Towards a UTP semantics for Modelica

Simon Foster[1], Bernhard Thiele[2], Ana Cavalcanti[1], and Jim Woodcock[1]

[1] Department of Computer Science, University of York, United Kingdom
{simon.foster,ana.cavalcanti,jim.woodcock}@york.ac.uk
[2] PELAB, Linköping University, Sweden
bernhard.thiele@liu.se

Abstract. We describe our work on a UTP semantics for the dynamic systems modelling language Modelica. This is a language for modelling a system's continuous behaviour using a combination of differential-algebraic equations and an event-handling system. We develop a novel UTP theory of hybrid relations, inspired by Hybrid CSP and Duration Calculus, that is purely relational and provides uniform handling of continuous and discrete variables. This theory is mechanised in our Isabelle implementation of the UTP, Isabelle/UTP, with which we verify some algebraic properties. Finally, we show how a subset of Modelica models can be given semantics using our theory. When combined with the wealth of existing UTP theories for discrete system modelling, our work enables a sound approach to heterogeneous semantics for Cyber-Physical systems by leveraging the theory linking facilities of the UTP.

1 Introduction

Cyber-Physical Systems (CPS) are a class of computerised system that integrate discrete computation with continuous physical processes. CPS are typically developed using a combination of discrete and continuous models, often in differing heterogeneous languages. This makes verification of trustworthiness challenging. There is a need for unifying semantic models to allow the integration of heterogeneous system components, whilst ensuring that a given set of safety properties is supported. Hoare and He's Unifying Theories of Programming (UTP) has been designed as a framework in which the integration of languages, through the common semantic domain of the alphabetised relation calculus, can be achieved. Semantic models for discrete modelling languages in UTP are already numerous [26,13,36,30], and, therefore, in this paper we focus on semantics of continuous models in the Modelica language.

Modelica [22] is a widely used language for description and modelling of hybrid dynamical systems that compose a continuously evolving physical plant with a discrete controller. Such systems are described using a mixture of differential-algebraic equations (DAEs), and event guards that trigger discontinuous jumps in system behaviour by execution of discrete equations and algorithms – so called "hybrid DAEs". Modelica has a number of commercial implementations

including Dymola[3], Wolfram SystemModeler[4], MapleSim[5] and the open-source implementation, OpenModelica[6]. However, the Modelica language has an incomplete formal semantics; though the semantics of DAEs is well known, the event iteration system currently does not have a formal semantics. Here we give a denotational semantics to a fragment of Modelica using a UTP theory of hybrid relations. Additionally to clarifying the semantics of Modelica, this allows us to consider the combination of continuous and discrete models through common theoretical factors and theory linking.

Our approach to giving a semantics to Modelica is three-fold. Firstly, we create a UTP theory of hybrid relations, building on the work of He [14,15], Zhou [33,32], Zhan [21], and others. This theory extends the alphabet of UTP predicates with continuous variables $\underline{c} \in \mathrm{con}\alpha$ and is defined by novel healthiness conditions that characterise these variables as piecewise continuous functions.

Secondly, we define the operators of our hybrid relational calculus, which is similar to the imperative subset of $\mathcal{HCSP}$ [34], but extended with an interval operator [33] that provides a continuous specification statement. In particular we provide support for semi-explicit DAEs and continous variable preemption. As with Hybrid CSP, we base the denotational semantics around the Duration Calculus [33], though the semantics is purely relational. Moreover, we provide a uniform account of both discrete and continuous variables by linking the latter to discrete "copy" variables that give the valuation at the beginning and end of a continuous evolution. Thus, both discrete and continuous variables can be manipulated with the same operators; in the latter case this provides initial value constraints. Our model of hybrid relations has also mechanised in our UTP proof assistant, *Isabelle/UTP* [10], that provides theorem proving facilities.

Thirdly, we define a preliminary denotational semantics for Modelica through a mapping into the hybrid relational calculus. This mapping primarily considers the event-handling mechanism of Modelica, whereby specific conditions on continuous variables can lead to both discontinuous jumps in variables, and also changes to the equations active in the DAE system.

The remainder of our paper is structured as follows. In section 2, we provide background on hybrid systems by briefly surveying the literature, with particular emphasis on works related to the UTP. In section 3 we briefly describe the UTP, and in section 4 we introduce the Modelica language. In section 5, we describe our UTP theory of hybrid relations. In section 6, we use our UTP theory to build a hybrid relational calculus, including operators for specifying continuous invariants, differential equations, and preemption. In section 7, we outline our mechanisation of the hybrid relational calculus in Isabelle [23,10]. In section 8, we use our hybrid relational calculus to give a high-level denotational semantics to the Modelica language, focusing principally on the interaction between

---

[3] http://www.3ds.com/products-services/catia/products/dymola
[4] http://www.wolfram.com/system-modeler/
[5] http://www.maplesoft.com/products/maplesim/
[6] https://www.openmodelica.org/

evolution of DAEs and the event handling system. Finally in section 9, we draw conclusions.

2 Related work: Hybrid Systems

The majority of the work on hybrid systems takes inspiration from Hybrid Automata [16], an extension of finite state automata that allows the specification of continuous behaviour. A hybrid automaton consists of a finite set of states labelled by ODEs, a state invariant, and initial conditions. 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 variable values continuously flow from one value to another. When one of the jump conditions of an outgoing edge is satisfied, the event, if present, can instantaneously execute, potentially resulting in a discontinuity, and the targeted hybrid state is activated. Thus a hybrid automata is characterised by behaviour that includes both continuous flows also discrete jumps. Hybrid automata are given a denotational semantics in terms of piecewise continuous functions [16] $\mathbb{R} \to \mathbb{R}^n$, also called trajectories, that are continuous except for in a finite number of places.

Verification of hybrid systems was made possible through the seminal work of Platzer [27]. This work develops a logic called Differential Dynamic Logic ($\mathsf{d}\mathcal{L}$) that allows us 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 $\mathsf{d}\mathcal{L}$ allows reasoning about hybrid programs. An implementation of $\mathsf{d}\mathcal{L}$ called *KeYmaera* [27] allows the automated verification of systems modelled as hybrid programs. Our notion of hybrid relation is inspired by Platzer's hybrid programs, though we focus on a UTP denotational semantics as opposed to an operational semantics. Our own setting of the Duration Calculus [33] provides us with the necessary machinery to similarly justify a dynamic logic. Moreover, we observe that, with a UTP model, we are in a strong position to extend the work to deal with concurrent hybrid programs, a notion that $\mathsf{d}\mathcal{L}$ does not consider.

Concurrency is considered in Hybrid CSP [14,34] ($\mathcal{HCSP}$), an extension of Hoare's process calculus CSP [17] that adds support for continuous variables as described by differential equations and modelled by standard trajectories, in a similar manner to hybrid automata. $\mathcal{HCSP}$ [14] extends CSP with continuous variables whose behaviour is described by differential equations of the form $\mathcal{F}(\dot{s}, s) = 0$. Interaction between discrete and continuous behaviour takes the form of preemption conditions on continuous variables, timeouts, and interruption of a continuous evolution through CSP events. $\mathcal{HCSP}$ has a denotational semantics that is presented in a predicative style similar to the UTP [18].

