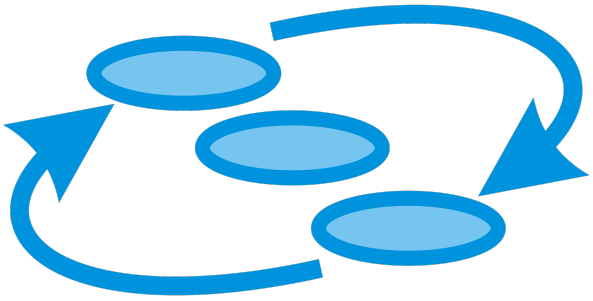
Grant Agreement: 644047

INtegrated TOol chain for model-based design of CPSs



INTO-CPS

Initial Semantics of Modelica

Technical Note Number: D2.2c

Version: 1.1

Date: December 2016

Public Document

http://into-cps.au.dk

Contributors:

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

Editors:

Simon Foster, UY

Reviewers:

Hassan Ridouane, 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	10-06-2016	Simon Foster	Initial document version
0.2	20-10-2016	Simon Foster	Nearly final version
1.0	29-10-2016	Simon Foster	Final draft for review
1.1	05-12-2016	Simon Foster	Final version for submission

Abstract

This deliverable reports on our work towards providing a continuous-time semantics for the dynamical systems modelling language, Modelica, in the context of Hoare and He's Unifying Theories of Programming (UTP) as a basis to reason about FMI simulations. Modelica is a language for modelling a system's continuous behaviour using a combination of differential-algebraic equations and an event-handling system. Inspired by Hybrid CSP and Duration Calculus, we develop a novel UTP theory of hybrid relations 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. We then 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. We demonstrate this by showing how our hybrid relational calculus can be integrated with the theory of reactive processes.

Contents

Acronyms

.

DAE	differential algebraic equation.
EOO	equation-based object-oriented.
FMI	Functional Mock-up Interface.
FMU	Functional Mock-up Unit.
IC	initial condition.
IVP	initial value problem.
LHS	left-hand side.
MLS	Modelica Language Specification.
ODE	ordinary differential equation.
RHS	right-hand side.
UTP	Unifying Theories of Programming.

1 Introduction

INTO-CPS multi-models are composed of models whose foundations lie in a variety of modelling notations, each of which has its own unique syntax, semantics, and underlying paradigmatic concepts, such as discrete or continuous time. The purpose of a multi-model is assign behaviour to a Cyber-Physical System (CPS) by composing the behaviours of the constituent models. Thus, in order to provide an integrated tool chain for trustworthy CPS development, there is a necessity for unification of these underlying semantic models to allow consistent integration of heterogeneous system components. This will then allow us to substantiate statements made about the multi-model with respect to the underlying mathematical core. Hoare and He's Unifying Theories of Programming [30] (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. In this deliverable we leverage the UTP to provide the foundations for continuous-time modelling in the INTO-CPS tool chain.

Modelling of continuous dynamical systems in the INTO-CPS tool chain is provided by the Modelica and 20-sim tools, both of which are based on differential equations. In the first year we provided a UTP theory of differential equations in the form of the hybrid relational calculus in Deliverable D2.1c [19], a minimal language to allow the specification of sequential hybrid systems with discrete behaviour and differential equations [14]. In this deliverable we provide an updated version of the calculus, with a new healthiness condition to account for piecewise continuous functions and appropriately updated operator definitions. We then use this new version of the calculus to give a continuous-time semantics for Modelica, which is the main objective of the deliverable. Our new calculus solves the problem we previously highlighted in Section 4.4 of [19] that two variable assignments at the same instance can induce errant behaviour. We do this by relaxing the requirement that such discrete behaviour is immediately reflected in the trajectory; in the new calculus the latter only happens when continuous behaviour occurs, which we do by dropping the previous version of ***HCT2*** as a healthiness condition. This key result in particular allows the integration of discrete relational behaviour with continuous evolution.

