Cavalcanti, J. Fitzgerald, and P. Larsen. An approach for managing semantic heterogeneity in systems of systems engineering. In *Proc. 9th Intl. Conf. on Systems of Systems Engineering*. IEEE, 2014.

[FPBA09] P. Fritzson, A. Pop, D. Broman, and P. Aronsson. Formal Semantics Based Translator Generation and Tool Development in Practice. In *Software Engineering Conference, 2009. ASWEC '09. Australian*, pages 256–266, April 2009. doi:10.1109/ASWEC.2009.46.

[Fri14] Peter Fritzson. *Principles of Object Oriented Modeling and Simulation with Modelica 3.3: A Cyber-Physical Approach*. Wiley IEEE Press, 2014.

[FSW11] S. Foster, G. Struth, and T. Weber. Automated engineering of relational and algebraic methods in Isabelle/HOL. In H. Swart, editor, *RAMICS 2011*, volume 6663 of *LNCS*, pages 52–67. Springer, 2011.

[FZW14] S. Foster, F. Zeyda, and J. Woodcock. Isabelle/UTP: A mechanised theory engineering framework. In David Naumann, editor, *Proc. 5th Intl. Symposium on Unifying Theories of Programming (UTP 2014)*, volume 8963 of *LNCS*, pages 21–41. Springer, 2014.

[Har98] J. Harrison. *Theorem Proving with the Real Numbers*. Distinguished Dissertations. Springer, 1998.

[Har05] J. Harrison. A HOL theory of Euclidean space. In J. Hurd and T. Melham, editors, *Proc. 18th Intl. Conf. on Theorem Proving in Higher Order Logics (TPHOLs)*, volume 3603 of *LNCS*, pages 114–129. Springer, 2005.

[HDM10] I. J. Hayes, S. E. Dunne, and L. Meinicke. Unifying theories of programming that distinguish nontermination and abort. In *Mathematics of Program Construction (MPC)*, volume 6120 of *LNCS*, pages 178–194. Springer, 2010.

[He94] Jifeng He. From CSP to hybrid systems. In A. W. Roscoe, editor, *A classical mind: essays in honour of C. A. R. Hoare*, pages 171–189. Prentice Hall, 1994.

[He15] Jifeng He. HRML: a hybrid relational modelling language. In *IEEE International Conference on Software Quality, Reliability and Security (QRS 2015)*, August 2015. URL: http://paris.utdallas.edu/qrs15/slides/keynote/QRS-2015-Keynote-03-Jifeng-He.pdf.

[Heh93] E. C. R. Hehner. *A Practical Theory of Programming*. Springer, 1993.

[Hen96] T. A. Henzinger. *The theory of hybrid automata*, pages 278–292. IEEE, 1996.

[HH98] Tony Hoare and Jifeng He. *Unifying Theories of Programming*. Prentice-Hall, 1998.

[HK13] B. Huffman and O. Kunčar. Lifting and transfer: A modular design for quotients in Isabelle/HOL. In *3rd Intl. Conf. on Certified Programs and Proofs*, volume 8307 of *LNCS*, pages 131–146. Springer, 2013.

[Hoa85] Tony Hoare. *Communicating Sequential Processes*. Prentice-Hall International, Englewood Cliffs, New Jersey 07632, 1985.

[IH12] F. Immler and J. Hölzl. Numerical analysis of ordinary differential equations in Isabelle/HOL. In *3rd Intl. Conf. on Interactive Theorem Proving (ITP)*, volume 7406 of *LNCS*, pages 377 – 392. Springer, 2012.

[Imm14] F. Immler. Formally verified computation of enclosures of solutions of ordinary differential equations. In *Proc. 6th NASA Formal Methods Symposium (NFM)*, volume 8430 of *LNCS*. Springer, 2014.

[KF98] David Kågedal and Peter Fritzson. Generating a Modelica compiler from natural semantics specifications. In *Proceedings of the 1998 Summer Computer Simulation Conference (SCSC'98)*, 1998.