Further work on $\mathcal{HCSP}$ [34] enriches 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 [35], which treats variables as piecewise continuous functions. This allows a more precise semantics for operators like preemption that are defined in terms of suitable variable limits. A Hoare logic for this calculus is presented in [21], through the adoption of Platzer's differential invariants, along with an operational semantics. Our work is heavily influenced by $\mathcal{HCSP}$, though we focus on formalising the sequential aspects of hybrid systems, and so formalise a subset of the operators with refined definitions. Our operators formalise continuous after variables by explicitly considering left-limits which is important for Modelica event iteration.

A theorem prover for $\mathcal{HCSP}$ called, HHL Prover [37], has also been developed and applied to verification of Simulink diagrams through a mapping into $\mathcal{HCSP}$ [31]. More recently the fundamentals of hybrid system modelling have been studied in a purely UTP relational setting [15]. This work has produced a language called the Hybrid Relational Modelling Language [15] (HRML), which draws on $\mathcal{HCSP}$, but uses signals rather than CSP's events as the main communication abstraction. Our notation is agnostic in this respect, and could be extended either to support the event or signal paradigm.

Duration Calculus [33] ($\mathcal{DC}$) provides specification of invariants over the continuous time domain, in order to facilitate verification real-time systems. For example, we can write $\lceil x^2 > 7 \rceil$, which specifies all possible intervals of over which $x^2 > 7$ is invariant. The chop operator $P \circ Q$ specifies that an interval may be broken into two subsequent intervals, over which P and then Q hold, respectively. $\mathcal{DC}$ has been extended to provide a semantics for hybrid real-time systems modelling [35], which is then used to give semantics to $\mathcal{HCSP}$ [34]. $\mathcal{DC}$ can also be used to give an account to typical operators of modal and temporal logics. Thus, grounding our semantics in $\mathcal{DC}$ enables us to form continuous specifications about hybrid systems. Different to $\mathcal{DC}$ we provide a purely relational UTP semantics, and also explictly distinguish continuous and discrete variables, instead of modelling the latter as step functions. This distinction allows us to retain standard relational definitions of the majority of discrete UTP operators.

3 Unifying Theories of Programming

Unifying Theories of Programming [18,4] (UTP) is a framework for the specification of formal semantics. It is based on the idea that any temporal model can be expressed as an alphabetised predicate that describes how variables change over time. This idea of "programs-as-predicates" means that the duality of programs and specifications all but disappears, as programs are just a subclass of specifications. 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 UTP further allows that different semantic presentations, such as denotational, algebraic, axiomatic, and operational, can be formally linked through mutual embeddings. This ensures that consistency is maintained between semantic models and that tools that implement them can be combined for multi-pronged analysis and verification of models [10].

Concretely, an alphabetised relation is a pair $(\alpha P, P)$ where αP is the alphabet and P is a predicate all of whose free variables belong to αP. The alphabet can in turn be subdivided $\alpha(P) = \text{in}\alpha(P) \cup \text{out}\alpha(P)$, with input variables $x, y \in \text{in}\alpha(P)$ and output variables $x', y' \in \text{out}\alpha(P)$. The calculus provides the operators typical of first order logic. UTP predicates are ordered by a refinement partial order $P \sqsubseteq Q$ that also defines a complete lattice. Imperative programs can be described using relational operators, such as sequential composition $P \,;\, Q$, if-then-else conditional $P \lhd b \rhd Q$, assignment $x :=_A v$ (for expression v and alphabet A), and skip $I\!I_A$, all of which are given predicative interpretations.

More sophisticated language constructs can be expressed by enriching the theory of alphabetised relations to create UTP theories. A UTP theory consists of (i) a set of observational variables, (ii) a signature, and (iii) 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 $clock : \mathbb{R}$. The signature uses these operational variables to encode the main operators of the target language.

The domain of a UTP theory can be constrained through healthiness conditions, which act as invariants over the observational variables. 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. We can eliminate this kind of behaviour description with an invariant $clock \leq clock'$. In the UTP such conditions are expressed as idempotent functions, for example $\mathbf{HT}(P) = P \wedge clock \leq clock'$, so that healthiness of a predicate P can be expressed as a fixed point equation: $P = \mathbf{HT}(P)$. If we apply $\mathbf{HT}$ to C, the result is miraculous predicate ***false*** and thus C is excluded 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 $\mathbf{HT}$ 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 [9]. Such a denotational semantics provides the "gold standard" for the meaning of language constructs and can then be used to derive other presentations, such as operational and, very often, algebraic.

4 Modelica

Modelica is an equation-based object-oriented language for describing the dynamic behaviour of CPS, standardised by the Modelica Language Specification (MLS) [22]. The MLS is described using English; therefore, its semantics is to some extent subject to interpretation. Quoting from [22, 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."

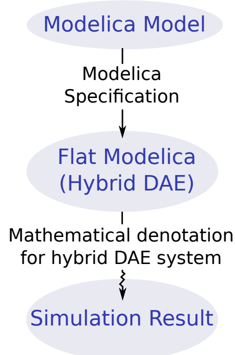

Fig. 1. From model to simulation result.

```
model BouncingBall
  Real h; Real v;
initial equation
  h = 1.0;
equation
  v = der(h);
  der(v) = -9.81;
  when h<0 then
    reinit(v, -0.8*pre(v));
  end when;
end BouncingBall;
```

Fig. 2. Bouncing ball in Modelica.

Fig. 1 illustrates the basic idea. 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. A classical model for a hybrid systems is the bouncing ball. A possible Modelica implementation for a ball with mass 1 kg and an impact coefficient of 0.8 that falls from an initial height of $h = 1\,\mathrm{m}$ is given in Fig. 2. When the ball hits the ground, it changes its velocity v discontinuously and bounces back. `der(h)` and `der(v)` denote the time derivatives $\dot{h}$ and $\dot{v}$ of variables h and v, respectively. The acceleration to the ground is determined by earth's gravitational acceleration $g = 9.81\,\mathrm{m/s}^2$. The discontinuous change of variable v is modelled using a conditionally activated reinitialization equation. The ball hits the ground when condition $h < 0$ becomes true. The `reinit()` operator is used for reinitializing v with the negative value of v (times the impact coefficient) just before condition $h < 0$ becomes true (`pre(v)` returns the left limit of variable v at the event instant).

Several formal specification approaches have been used to give semantics to subsets of the Modelica language. Most of the approaches describe the instantiation and flattening of Modelica models (*i.e.,* the *static semantics*, corresponding to the first stage in Fig. 1) [20,1,28] while others are restricted to discrete-time language subsets [29].

Flat Modelica can be conceptually mapped to a set of differential, algebraic and discrete equations of the following form [22, Appendix C]:

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

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

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

where t denotes time; p is a vector of parameters and constants; $x(t)$ is a vector of dynamic variables of type `Real` and $\dot{x}(t)$ is the vector of its

derivatives; $y(t)$ is a vector of algebraic variables of type `Real`; $m(t_e)$ is a vector of discrete-time variables of type `discrete Real`, `Boolean`, `Integer`, or `String` which changes only at event instants t_e; $m_{\text{pre}}(t_e)$ are the values of m immediately before the current event at event instant t_e; and $c(t_e)$ is a vector containing all Boolean condition expressions, *e.g.*, if-expressions.