As previously highlighted in D2.1b [18] and D2.1c [19], our overall approach to giving formal semantics is through the description of a *lingua franca* for INTO-CPS to which we have given the name *CyPhyCircus*. We will then define mappings from the core languages in *CyPhyCircus* which, as illustrated Figure 1, will also enable access to a number of static analysis tools and techniques, such as model checking [42, 3, 4] and theorem proving [15, 16, 53]. *CyPhyCircus* will build on the existing work of the ***Circus*** language family [50, 41, 51], a suite of formal languages that combines rich state modelling (like as in the Z specification language [52]) with concurrency (like as in CSP [29]), with various other programming paradigms such as object orientation [7] and discrete real-time modelling [49]. Our intention is to create a language that combines rich-state modelling, concurrent reactive processes, real-time modelling, continuous variables, and differential equations. The theory of hybrid relations which we describe in this deliverable provides the foundations for such hybrid dynamical behaviour in *CyPhyCircus*.

Modelica [36] is a widely used language for description and modelling of hybrid dynamical systems that compose a continuously evolving physical plant with a discrete controller.

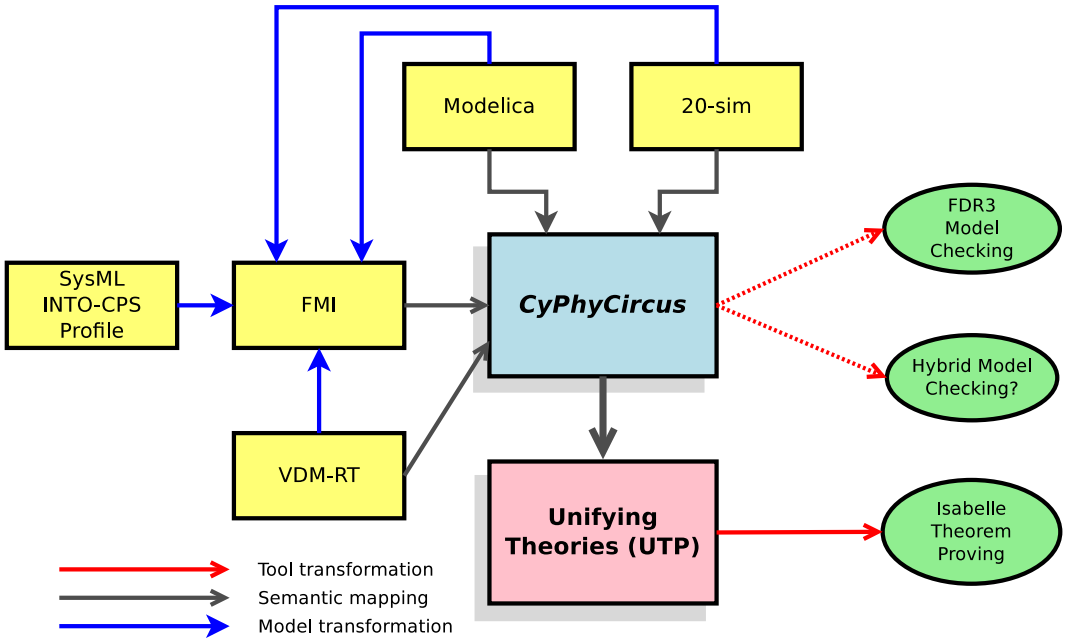


Figure 1: *CyPhyCircus* as the INTO-CPS lingua franca

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[1], Wolfram SystemModeler[2], MapleSim[3] and the open-source implementation, OpenModelica[4]. 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. Moreover, since we focus on a general theory of hybrid relations, it is possible to also treat other languages like 20-sim and Simulink in this context.