[Koz90] D. Kozen. On Kleene algebras and closed semirings. In B. Rovan, editor, *MFCS'90*, volume 452 of *LNCS*, pages 26–47. Springer, 1990.

[LLQ+10] J. Liu, J. Lv, Z. Quan, N. Zhan, H. Zhao, C. Zhou, and L. Zou. A calculus for Hybrid CSP. In *8th Asian Symp. on Programming Languages and Systems (APLAS)*, volume 6461 of *LNCS*, pages 1–15. Springer, 2010.

[LZ05] Edward A. Lee and Haiyang Zheng. Operational Semantics of Hybrid Systems. In Manfred Morari and Lothar Thiele, editors, *Hybrid Systems: Computation and Control*, volume 3414 of *Lecture Notes in Computer Science*, pages 25–53. Springer Berlin / Heidelberg, 2005. doi:10.1007/978-3-540-31954-2_2.

[LZ07] Edward A. Lee and Haiyang Zheng. Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. In *EMSOFT '07: Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 114–123, New York, NY, USA, 2007. ACM. doi:10.1145/1289927.1289949.

[LZZ08] Zhihua Li, Ling Zheng, and Huili Zhang. Solving PDE models in Modelica. In *Information Science and Engineering, 2008. ISISE'08. International Symposium on Information Science and Engieering. ISISE*, volume 1, pages 53–57. IEEE, 2008.

[Mod14] Modelica Association. Modelica - A Unified Object-Oriented Language for Systems Modeling - Version 3.3 Revision 1. Standard Specification, July 2014. URL: http://www.modelica.org/.

[Mos85] B. Moszkowski. A temporal logic for multi-level reasoning about hardware. *IEEE Computer*, 18(2):10–19, 1985.

[MS93] Sven Erik Mattsson and Gustaf Söderlind. Index Reduction in Differential-Algebraic Equations Using Dummy Derivatives. *SIAM Journal on Scientific Computing*, 14(3):677–692, 1993. doi:10.1137/0914043.

[NWP02] T. Nipkow, M. Wenzel, and L. C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[OCW07] M. V. M. Oliveira, A. L. C. Cavalcanti, and J. C. P. Woodcock. A UTP Semantics for Circus. *Formal Aspects of Computing*, **21**(1):3 – 32, 2007.

[OEM99] M. Otter, H. Elmqvist, and S.E. Mattsson. Hybrid Modeling in Modelica based on the Synchronous Data Flow Principle. In *Computer Aided Control System Design, 1999. Proceedings of the 1999 IEEE International Symposium on*, pages 151–157, September 1999. doi:10.1109/CACSD.1999.808640.

[Pan88] Constantinos C. Pantelides. The consistent initialization of differential-algebraic systems. *SIAM Journal on Scientific and Statistical Computing*, 9(2):213–231, 1988. doi:10.1137/0909014.

[Pet95] Mikael Pettersson. *Compiling Natural Semantics*. Doctoral thesis No 413, Department of Computer and Information Science, Linköping University, Sweden, 1995.

[PF06] Adrian Pop and Peter Fritzson. Metamodelica: A unified equation-based semantical and mathematical modeling language. In DavidE. Lightfoot and Clemens Szyperski, editors, *Modular Programming Languages*, volume 4228 of *Lecture Notes in Computer Science*, pages 211–229. Springer Berlin Heidelberg, 2006. doi:10.1007/11860990_14.

[Pla07] André Platzer. Differential dynamic logic for verifying parametric hybrid systems. Technical Report 5-2007, Carnegie Mellon University, May 2007.

[Pla08] André Platzer. Differential dynamic logic for hybrid systems. *Journal of Automated Reasoning*, 41(2):143–189, 2008.

[Pla10a] André Platzer. Differential-algebraic dynamic logic for differential-algebraic programs. *Journal of Logic and Computation*, 20(1):309–352, 2010.

[Pla10b] André Platzer. *Logical Analysis of Hybrid Systems*. Springer, 2010.

[Pne77] A. Pneuli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science*, pages 46–57, 1977.

[Ros90] K. I. Rosenthal. Quantales and their applications. *Pitman Research Notes in Mathematics Series*, 234, 1990.