2. *Discrete-time behaviour.* The behaviour at an event at time t_e is described by following discrete equations:

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

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

An event fires if any of the conditions $c(t_e)$ change from **false** to **true**. The vector-valued function f_m specifies new values for the discrete variables $m(t_e)$. The vector $c(t_e)$ is defined by the vector-valued function f_e, which contains all `Boolean` condition expressions evaluated at the most recent event t_e; $\mathit{rel}(v(t_e)) = \mathit{rel}([x(t);\ \dot{x}(t);\ y(t);\ t;\ m(t_e);\ m_{\text{pre}}(t_e);\ p])$ is a Boolean-typed vector-valued function containing variables v_i, *e.g.*, $v_1 > v_2$, $v_3 \geq 0$; $m^{\mathbf{B}}(t_e)$ is a vector of discrete-time variables of type `Boolean`, $m^{\mathbf{B}}(t_e) \subseteq m(t_e)$, and $m^{\mathbf{B}}_{\text{pre}}(t_e)$ are the values of $m^{\mathbf{B}}$ immediately before the current event at event instant t_e; $p^{\mathbf{B}}$ are parameters and constants of type `Boolean`, $p^{\mathbf{B}} \subseteq p$.

Simulation means that an initial value problem (IVP) is solved. The equations define a DAE which may have discontinuities and a variable structure and may be controlled by a discrete-event system.

5 Theory of Hybrid Relations

We now proceed to describe our theory of hybrid relations to enable the definition of a relational calculus for modelling sequential hybrid processes. Our model unifies the treatment of discrete and continuous variables so that the same operators may be used for manipulating both. In Modelica, DAEs are used to describe continuously evolving dynamic behaviour of a system. Thus, in the UTP, we first introduce a theory of continuous time processes that embeds trajectories into alphabetised predicates and shows how continuous variables evolve over a given interval. These intervals are used to divide up the evolution of a system into piecewise continuous segments.

Our theory is based on vanilla UTP alphabetised relations, and so is insensitive to termination and stability of continuous processes. Following the UTP philosophy, we consider hybrid behaviour in isolation, and then later augment it with additional structure to allow the finer expression of such properties. Our theory can, for instance, be embedded into timed reactive designs [13,30].

Alphabet. Our model of continuous time introduces observational variables $ti, ti' : \mathbb{R}_{\geq 0}$ that define the start and end time of the current computation interval, as in $\mathcal{DC}$ [35]. We also introduce the expression ℓ to denote the duration of the current interval, where $\ell \triangleq ti' - ti$.

As already said, the alphabetised relational calculus divides the alphabet into input $\text{in}\alpha(P)$ and output variables $\text{out}\alpha(P)$. Inspired by [15], we add a further subdivision $\underline{x}, \underline{y}, \underline{z} \in \text{con}\alpha(P)$, the set of continuous variables, that is orthogonal to the discrete program variables, that is $\text{con}\alpha(P) \cap (\text{in}\alpha(P) \cup \text{out}\alpha(P)) = \emptyset$. The elements of $\text{con}\alpha(P)$ are the variables to be used in differential equations and other continuous constructs.

We assume that all variables consist of a name, type, and optional 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*.

Continuous variables come in two varieties that allows us to talk about a particular instant or about the whole time continuum:

- instant variables – these are continuous variables of type $\mathbb{R}$ that refer to the value at a particular instant;
- trajectory variables – these are time-dependent variables of type $\mathbb{R}_{\geq 0} \to \mathbb{R}$ and give the values over a whole trajectory.

Trajectory variables are total rather than partial functions. This has the advantage that composition operators need not consider explicit combination of trajectories through overriding. Instead, composition further constrains the trajectory functions, potentially over disjoint time domains (as is the case for ;). Valuations of the trajectory exist outside $[ti, ti')$, but they have no relevance.

We require that each trajectory variable $\underline{x} : \mathbb{R}_{\geq 0} \to \mathbb{R}$ is accompanied by discrete before and after "copy" variables with the same name – $x, x' : \mathbb{R}$ – that record the values at the start and limit of the current interval. This, crucially, allows us to use the standard operators of relational calculus for manipulating continuous variables via discrete copies. This allows us to consider the set of purely discrete variables that are not discrete copies of a continuous variable:

$$\text{dis}\alpha(P) = \{x \in \text{in}\alpha(P) \mid \underline{x} \notin \text{con}\alpha(P)\} \cup \{x' \in \text{out}\alpha(P) \mid \underline{x} \notin \text{con}\alpha(P)\}$$

We introduce the following @ operator borrowed from [6] that lifts a predicate in instant variables to one in trajectory variables.

Definition 1. *Continuous variable lifting*

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

The dagger (†) operator is a nominal substitution operator. It applies the given partial function, which maps variables to expressions, as a substitution to the given predicate, so that $P[v/x] = \{x \mapsto v\} \dagger P$. We construct a substitution that maps every flat continuous variable (other than the distinguished time variable $\underline{t} \in [ti..ti')$) 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 that is potentially free in P. Each flat continuous variable $\underline{x} : T$ is thus transformed to have a time-dependent function $\underline{x} : \mathbb{R} \to T$ type. This operator is used to lift time predicates over intervals.

$$P, Q ::= P \,;\, Q \mid P \lhd b \rhd Q \mid x := e \mid P^{*} \mid P^{\omega} \mid \llbracket P \rrbracket \mid \langle F_n \,|\, b \rangle \mid P\,[\,b\,]\,Q$$

Table 1. Signature of hybrid relational calculus

Healthiness conditions. We introduce two healthiness conditions:

$$\textbf{\textit{HCT1}}(P) \triangleq P \land ti \leq ti'$$

$$\textbf{\textit{HCT2}}(P) \triangleq P \land \left(ti < ti' \Rightarrow \bigwedge_{\underline{v} \in \mathrm{con}\alpha(P)} \left(\begin{array}{l} \exists\, I : \mathbb{R}_{\mathsf{oseq}} \bullet \mathrm{ran}(I) \subseteq \{ti \ldots ti'\} \\ \land \{ti, ti'\} \subseteq \mathrm{ran}(I) \land \\ \land (\forall\, n < \#I - 1 \bullet \\ \qquad \underline{v} \;\mathsf{cont\text{-}on}\; [I_n, I_{n+1})) \end{array} \right) \right)$$

where
$$\mathbb{R}_{\mathsf{oseq}} \triangleq \{x : \mathrm{seq}\,\mathbb{R} \mid \forall\, n < \#x - 1 \bullet x_n < x_{n+1}\}$$
$$f \;\mathsf{cont\text{-}on}\; [m, n) \triangleq \forall\, t \in [m, n) \bullet \lim_{x \to t} f(x) = f(t)$$

HCT1 states that time may only ever go forward, as should be the case, and thus the time interval is well-defined. ***HCT2*** states that every continuous variable $\underline{v}$ should be piecewise continuous, that is, that for non-empty intervals there exists a finite number of points (range of I) between ti and ti' where discontinuities occur. We define the set of totally ordered sequences $\mathbb{R}_{\mathsf{oseq}}$ that captures this set of discontinuities, and the continuity of f is defined in the usual way by requiring that at each point in $[ti, ti')$, the limit correctly predicts where the function goes.