Our approach to giving a semantics to Modelica is three-fold [14]. Firstly, we create the UTP theory of hybrid relations, building on the work of He [26, 27], Zhou [56, 55], Zhan [35], 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 Hybrid CSP [57] ($\mathcal{HCSP}$), but extended with an interval operator [56] 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 [56], though the semantics is purely relational. Moreover, we provide a uniform account of both discrete

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

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 been mechanised in our UTP proof assistant, *Isabelle/UTP* [15], 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. More advanced concepts like blocks are not directly considered and are left as future work for the final year.

In Section 2 we outline related work for our theory of hybrid relations. In Section 3 we give a brief overview of the UTP, and in Section 4 we give a brief overview of the Modelica language (for more details, please see D2.1c [19]). In Section 5 we review our UTP theory of hybrid relations, and in Section 6 we use this theory to define the hybrid relational calculus, which is finally mechanised in Isabelle/UTP in Section 7. Afterwards, in Section 8 we use the hybrid relational calculus to give an initial semantics for hybrid differential-algebraic equations, the core semantic domain of Modelica. Finally, in Section 9 we show how our hybrid relational calculus can be integrated with reactive processes to create hybrid reactive designs, which are the core semantic model for *CyPhyCircus*. Section 10 concludes this deliverable, outlining the next steps for our work.

2 Related Work: Hybrid Systems

This section provides a brief overview of related works on formal semantics for hybrid systems. It thus provides the theoretical context for the UTP theory we will develop in Section 5 to give a semantics to Modelica and hybrid computation in general. The majority of the work on hybrid systems takes inspiration from Hybrid Automata [28], 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 [28] $\mathbb{R} \to \mathbb{R}^n$, 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 [44]. 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* [44] 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 [56] 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 [26, 57] ($\mathcal{HCSP}$), an extension of Hoare's process calculus CSP [29] that adds support for continuous variables as described by differential equations and modelled by standard trajectories, in a similar manner to hybrid automata. $\mathcal{HCSP}$ [26] 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 [30].

Further work on $\mathcal{HCSP}$ [57] 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 [58], 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 [35], 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 – which give the valuation of the variable at the end of an evolution – by explicitly considering left-limits, which is important for Modelica event iteration.

A theorem prover for $\mathcal{HCSP}$ called, HHL Prover [59], has also been developed and applied to verification of Simulink diagrams through a mapping into $\mathcal{HCSP}$ [54]. More recently the fundamentals of hybrid system modelling have been studied in a purely UTP relational setting [27]. This work has produced a language called the Hybrid Relational Modelling Language [27] (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 [56] ($\mathcal{DC}$) provides specification of invariants over the continuous time domain, in order to facilitate the verification of 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 [58], which is then used to give semantics to $\mathcal{HCSP}$ [57]. $\mathcal{DC}$ 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. In contrast to $\mathcal{DC}$, we provide a purely relational UTP semantics, and also explictly distinguish continuous and discrete variables, instead

$$
\begin{aligned}
x := v &\triangleq x' = v \land y' = y \\
P \,;\, Q &\triangleq \exists\, x_0 \bullet P[x_0/x'] \land Q[x_0/x] \\
P \lhd b \rhd Q &\triangleq (b \land P) \lor (\neg b \land Q) \\
P^* &\triangleq \nu X \bullet P \,;\, X
\end{aligned}
$$

Table 2: UTP programs-as-predicates

of modelling the latter as step functions. This distinction allows us to retain standard relational definitions of the majority of discrete UTP operators.

Relational predicative semantics for real-time programs has also previously been studied by Hayes and others [22, 23, 25], mainly in the context of the real-time refinement calculus [24]. The real-time refinement calculus is an extension of refinement calculus [37] which in addition to enabling specification of pre-/postcondition style specifications, also allows the specification of real-time constraints like deadlines. The model given for real-time refinement calculus is a form of *timed trace*, which are partial functions from $\mathbb{R}_{\geq 0} \nrightarrow \Sigma$, where Σ is a type denoting the system's state. Such a timed trace assigns to a subset of the time instants in $\mathbb{R}_{\geq 0}$ updates to the state that occur at that instant. In this way, timed traces can be used to give a semantics to continuous and hybrid systems. We will adapt this model for a refinement of our hybrid relational theory in Section 9.