[Sal06] Levon Saldamli. *PDEModelica - A High-Level Language for Modeling with Partial Differential Equations*. Doctoral thesis No 1016, Department of Computer and Information Science, Linköping University, Sweden, May 2006. URL: http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-7281.

[SCAP15] Lucas Satabin, Jean-Louis Colaço, Olivier Andrieu, and Bruno Pagano. Towards a Formalized Modelica Subset. In Hilding Elmqvist Peter Fritzson, editor, 11th *Int. Modelica Conference*, Versailles, France, September 2015. doi:10.3384/ecp15118637.

[SCS06] Thiago Santos, Ana Cavalcanti, and Augusto Sampaio. Object-Orientation in the UTP. In S. Dunne and B. Stoddart, editors, *UTP 2006: First International Symposium on Unifying Theories of Programming*, volume 4010 of *LNCS*, pages 20–38. Springer-Verlag, 2006.

[SFP14] Martin Sjölund, Peter Fritzson, and Adrian Pop. Bootstrapping a Compiler for an Equation-Based Object-Oriented Language. *Modeling, Identification and Control*, 35(1):1–19, 2014. doi:10.4173/mic.2014.1.1.

[Thi15] Bernhard Thiele. *Framework for Modelica Based Function Development*. Dissertation, Technische Universität München, 2015. URL: http://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:bvb:91-diss-20150902-1249772-1-2.

[TKF15] Bernhard Thiele, Alois Knoll, and Peter Fritzson. Towards Qualifiable Code Generation from a Clocked Synchronous Subset of Modelica. *Modeling, Identification and Control*, 36(1):23–52, 2015. doi:10.4173/mic.2015.1.3.

[WD96] Jim Woodcock and Jim Davies. *Using Z – Specification, Refinement, and Proof*. Prentice Hall International Series in Computer Science, 1996.

[Woo14] Jim Woodcock. Engineering UToPiA - Formal Semantics for CML. In Cliff Jones, Pekka Pihlajasaari, and Jun Sun, editors, *FM 2014: Formal Methods*, volume 8442 of *Lecture Notes in Computer Science*, pages 22–41. Springer International Publishing, 2014.

[WWC13] Kun Wei, Jim Woodcock, and Ana Cavalcanti. Circus Time with Reactive Designs. In *Unifying Theories of Programming*, volume 7681 of *LNCS*, pages 68–87. Springer, 2013.

[ZGZ99] C. Zhou, D. P. Guelev, and N. Zhan. A higher-order duration calculus. Technical Report 167, United Nations University International Institute for Software Technology (UNU/IIST), July 1999.

[ZH96] C. Zhou and M. R. Hansen. Chopping a point. In *Proc. 7th BCS-FACS Refinement Workshop*. Springer, 1996.

[ZHR91] C. Zhou, C. A. R. Hoare, and A. P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1991.

[ZJR96] C. Zhou, W. Ji, and A. P. Ravn. A formal description of hybrid systems. In R. Alur, T. A. Henzinger, and E. D. Sontag, editors, *Hybrid Systems III*, volume 1066 of *LNCS*, pages 511–530. Springer, 1996.

[ZRH93] C. Zhou, A. P. Ravn, and M. R. Hansen. An extended duration calculus for hybrid real-time systems. In R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, editors, *Hybrid Systems*, volume 736 of *LNCS*, pages 36–59. Springer, 1993. doi:10.1007/3-540-57318-6_23.

[ZYZ+14] H. Zhao, M. Yang, N. Zhan, B. Gu, L. Zou, and Y. Chen. Formal verification of a descent guidance control program of a lunar lander. In *19th International Symposium on Formal Methods (FM)*, volume 8442 of *LNCS*, pages 733–748. Springer, 2014.

[ZZWF15] L. Zou, N. Zhan, S. Wang, and M. Fränzle. Formal verification of simulink/stateflow diagrams. In *13th International Symposium on Automated Technology for Verification and Analysis*, volume 9364 of *LNCS*, pages 464–481. Springer, 2015.