HCT1 and ***HCT2*** are idempotent, monotone, and commutative as they are both conjunctive. We then have that ***HCT*** $=$ ***HCT2*** $\circ$ ***HCT1*** also satisfies all these properties. Furthermore it defines a complete lattice.

Theorem 1. ***HCT*** *predicates form a complete lattice under* $\bigsqcap$ *and* $\bigsqcup$*, with* $\top_H =$ ***HCT***(***true***) *and* $\bot_H =$ ***false****.*

Proof. By conjunctivity of ***HCT***. Properties of conjunctive healthiness conditions are proved in [12]. □

6 Hybrid relational calculus

The signature of our theory is given in Table 1. It consists of the standard operators of the alphabetised relational calculus together with operators to specify intervals $\llbracket P \rrbracket$, differential algebraic equations $\langle F_n \,|\, b \rangle$, and preemption $P\,[\,b\,]\,Q$. Using this calculus, we can describe the bouncing ball example from Fig. 2:

Example 1. Bouncing ball in hybrid relational calculus

$$h, v := 1, 0 \,;\, \left(\left\langle\, \dot{\underline{h}} = \underline{v};\; \dot{\underline{v}} = -9.81 \right\rangle [\,\underline{h} < 0\,]\, v := -v \cdot 0.8) \right)^{\omega}$$

This hybrid program has two continuous variables for height $\underline{h}$ and velocity $\underline{v}$. Initially we set these two variables to 1 and 0, and then initiate the system of ODEs. The system evolves until $\underline{h} < 0$, at which point a discrete command is executed that assigns $-v \cdot 0.8$ to v, that is, the velocity is reversed with a dampening factor. The system infinitely iterates, allowing the system dynamics to continue evolving, but with new initial values. Such a system only requires an ODE with no algebraic equations; to illustrate DAEs we give another example.

Example 2. Cartesian pendulum in hybrid relational calculus

$$\left\langle \underline{\dot{x}} = \underline{u};\ \underline{\dot{u}} = \underline{\lambda} \cdot \underline{x};\ \underline{\dot{y}} = \underline{v};\ \underline{\dot{v}} = \underline{\lambda} \cdot \underline{y} - 9.81 \,\middle|\, \underline{x}^2 + \underline{y}^2 = l^2 \right\rangle$$

This system consists of four differential and one algebraic equation in terms of the position (x, y), horizontal and vertical velocities u and v, and the length l of the pendulum cable. The differential equations describe the horizontal and vertical components of the pendulum's movement vector, governed by the laws of conservation of energy and gravity using a constant λ previously defined. The algebraic equation ties x and y together through the Pythagorean theorem, ensuring that the length of the cable must be respected by the movement. □

We note that many of the standard operators of the alphabetised relational calculus retain their standard denotational semantics [18] in this setting, but over the expanded alphabet. Indeed, an alphabetised relation is simply a hybrid relation with the degenerate alphabet $\mathrm{con}\alpha(P) = \emptyset$. 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 of P also defines the initial condition for Q as in the Z schema composition operator.

Similarly, other operators like the Kleene star and Omega iteration operators P^* and P^ω, being defined solely in terms of sequential composition, disjunction (internal choice), $I\!I$, and fixed point operators, also remain valid in this context. Thus we already have the core operators of an imperative programming language at our disposal. We prove that these core operators satisfy our two healthiness conditions in Isabelle (cf. section 7), but for now we state the following theorem.

Theorem 2. *The following operators of relational calculus $P \,;\, Q$, $P \lhd b \rhd Q$, P^*, $I\!I$, $x := v$, and **false** are **HCT** closed.*

The maximally nondeterministic relation ***true*** is of course not ***HCT*** healthy, and so we supplement our theory with $\textbf{\textit{true}}_H \triangleq \textbf{\textit{HCT}}(\textbf{\textit{true}})$. We define the interval operator from $\mathcal{DC}$ [33] and our own variant.

Definition 2. *Interval operators*

$$\begin{aligned} \lceil P \rceil &\triangleq \textbf{\textit{HCT2}}(\ell > 0 \wedge (\forall\, \underline{t} \in [ti, ti') \bullet P \mathbin{@} \underline{t})) \\ [\![P]\!] &\triangleq \lceil P \rceil \wedge \bigwedge_{\underline{v} \in \mathrm{con}\alpha(P)} (v = \underline{v}(ti) \wedge v' = \lim_{t \to ti'} (\underline{v}(t))) \wedge I\!I_{\mathrm{dis}\alpha(P)} \end{aligned}$$

$\lceil P \rceil$ is a continuous specification statement that P holds at every instant over all non-empty right-open intervals from ti to ti'; it corresponds to the standard $\mathcal{DC}$ operator. We apply ***HCT2*** to ensure that all variables are also piecewise continuous. In this setting we can use sequential composition $P \,;\, Q$ to express the $\mathcal{DC}$ chop operator $(P \circ Q)$ to decompose an interval. Our additional interval operator $[\![P]\!]$ pairs continuous variables with discrete variables at the start and limit of the interval, whilst holding other discrete variables constant. The initial

$$\lceil true \rceil = \ell > 0 \qquad \lceil false \rceil = \textit{\textbf{false}}$$
$$\lceil P \wedge Q \rceil = \lceil P \rceil \wedge \lceil Q \rceil \qquad \lceil P \vee Q \rceil \sqsubseteq \lceil P \rceil \vee \lceil Q \rceil$$
$$\llbracket P \rrbracket \sqsubseteq \llbracket P \rrbracket \,;\, \llbracket P \rrbracket$$

Table 2. Algebraic laws of durations

condition of each continuous variable $\underline{x}$ in the interval is constrained by the valuation of the corresponding discrete copy x. Likewise, the condition at the limit of the interval is recorded in the corresponding discrete after variable x'.

Crucially, this provides a uniform view of discrete and continuous variables when handled over an interval, and allows the use of standard relational operators for their manipulation. Moreover, by taking the limit rather than the final value of a continuous variable we do not constrain the trajectory valuation at ti' meaning it can be defined by a suitable discontinuous discrete assignment at this instant. Following [14] we ground our definition of differential equation systems in this interval operator. This will, for example, allow us to formally refine a DAE, under given initial conditions, to a suitable solution expressed using the interval operator. Intervals satisfy a number of standard laws of $\mathcal{DC}$ illustrated in Table 2, which we prove in section 7.

We next introduce an operator, adapted from $\mathcal{HCSP}$ [34,21], to describe the evolution of a system of differential-algebraic equations.