3 Unifying Theories of Programming

In this section we briefly introduce the UTP semantic framework which we use to describe of theory of hybrid relations, and thereafter to give a denotational semantics to Modelica. More background on the UTP can be found in our sister deliverable [17] and the UTP tutorial [6]. Unifying Theories of Programming [30, 6] (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 made up of logical formulae. 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 [15].

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), skip II_A, and iteration P^*, all of which are given predicative interpretations as illustrated in Table 2.

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 $C \triangleq clock = 3 \land clock' = 1$ 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 $\mathbf{HT}(P) = P \land clock \leq clock'$, 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 $\mathbf{false}$ 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 [13]. 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

In this section we give background to the Modelica language, which we will give a semantics to. Modelica is an equation-based object-oriented language for describing the dynamic behaviour of CPS, standardised by the Modelica Language Specification (MLS) [36]. The MLS is described using English; therefore, its semantics is to some extent subject to interpretation. Quoting from the MLS [36, 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 2 illustrates the basic idea. The squiggle arrow denotes a degree of fuzziness — a simulation result is an *approximation* to the inaccessible (in general) exact solution of the equation system. The specification does not prescribe a particular solution approach. A classical model for a hybrid system 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

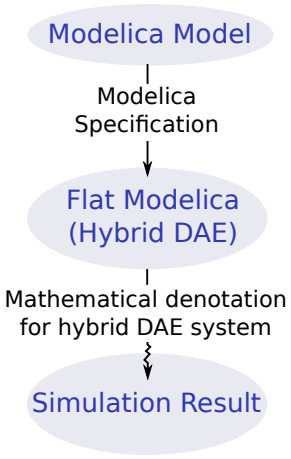


Figure 2: From model to simulation

```
1  model BouncingBall
2    Real h; Real v;
3  initial equation
4    h = 1.0;
5  equation
6    v = der(h);
7    der(v) = -9.81;
8    when h<0 then
9      reinit(v, -0.8*pre(v));
10   end when;
11 end BouncingBall;
```

Figure 3: Bouncing ball in Modelica.

initial height of $h = 1\,\mathrm{m}$ is given in Figure 3. When the ball hits the ground, it changes its velocity v discontinuously and bounces back. `der(h)` and `der(v)` (lines 6 and 7) 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 (when the ball bounces) is modelled using a conditionally activated reinitialization equation (lines 8-10). 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 (multiplied by the impact coefficient), just before condition $h < 0$ becomes true (where `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 Figure 2) [33, 1, 46] while others are restricted to discrete-time language subsets [48].

Flat Modelica can be conceptually mapped to a set of differential, algebraic and discrete equations of the following form [36, 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; $m_{\mathrm{pre}}(t_e)$ 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:

$$m(t_e) := f_m\big(x(t_e), \dot{x}(t_e), y(t_e), m_{\text{pre}}(t_e), p, c(t_e)\big) \tag{2}$$
$$c(t_e) := f_e\big(m^{\mathbf{B}}(t_e), m^{\mathbf{B}}_{\text{pre}}(t_e), p^{\mathbf{B}}, rel(v(t_e))\big) \tag{3}$$

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; $rel(v(t_e)) = rel([x(t);\ \dot{x}(t);\ y(t);\ t;\ m(t_e);\ m_{\text{pre}}(t_e);\ p])$ is a Boolean-typed vector-valued function containing 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, which will be used in Section 8 to give a denotational semantics to Modelica. 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 – real-valued functions representing continuous evolution – 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 could, for instance, be embedded into timed reactive designs [25, 49].