Definition 3. *DAE system in semi-explicit form*

$$\begin{aligned} &\langle\, \dot{\underline{v}}_1 = f_1;\ \cdots\ ;\ \dot{\underline{v}}_n = f_n \,|\, 0 = b_1;\ \cdots\ ;\ 0 = b_m \,\rangle \\ &\triangleq \llbracket (\forall\, i \in 1..n, \forall\, j \in 1..m \bullet\ \dot{\underline{v}}_i(\underline{t}) = f_i(\underline{t}, \underline{v}_1(\underline{t}), \cdots, \underline{v}_n(\underline{t}), \underline{w}_1(\underline{t}), \cdots, \underline{w}_m(\underline{t}))) \\ &\qquad\qquad\qquad\qquad \wedge\ 0 = b_j(\underline{t}, \underline{v}_1(\underline{t}), \cdots, \underline{v}_n(\underline{t}), \underline{w}_1(\underline{t}), \cdots, \underline{w}_m(\underline{t}))\rrbracket \end{aligned}$$

A DAE $\langle\, F_n \,|\, B_m \,\rangle$ consists of a set of n functions $f_i : \mathbb{R} \times \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$ each of which defines the derivative of variable $\underline{v}_i$ in terms of the independent time variable $\underline{t}$ and $n + m$ dependent variables. It also contains algebraic constraints $b_j : \mathbb{R} \times \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$ that must be invariant for any solution and do not refer to derivatives. For $m = 0$ the DAE corresponds to an ODE, which we write as $\langle\, F_n \,\rangle$. The DAE operator is defined using the interval operator to be all non-empty intervals over which a solution satisfying both the ODEs and algebraic constraint exists. Non-emptiness is important as it means that a DAE must make progress: it cannot simply take zero time since $\ell > 0$, and so a DAE cannot directly cause "chattering Zeno" effects when placed in the context of a loop, though normal Zeno effects remain a possibility.

As previously explained, at the initial time (ti) each continuous variable $\underline{v}_i$ of the system is equated to the value of the corresponding discrete input variable v_i. To obtain a well defined problem description, we require the following conditions to hold [2]: (i) the system of equations is consistent and neither underdetermined nor overdetermined; (ii) the discrete input variables v_i provide consistent initial conditions (ICs[7]); (iii) the equations are specific enough to define a unique solution during the interval ℓ. The system is then allowed to evolve from this point in

[7] Notice that in the general case ICs for DAE systems may actually involve derivatives $\dot{\underline{v}}_i$ of $\underline{v}_i$ [25]. Modelica supports the general case and sophisticated algorithms for

the interval between ti and ti' 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.

Finally, we define the preemption operator, adapted from $\mathcal{HCSP}$.

Definition 4. *Preemption operator*

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

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

7 Mechanisation in Isabelle/UTP

Our Isabelle [23] mechanisation serves two purposes: firstly it validates the model by enabling us to prove algebraic laws, and secondly it enables theorem proving for hybrid programs. It is based in a shallow embedding of the UTP[8], which provides direct proof automation through a combination of *Isabelle/Circus* [5] and our own deep model [10]. UTP relations are represented by predicates over bindings, and bindings over a given alphabet are represented using record types, where each field corresponds to a variable. The model is based on a UTP expression type $('a,\ '\alpha)$ *uexpr* ranging over alphabet type $'\alpha$ and with return type $'a$. Alphabetised predicates $'\alpha$ *upred* are expressions with a boolean return type, and relations are predicates over a product type $('\alpha \times '\beta)$ *upred*.

We mimic the syntax of UTP predicates as given in most standard publications (e.g. [18,4]). Where this is not possible, we supplement the same syntax with an added subscript u. For example, equality in Isabelle "=" denotes HOL equality, so we use $=_u$ for UTP equality. Input variable and output variable expressions are written $\$x$ and $\$x\acute{}$ respectively. We also make use of Isabelle's implementation of Cauchy real numbers and analysis [7,11]. Our proofs make heavy use of Isabelle's automated proof facilities like auto and sledgehammer [3]. This has allowed us to use Isabelle to validate the healthiness conditions and definitions given in the previous sections. We prove that they respect appropriate laws, which increases confidence in the correctness of our UTP theory.

finding consistent ICs from "guess" values exist [2,24]. However, numerical/symbolic methods for solving IVPs is not within the scope of our current work. Hence, we only consider less general ICs and presume that consistent ICs are provided.

[8] See https://github.com/isabelle-utp/utp-main/tree/shallow

This section has been compiled using Isabelle's document preparation system: all definitions and theorems have been mechanically verified[9].

record $('d,\ 'c)\ \mathit{hyst} =$
  $\mathit{state}_u :: 'd \times 'c$
  $\mathit{time}_u :: \mathit{real}$
  $\mathit{traj}_u :: \mathit{real} \Rightarrow 'c$

type-synonym $('d,\ 'c)\ \mathit{hyrel} = ('d,\ 'c)\ \mathit{hyst}\ \mathit{hrelation}$

A hybrid state $('d,\ 'c)\ \mathit{hyst}$ represents the alphabet, or equivalently the state of the hybrid relation, at a particular instant. We represent this using a record with three fields: state_u denoting the state variables, time_u denoting the time, and traj_u denoting the trajectory of continuous variables. The record type is parametrised by the discrete portion of the alphabet, denoted by type $'d$ and the continuous portion denoted by type $'c$. The state field's type is a product of the discrete and continuous state, whilst the trajectory refers only to the continuous state. Intuitively, this encodes the distinction between discrete and continuous variables. A hybrid relation is then a homogeneous relation (*hrelation*) over the hybrid state. We next give the healthiness conditions of our theory.

definition $\mathit{HCT1}(P) = (P \wedge \$\mathit{time} \geq_u 0 \wedge \$\mathit{time} \leq_u \$\mathit{time}')$

HCT1 is broadly the same as in section 6, though we additionally require that the initial time be no less than zero; this is due to our use of the standard type *real* that also encompasses negative numbers.

definition $\mathit{HCT2}(P) =$

$$(P \wedge (\$\mathit{time}' >_u \$\mathit{time} \Rightarrow (\exists\ I \cdot \{\$\mathit{time}, \$\mathit{time}'\}_u \subseteq_u \mathit{ran}_u(I) \wedge \mathit{ran}_u(I) \subseteq_u \{\$\mathit{time} .. \$\mathit{time}'\}_u \wedge (\forall\ n \cdot n <_u \#_u(I) - 1 \Rightarrow \$\mathit{traj}\ \mathit{cont{-}on}_u\ \{I(\!|n|\!)_u ..< I(\!|n+1|\!)_u\}_u) \wedge \mathit{sorted}_u(I) \wedge \mathit{distinct}_u(I))))$$

HCT2 also explicitly requires that the trajectory sequence I is both sorted and distinct, which equates to it being linearly sorted as required.

definition $\mathit{HTRAJ}(P) = (P \wedge \$\mathit{traj} =_u \$\mathit{traj}')$

We also have to add an auxiliary healthiness condition *HTRAJ*. This allows us to use standard HOL binary relations, where there is only inputs and outputs, to represent hybrid relations. Specifically, we have two copies of the trajectory, a before version and an after version and so this healthiness condition ensures the trajectory remains constant throughout. Monotonicity and idempotence of the healthiness conditions is proved by our automated relational calculus tactic.

With our healthiness conditions defined, we can proceed to define the operators. The basic operators, such as $I\!I$ and @ are elided here, and we instead focus on the continuous operators. We first define the two interval operators.

definition

[9] Our Isabelle/UTP theory development, including all omitted proofs, is available at http://www.cs.york.ac.uk/~simonf/utp2016.

$hInt\ P = HCT(\$time\acute{} >_u \$time \wedge (\forall\ t \in \{\$time\ ..< \$time\acute{}\}_u \cdot P \bullet_u t))$

Definition *hInt* corresponds to the interval operator $\lceil P \rceil$, and has an almost identical definition. In our mechanisation, an interval can be written as $\lceil P \rceil_H$ where P is a predicate with the time variable τ free.

definition

$hDisInt\ P = (hInt\ P \wedge \pi_1(\$state\acute{}) =_u \pi_1(\$state) \wedge \pi_2(\$state) =_u \$traj(\!|\$time|\!)_u$
$\wedge\ \pi_2(\$state\acute{}) =_u lim_u(x \to \$time\acute{}^-)(\$traj(\!|x|\!)_u))$

Our modified interval operator $\llbracket P \rrbracket$, represented here by *hDisInt* conjoins the standard interval operator with predicates that ensure that discrete variables remain const and and that continuous variable copies match the initial value in the trajectory, and the left limit of the trajectory at the end. Here π_n is a function that returns the nth element of a product, $f(\!|x|\!)_u$ represents function application, and $lim_u(x \to t^-)$ denotes the left-limit. This interval operator is written $\lceil|\ P\ |\rceil_H$, again with τ free.

Next we define the operators for ODEs and DAEs. The first step is to formally mechanise the notion of time derivatives ($\dot{x}$). Thus we define a predicate *hasDerivAt* that relates ODEs to solution functions using the lifting package [19].

type-synonym $'c\ ODE = real \times 'c \Rightarrow 'c$

lift-definition *hasDerivAt* ::
$(real \Rightarrow 'c :: real\text{-}normed\text{-}vector) \Rightarrow 'c\ ODE \Rightarrow real \Rightarrow ('a, 'b)\ relation$
$(\text{-}\ has{-}deriv\ \text{-}\ at\ \text{-}\ [90,\ 0,\ 91]\ 90)$
is $\lambda\ \mathcal{F}\ \mathcal{F}'\ \tau\ A.\ (\mathcal{F}\ has\text{-}vector\text{-}derivative\ (\mathcal{F}'\ (\tau, \mathcal{F}\ \tau)))\ (at\ \tau\ within\ \{0..\})$ **.**

An explicit system of ODEs ($'c\ ODE$) is encoded as a function $real \times 'c \Rightarrow 'c$, where the real is the time parameter, and $'c$ is a vector of real variables. We require that $'c$ be within the type class *real-normed-vector* of real vector spaces. Isabelle's Multivariate Analysis library contains a function *has-vector-derivative* that relates a solution function $\mathcal{F} : \mathbb{R} \to \mathbb{R}^n$ with its deriatives $\dot{\mathcal{F}} : \mathbb{R}^n$ at instant τ within a particular range. It represents the Fréchet derivative of differential equations in a vector space. We use this to define a construct $\mathcal{F}\ has{-}deriv\ \mathcal{F}'\ at\ \tau$ where $\mathcal{F}$ is a solution function, $\mathcal{F}'$ is the system of ODEs. This predicate is accompanied by a large number of rules that can be used to certify derivatives of polynomial functions. We now use these to encode operators for ODEs, DAEs, and ODEs under an initial condition.

definition $\langle \mathcal{F}' \rangle_H = (\exists\ \mathcal{F} \cdot \lceil|\ \mathcal{F}\ has{-}deriv\ \mathcal{F}'\ at\ \tau \wedge \&con\alpha =_u \mathcal{F}(\!|\tau|\!)_u\ |\rceil_H)$
definition $\langle \mathcal{F}' | B \rangle_H = (\langle \mathcal{F}' \rangle_H \wedge \lceil|B|\rceil_H)$
definition $\mathcal{I} \models \langle \mathcal{F}' \rangle_H = (\langle \mathcal{F}' \rangle_H \wedge \$traj(\!|\$time|\!)_u =_u \mathcal{I})$

We choose to implement ODEs and DAEs as separate constructs, as the definitions are simpler, though equivalent to those in the previous section. An ODE $\langle \mathcal{F}' \rangle_H$ specifies that a solution function $\mathcal{F}$ to the given ODE must exist and that at each point of the interval the values of all continuous variables ($con\alpha$) track this solution function. A DAE $\langle \mathcal{F}' | B \rangle_H$ is then simply an ODE constrained with the algebraic predicate throughout the interval. We also provide a representation

of ODEs as explicit initial value problems by $\mathcal{I} \models \langle \mathcal{F}' \rangle_H$ where $\mathcal{I}$ gives initial values to all continuous variables.

Finally, we prove some key laws about our hybrid relational calculus. Firstly we show that sequential composition is *HCT* closed, which partly validates our healthiness conditions with respect to the standard relational calculus. This is proved by an apply-style Isabelle proof which is omitted.

theorem *seq-r-HCT-closed*:
  assumes *P is HCT* **and** *Q is HCT*
  shows $(P \;;\; Q)$ *is HCT*
  by (*metis HCT-seq-r Healthy-def′ assms*(*1*) *assms*(*2*))

In order to demonstrate the use of ODEs in this framework, we take the ODE from the bouncing ball example, and show how its solution can be expressed as a refinement statement.

theorem *gravity-ode-refine*:
  $((v_0,\ h_0)_u \models \langle \lambda\ (t,\ v,\ h).\ (-\ g,\ v)\rangle_H \wedge \$time =_u 0) \sqsubseteq$
  $(\lceil| \ \&con\alpha =_u (v_0 - g{\cdot}\tau,\ v_0{\cdot}\tau - g{\cdot}(\tau{\cdot}\tau)\ /\ 2 + h_0)_u\ |\rceil_H \wedge \$time =_u 0)$
  by (*rel-tac* ; *rule exI* ; *auto* ; *vderiv-tac*)

As in Example 1, we specify the ODE with two variables, v and h that will give the velocity and height about the ground of the ball. We refine this in the window $time = 0$ as it makes the solution simpler via an appropriate conjunction. Given initial conditions of v_0 and h_0 for the respective variables, solutions to the ODE equations are $v_0 - g \cdot \tau$ and $(v_0 \cdot \tau - g \cdot \tau^2)/2 + h_0$, respectively. The solutions are proved correct in Isabelle automatically by application of our relational calculus tactic *rel-tac*, followed by existential introduction (*exI*) to introduce the ODE solution, application of the *auto* tactic, and then finally application of our own tactic vderiv-tac. This tactic recursively applies the set of introduction for differentiation in an effort to show that a given ODE is the derivative of a given solution. This example serves to demonstrate how a theorem prover can reason about differential equations in terms of their solution intervals making use of refinement and the Duration Calculus.