5.1 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 $\mathcal{DC}$ [58]. 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 variables in$\alpha(P)$, and output variables, out$\alpha(P)$. Inspired by [27], 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 the discrete program variables, that is $\mathrm{con}\alpha(P) \cap (\mathrm{in}\alpha(P) \cup \mathrm{out}\alpha(P)) = \emptyset$. 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 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. $\alpha(P) = \mathrm{in}\alpha(P) \cup \mathrm{out}\alpha(P) \cup \mathrm{con}\alpha(P)$, is called a *hybrid relation*.

Continuous variables come in two varieties that allow 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 sequential composition). 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:

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

We introduce the following @ operator borrowed from [11] that lifts a predicate in instant variables to one in trajectory variables.

Definition 5.1 (Continuous variable lifting)

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

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.

5.2 Healthiness conditions

We introduce two healthiness conditions for hybrid relations:

Definition 5.2 (Healthiness conditions)

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

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

where

$$\mathbb{R}_{\mathsf{oseq}} \triangleq \{x : \mathrm{seq}\,\mathbb{R} \mid \forall\, n < \#x - 1 \bullet x_n < x_{n+1}\}$$

$$f\ \textit{cont-on}\, [m, n) \triangleq \forall\, t \in [m, n) \bullet \lim_{x \to t} f(x) = f(t)$$

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 $\underline{x}$ 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.

The healthiness conditions differ from those presented in D2.1c [19], in that we have added piecewise continuity, and have dropped the requirement that each continuous variable $\underline{v}$ is always tracked in its before and after variables (v and v'). Specifically, it need not always be the case that $v = \underline{v}(ti)$ and $v' = \underline{v}(ti')$ because this disallows the behaviour where v' can vary with respect to the final valuation of $\underline{v}$. This would mean, for example, that we could not make instantaneous assignments to variables without producing contradictory, and thus miraculous, predicates. Instead, this invariant is now imposed only within continuous evolution operators, as will be seen in Section 6.

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

Proof 5.1 *By conjunctivity of* ***HCT****. Properties of conjunctive healthiness conditions are proved in [21].*

6 Hybrid Relational Calculus

In this section we use the theory of hybrid relations defined in Section 5 to define the hybrid relational calculus. The signature of our theory is given in Figure 4. It consists of the standard operators of the alphabetised relational calculus together with operators to specify intervals $[\![P]\!]$, differential algebraic equations $\langle\, F_n \mid b \,\rangle$, and preemption $P\,[\,b\,]\,Q$. Using this calculus, we can describe the bouncing ball example from Figure 3:

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

Figure 4: Signature of hybrid relational calculus

Example 6.1 *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 6.2 *Cartesian pendulum in hybrid relational calculus*

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

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 [30] 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), $\sqcap\!\!\sqcap$, 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 6.1 *The following operators of relational calculus $P \,;\, Q$, $P \lhd b \rhd Q$, P^*, $\sqcap\!\!\sqcap$, $x := v$, and* ***false*** *are* ***HCT*** *closed.*

$$\begin{array}{rclcrcl} \lceil true \rceil & = & \ell > 0 & & \lceil false \rceil & = & \textbf{\textit{false}} \\ \lceil P \wedge Q \rceil & = & \lceil P \rceil \wedge \lceil Q \rceil & & \lceil P \vee Q \rceil & \sqsubseteq & \lceil P \rceil \vee \lceil Q \rceil \end{array}$$
$$\llbracket P \rrbracket \sqsubseteq \llbracket P \rrbracket \; ; \; \llbracket P \rrbracket$$

Table 3: Algebraic laws of durations

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 first define the interval operator from $\mathcal{DC}$ [56] and then our own variant.

Definition 6.1 *Interval operators*

$$\begin{array}{rcl} \lceil P \rceil & \triangleq & \textbf{\textit{HCT2}}(\ell > 0 \wedge (\forall \underline{t} \in [\mathrm{ti}, \mathrm{ti}') \bullet P \mathbin{@} \underline{t})) \\ \llbracket P \rrbracket & \triangleq & \lceil P \rceil \wedge \bigwedge_{\underline{v} \in \mathrm{con}\alpha(P)} (v = \underline{v}(\mathrm{ti}) \wedge v' = \lim_{t \to \mathrm{ti}'} (\underline{v}(t))) \wedge \mathit{II}_{\mathrm{dis}\alpha(P)} \end{array}$$

$\lceil P \rceil$ 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 $\llbracket P \rrbracket$ pairs continuous variables with discrete variables at the start and limit of the interval via a coupling invariant, whilst holding other discrete variables constant. The initial 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'. In D2.1c [19] this invariant was imposed universally, but here we have localised it to only continuous operators defined in terms of $\llbracket P \rrbracket$.

Crucially, this construction 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 (in contrast to D2.1c [19]) we do not constrain the trajectory valuation at ti' meaning it can be defined by a suitable discontinuous discrete assignment at this instant. Following [26] 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 3, which we prove in Section 7.

We next introduce an operator, adapted from $\mathcal{HCSP}$ [57, 35], to describe the evolution of a system of differential-algebraic equations.

Definition 6.2 *DAE system in semi-explicit form*

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

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" [34] effects when placed in the context of a loop, though normal Zeno effects remain a possibility. Chattering Zeno refers to the situation when a system makes no progress in time, but is engaged in an infinitely long internal computation. It thus differs from normal Zeno which makes progress though less and less at each step.

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]:

1. the system of equations is consistent and neither underdetermined nor overdetermined;
2. the discrete input variables v_i provide consistent initial conditions (ICs[5]);
3. 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 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 6.3 *Preemption operator*

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

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.

[5]Notice that in the general case ICs for DAE systems may actually involve derivatives $\dot{\underline{v}}_i$ of $\underline{v}_i$ [43]. Modelica supports the general case and sophisticated algorithms for finding consistent ICs from "guess" values exist [2, 39]. However, numerical/symbolic methods for solving Initial Value Problems (IVPs) is not within the scope of our current work. Hence, we only consider less general ICs and presume that consistent ICs are provided.

7 Mechanisation in Isabelle/UTP

Our Isabelle [38] 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[6], which provides direct proof automation through a combination of *Isabelle/Circus* [10] and our own deep model [15]. 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. [30, 6]). 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 $\$x'$ respectively. We also make use of Isabelle's implementation of Cauchy real numbers and analysis [12, 20]. Our proofs make heavy use of Isabelle's automated proof facilities like auto and sledgehammer [5]. 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. This section has been compiled using Isabelle's document preparation system: all definitions and theorems have been mechanically verified[7].

record $('d, 'c)$ *hyst* =
  $state_u$ :: $'d \times 'c$
  $time_u$ :: *real*
  $traj_u$ :: *real* $\Rightarrow$ $'c$

type-synonym $('d, 'c)$ *hyrel* = $('d, 'c)$ *hyst hrelation*

A hybrid state $('d, 'c)$ *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 $HCT1(P) = (P \wedge \$time \geq_u 0 \wedge \$time \leq_u \$time')$

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 $HCT2(P) =$

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

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

$$(P \wedge (\$time' >_u \$time \Rightarrow (\exists\ I \cdot \{\$time, \$time'\}_u \subseteq_u ran_u(I) \wedge ran_u(I) \subseteq_u \{\$time\ ..\ \$time'\}_u \wedge (\forall\ n \cdot n <_u \#_u(I) - 1 \Rightarrow \$traj\ cont{-}on_u\ \{I(\!|n|\!)_u\ ..<\ I(\!|n+1|\!)_u\}_u) \wedge sorted_u(I) \wedge distinct_u(I))))$$

HCT2 also explicitly requires that the trajectory sequence I is both sorted and distinct, which equates to it being linearly sorted as required by Definition 5.2.