8 Modelica Semantics

In this section we give a semantics for flat Modelica whose models are given by a set of conditional differential, algebraic, and discrete equations. More specifically, we assume that a Modelica model consists of a set of dynamic variables x, algebraic variables y, and discrete variables q, and

- a set of $k \in \mathbb{N}_{>0}$ conditional DAEs, consisting of:
  - differential equations $\dot{x} = \mathcal{F}_i(x, y, q)$ for $i \in 1..k$;
  - algebraic equations $y = \mathcal{B}_i(x, y, q)$ for $i \in 1..k$;
  - boolean DAE guards $\mathcal{G}_i(x, y, q)$ for $i \in 1..k-1$, that give the conditions under which the corresponding set of differential and algebraic equations is active in terms of the values of discrete and continuous variables at

initialisation or the previous event. We assume that at least one set of equations is active at any time;

- a set of $l \in \mathbb{N}$ boolean event conditions $\mathcal{C}_i(x, y, q)$ for $i \in 1..l$, that trigger an event when changing value. These must be specified in terms of the core Modelica relational operators, namely $\leq$, $<$, $=$, and $\neq$;
- a set of $m \in \mathbb{N}$ conditional discrete equation blocks, consisting of:
  - n boolean discrete-event guards $\mathcal{H}_{i,j}(x, y, q, q_{pre})$ for $i \in 1..m$, $j \in 1..n$;
  - n discrete equations / algorithms $\mathcal{P}_{i,j}(x, y, q, q_{pre})$ for $i \in 1..m$, $j \in 1..n$. We assume the discrete equations are sorted into a suitable sequence.

Each conditional DAE describes a possible continuous behaviour using a collection of differential and algebraic equations. The particular behaviour to be executed is chosen based on the evaluation of the guards, which take as input the valuations of the discrete and continuous variables at the (re)start of the continuous evolution. The possible events that can occur are described by a collection of boolean event conditions, which act as guards that can stop the continuous evolution. Once one or more of these guards changes value an event is fired, and possible discrete behaviour is executed. Usually such guards are implemented in terms of a zero crossing function, though our semantics specifies them abstractly. The appropriate discrete behaviours are then chosen through a collection of discrete event guards, and the resulting behaviour by an appropriate discrete equation that may be specified by a suitable algorithm.

$$
\begin{aligned}
\mathcal{M} &= \mathsf{Init} \,;\, (\mathsf{DAE}\,[\,\mathsf{Events}\,]\,\mathsf{Discr})^{\omega} \\
\mathsf{Init} &= \underline{x}, \underline{y}, q := u, v, w \\
\mathsf{DAE} &= \left\langle\, \underline{x} = \mathcal{F}_1(\underline{\dot{x}}, \underline{y}, q) \,\middle|\, \mathcal{B}_1(\underline{x}, \underline{y}, q) \,\right\rangle \lhd \mathcal{G}_1 \rhd \cdots \\
&\quad \lhd \mathcal{G}_{n-1} \rhd \left\langle\, \underline{\dot{x}} = \mathcal{F}_n(\underline{x}, \underline{y}, q) \,\middle|\, \mathcal{B}_n(\underline{x}, \underline{y}, q) \,\right\rangle \\
\mathsf{Events} &= \bigvee_{i \in \{1..k\}} \mathcal{C}_i(\underline{x}, \underline{y}, q) \neq \mathcal{C}_i(x, y, q) \\
\mathsf{Discr} &= \textbf{\textit{var}}\ q_{pre} \bullet \\
&\quad \textbf{\textit{until}}\ q_{pre} = q\ \textbf{\textit{do}} \\
&\quad\quad q_{pre} := q \,; \\
&\quad\quad \mathcal{P}_{1,1}(\underline{x}, \underline{y}, q, q_{pre}) \lhd \mathcal{H}_{1,1}(\underline{x}, \underline{y}, q, q_{pre}) \rhd \mathcal{P}_{1,2}(\underline{x}, \underline{y}, q, q_{pre}) \lhd \cdots \,;\, \cdots \,; \\
&\quad\quad \mathcal{P}_{m,1}(\underline{x}, \underline{y}, q, q_{pre}) \lhd \mathcal{H}_{m,1}(\underline{x}, \underline{y}, q, q_{pre}) \rhd \mathcal{P}_{m,2}(\underline{x}, \underline{y}, q, q_{pre}) \lhd \cdots \,; \\
&\quad \textbf{\textit{od}}
\end{aligned}
$$

Fig. 3. Overall semantics of a Modelica model $\mathcal{M}$

We give the semantics for such a Modelica model $\mathcal{M}$, which is shown in Fig. 3, in terms of four main definitions.

Init denotes the initialisation phase of a Modelica model, where initial values are assigned to the discrete and continuous variables. For now, we assume that initial values u, v, and w can be unambiguously assigned to each. Following initialisation, an infinite loop is entered representing the main body of behaviour.

DAE denotes the conditional system of differential and algebraic equations active during the continuous evolution of the model. It is represented by a conditional predicate that selects an appropriate set of differential and algebraic equations based on initial values of discrete and continuous variables.

Events denotes the event preemption condition, and is a disjunction of all possible event conditions ("relations" in Modelica terminology) in the Modelica model. In this way, the DAE remains active until one of the event conditions changes from its initial value, at which point it is preempted.

Finally, Discr describes possible discrete behaviour to be executed during event iteration; a finite event loop adapted from the pseudo code given on page 263 of [22]. The initial value of all discrete variables is first copied by creation of a local variable q_{pre} that holds the initial value of q. Each conditional discrete equation is then evaluated, which may lead to updates to q, and then the procedure iterates. The event iteration terminates when no more updates to q are made: a fixed point is reached. In Modelica the existence of a fixed point is not guaranteed and event iteration can potentially lead to an infinite loop.

To illustrate, we use the bouncing ball Modelica example from Fig. 2. It has continuous variables representing the height of the ball above the ground h and the velocity of the ball v. For giving a semantics to this we convert the **when** expression to an **if** expression, so we need only consider semantics of the latter, using the conceptual mapping in section 8.3.5.1 of [22], which will yield:

```
c = h<0;
if (c and not(pre(c))) then
  reinit(v, -0.8*pre(v));
end if;
```

An additional variable `c` of type Boolean is added, and assigned the condition of the **when** statement. The **when** equation itself is replaced by an **if** equation whose condition is that `c` is true now, and was not true previously – i.e. it has become true at the current instant. We can now give the semantics of this model.