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

We also have to add an auxiliary healthiness condition *HTRAJ*. This allows us to use standard HOL binary relations, where there are 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 $\amalg$ and @ are elided here, and we instead focus on the continuous operators. We first define the two interval operators.

definition
$hInt\ P = HCT(\$time' >_u \$time \wedge (\forall\ t \in \{\$time\ ..<\ \$time'\}_u \cdot P \bullet_u t))$

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

Our modified interval operator $\llbracket P \rrbracket$, represented here by *hDisInt* conjoins the standard interval operator with predicates that ensure that discrete variables remain constant 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 $lim_u(x \rightarrow t^-)$ 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 [32].

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

lift-definition *hasDerivAt* ::
$(real \Rightarrow 'c :: real\text{-}normed\text{-}vector) \Rightarrow 'c\ ODE \Rightarrow real \Rightarrow ('a, 'b)\ relation$
$(-\ has{-}deriv\ -\ at\ -\ [90, 0, 91]\ 90)$
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* τ 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 6.1, 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 Semantics of Hybrid DAEs

In this section we give a semantics for flat Modelica whose models are given by a set of conditional differential, algebraic, and discrete equations, in terms of hybrid relational calculus.. More specifically, we assume that a Modelica model consists of:

- a set of dynamic variables x;
- algebraic variables y;
- discrete variables q;
- 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.

We give the semantics for such a Modelica model $\mathcal{M}$, which is shown in Figure 5, 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.

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

Figure 5: Overall semantics of a Modelica model $\mathcal{M}$

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 [36]. 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 Figure 3. 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 [36], 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 8.1 *Bouncing ball semantics in hybrid relational calculus*

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

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 Hybrid Reactive Designs

In this section we will show how the operators of the hybrid relational calculus can be integrated with reactive processes [6] in order to allow the expression of concurrent hybrid systems. In the previous section we showed how hybrid relations can be used to give a semantics to Modelica, and so this semantic integration will further allow composition of Modelica models with models described using (timed) reactive processes such as VDM-RT [17], and thus provide the basis for formal characterisation of FMI networks (see also Deliverable D2.2d [8]).

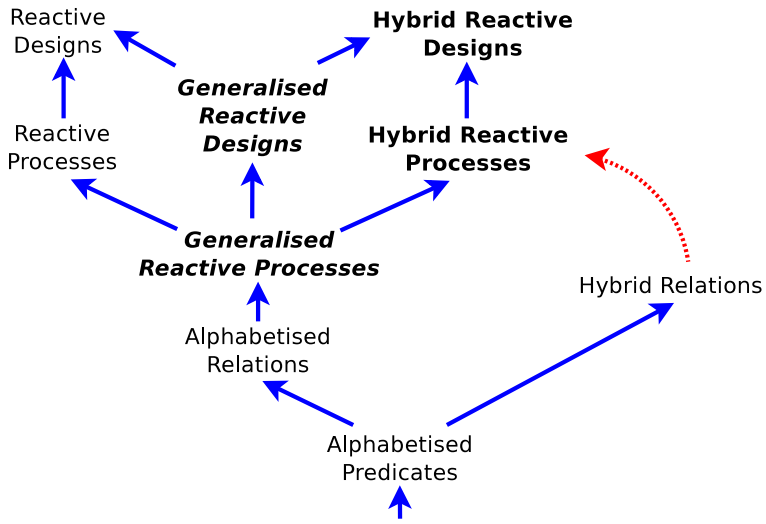


Figure 6: UTP hybrid relation hierarchy

We will achieve the integration through a new denotational model that replaces the CSP-style sequence based trace model with a continuous time trajectory representation that draws inspiration from the works of Hayes [23, 25] and Höfner [31]. For details of UTP reactive processes, please see the *"Reactive Designs"* section in Deliverable D2.2b [17]. This new model will allow us to embed both CSP-style events and continuous variables into the trace, which will allow us to embed both the operators of CSP and the hybrid relational calculus. We achieve this through the creation of the UTP theory hierarchy illustrated in Figure 6, which we will explain in the following sections.

We give a summary of this work in progress, which will be fully brought to fruition in the final year of INTO-CPS to give a semantics to *CyPhyCircus* and thence the language of the INTO-CPS toolchain. These preliminary results have been mechanised in Isabelle/UTP[8].