Example 3. Bouncing ball semantics in hybrid relational calculus

$$
\begin{array}{l}
h, v, c := 1, 0, \mathit{false} \ ; \\
\quad (\left\langle\, \underline{\dot{v}} = -9.81;\ \underline{\dot{h}} = \underline{v} \,\right\rangle \\
\qquad [(\underline{h} < 0) \neq (h < 0)] \\
\qquad \textbf{\textit{var}}\ c_{pre} \bullet \\
\qquad\quad \textbf{\textit{until}}\ (c_{pre} = c)\ \textbf{\textit{do}} \\
\qquad\qquad c_{pre} := c \ ;\ c := h < 0 \ ; \\
\qquad\qquad v := -0.8 \cdot v \ \lhd\ c \wedge \neg c_{pre} \ \rhd\ I\!I \\
\qquad\quad \textbf{\textit{od}})^{\omega}
\end{array}
$$

We assign initial values for the three variables, and assume that the condition c is false initially. The DAE is then activated and evolves until the valuation of the **if** guard $h < 0$ at time t is different from the initial value, that is $(\underline{h} < 0) \neq (h < 0)$. We note that $\underline{h}$ and h are two different variables: $\underline{h}$ denotes h at time t, whilst h

denotes its value at the beginning of the present DAE evolution, so the inequality corresponds to the value of this boolean guard changing. At this point, the event iteration begins. We create a variable to denote the previous value of c, and then enter into the event loop. We then assign c to c_{pre}, and evaluate the discrete equations. First of all, we evaluate the new value of c, which is the event condition. Secondly, if c is true and different from its previous value, we also update v, otherwise we skip. The loop terminates once the value of c has stablised (which it has in the second iteration). Following this, we iterate the whole loop and restart the DAE with the new initial values.

This example serves to illustrate the behaviour of a Modelica model in the hybrid relational calculus. Our preliminary semantics considers a fragment of the event handling mechanism, excluding practical problems of initialization and numerical integration of DAEs. Present limitations include the separation of continuous and discrete equations during the event handling mechanism. More complete Modelica semantics require to solve a *mixed* system of the discrete and continuous equations during events. We will consider these in future iterations of this semantics, define a more complete translation, and apply it to more substantive examples.

9 Conclusions

We have presented a denotational semantics for the dynamical systems modelling language Modelica, in terms of a hybrid relational calculus that has been mechanised in Isabelle. The semantics elaborates the event iteration system, showing how continuous evolution transitions to discrete behaviour and vice-versa. Nevertheless, our translation is currently relatively informal and thus in future work we will define a comprehensive mapping from Modelica to hybrid relations, including its expression language and collection of imperative language constructs. We will also combine our theory of hybrid relations with timed reactive designs [13] to provide a rich semantic model providing termination, stability, and concurrency in the form of CSP.

This work supports the goals of a large EU project called INTO-CPS[10], which aims at building an integrated tool-chain for model based development of Cyber-Physical Systems. This tool-chain will support the integration of heterogeneous discrete and continuous system models through the Functional Mockup Interface [8] (FMI), a language that allows the composition of continuous time and discrete event models, and their concurrent simulation to support empirical evaluation. We will use our UTP theory of hybrid relations combined with timed reactive designs to develop a common semantic domain into which all these language can be mapped and verified.

We also plan to further experiment with theorem proving in Isabelle, for example through a mechanisation of Hybrid Hoare Logic [37]. As stated in sec-

[10] *An Integrated Tool-chain for Model-based Design of Cyber-Physical Systems*. EU H2020 grant agreement 644047. http://into-cps.au.dk/

tion 8, Modelica does not guarantee that event iteration terminates and so we could use such a prover, in the context of reactive designs, to verify termination.

References

1. J. Åkesson, T. Ekman, and G. 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).
2. B. Bachmann, P. Aronsson, and P. Fritzson. Robust initialization of differential algebraic equation. In 5$^{\text{th}}$ *Intl. Modelica Conference*, Austria, September 2006.
3. 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.
4. A. Cavalcanti and J. Woodcock. A tutorial introduction to CSP in unifying theories of programming. In *Refinement Techniques in Software Engineering*, volume 3167 of *LNCS*, pages 220–268. Springer, 2006.
5. A. Feliachi, M.-C. Gaudel, and B. Wolff. Isabelle/Circus: a process specification and verification environment. In *VSTTE 2012*, volume 7152 of *LNCS*, pages 243–260. Springer, 2012.
6. 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.
7. 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.
8. FMI development group. Functional mock-up interface for model exchange and co-simulation, 2.0. https://www.fmi-standard.org, 2014.
9. 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.
10. S. Foster, F. Zeyda, and J. Woodcock. Isabelle/UTP: A mechanised theory engineering framework. In *Unifying Theories of Programming*, volume 8963 of *LNCS*, pages 21–41. Springer, 2014.
11. 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.
12. W. Harwood, A. Cavalcanti, and J. Woodcock. A theory of pointers for the UTP. In *Proc. 5th. Intl. Colloq. on Theoretical Aspects of Computing (ICTAC)*, volume 5160 of *LNCS*, pages 141–155. Springer, 2008.
13. 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.
14. J. 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.
15. J. He. HRML: a hybrid relational modelling language. In *IEEE International Conference on Software Quality, Reliability and Security (QRS 2015)*, August 2015.
16. T. A. Henzinger. *The theory of hybrid automata*, pages 278–292. IEEE, 1996.
17. T. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, Englewood Cliffs, New Jersey 07632, 1985.

18. T. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall, 1998.
19. 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.
20. D. Kågedal and P. Fritzson. Generating a Modelica compiler from natural semantics specifications. In *Proc. 1998 Summer Computer Simulation Conference (SCSC'98)*, 1998.
21. 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.
22. Modelica Association. Modelica - A Unified Object-Oriented Language for Systems Modeling - Version 3.3 Revision 1. Standard Specification, July 2014.
23. T. Nipkow, M. Wenzel, and L. C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
24. L. A. Ochel and B. Bachmann. Initialization of Equation-Based Hybrid Models within OpenModelica. In 5th *Intl. Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*, pages 97–103, Nottingham, UK, April 2013.
25. C. Pantelides. The consistent initialization of differential-algebraic systems. *SIAM Journal on Scientific and Statistical Computing*, 9(2):213–231, 1988.
26. J. Perna and J. Woodcock. UTP Semantics for Handel-C. In *Unifying Theories of Programming*, volume 5713 of *LNCS*, pages 142–160. Springer, 2010.
27. A. Platzer. *Logical Analysis of Hybrid Systems*. Springer, 2010.
28. L. Satabin, J. Colaço, O. Andrieu, and B. Pagano. Towards a Formalized Modelica Subset. In 11th *Int. Modelica Conference*, September 2015.
29. B. Thiele, A. Knoll, and P. Fritzson. Towards Qualifiable Code Generation from a Clocked Synchronous Subset of Modelica. *Modeling, Identification and Control*, 36(1):23–52, 2015.
30. K. Wei, J. Woodcock, and A. Cavalcanti. Circus Time with Reactive Designs. In *Unifying Theories of Programming*, volume 7681 of *LNCS*, pages 68–87. Springer, 2013.
31. 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 *Proc. 19th Intl. Symp. on Formal Methods (FM)*, volume 8442 of *LNCS*, pages 733–748. Springer, 2014.
32. C. Zhou and M. R. Hansen. Chopping a point. In *Proc. 7th BCS-FACS Refinement Workshop*. Springer, 1996.
33. C. Zhou, C. A. R. Hoare, and A. P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1991.
34. 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.
35. 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.
36. H. Zhu, F. Yang, and J. He. Generating denotational semantics from algebraic semantics for event-driven system-level language. In S. Qin, editor, *Unifying Theories of Programming*, volume 6445 of *LNCS*, pages 286–308. Springer, 2010.
37. 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.