9.1 Relations and Timed Reactive Programs

The model of hybrid relations given in Section 5, whilst an accurate domain for expressing hybrid behaviour and sufficient to express the semantics of Modelica (see Section 8), imposes some obstacles to integration with other UTP theories and also with practical reasoning. This is mainly because the theory, unlike most program models in the UTP, is not purely relational, but encapsulates continuous variables that are orthogonal to both input and output variables. This is not a problem for the UTP, since we are simply extending alphabetised predicates rather than relations as in the right most arm of Figure 6. However, it does mean that, when using hybrid relations, one has to reason about trajectories using different laws to those of the usual relational calculus.

For example, given a continuous variable $\underline{x}$ in a relational composition $P \; ; \; Q$, it is not easy to subdivide the overall trajectory into the part contributed by P and that contributed by Q. This is because continuous variables are modelled as total functions. This ensures

[8]See in particular github.com/isabelle-utp/utp-main/blob/master/utp/utp_trd.thy

that the relational operators work as expected, but makes it difficult to decompose the behaviour of the system.

The super-relational nature of hybrid relations also complicates mechanical reasoning. As seen in Section 7, to make use of HOL's built-in relational calculus it is necessary to model trajectories as normal relational variables and then use the extra healthiness condition ***HTRAJ*** to ensure they do not change. Although this works, it complicates reasoning.

Another issue that must be addressed is specifically integration of hybrid relations with reactive processes and reactive designs, the general UTP theories for concurrent reactive programs. Reactive programs, in addition to modelling sequential behaviour, also enable interaction with their environment through abstract communication events. A reactive process specifies its behaviour in terms of observational variables $tr, tr' : \text{seq } Event$ that record the sequence of events (the trace) before and after execution, $wait, wait' : \mathbb{B}$ that describe whether a process is waiting for interaction or otherwise in an intermediate state, and $ref' : \mathbb{P}\, Event$ that records the set of events refused by a process after a given trace (as in the failures model of CSP [29, 45]).

Reactive designs are a specialisation of reactive process of the form $\mathbf{R}(P \vdash Q)$, where $\mathbf{R} \triangleq \mathbf{R3} \circ \mathbf{R2} \circ \mathbf{R1}$ encapsulates the three reactive healthiness conditions. They are specified in terms of an assumption P and commitment Q, both of which are used to constrain the reactive behaviour of the system in terms of tr, tr', and ref'. Reactive designs, in particular, are used to provide a UTP model for CSP [6].

Our sister deliverable, D2.2b [17], shows how to give a semantics to VDM-RT using timed reactive designs — a form of reactive design with support for modelling discrete time — in the form of the ***CML*** language [51]. So, integrations of hybrid relations and timed reactive processes is the core result needed to give semantics to co-models written using VDM-RT and Modelica. Clearly, if we are to augment continuous-time modelling with reactive behaviour, we need to combine the underlying theories to create hybrid reactive designs, a model for concurrent hybrid programs. In particular this needs to consider the combination of timed reactive events and continuous evolution, both of which progress with respect to time.

Timed reactive designs further subdivide the trace into a number of equal time periods, during which a sequence of events can occur. The original model of timed reactive processes [47] replaces tr, tr', and ref with $tr_t, tr'_t : \text{seq}^+(\text{seq } Event \times \mathbb{P}\, Event)$ which record a non-empty sequence of time instants, each of which contains a sequence of events and a set of refused events. The interpretation of the time instant can be any non-zero period of time, for example, a millisecond or nanosecond, and gives the maximum granularity for measuring time separation of events. A similar model of discrete time, called timed reactive designs, is used to give a semantics to ***CML***. Timed reactive designs use the standard tr and tr' variables, but there exists a distinguished event constructor $tock : \mathbb{P}\, Event \to Event$ that takes as a parameter a set R of events and provides an event that records the passage of one unit of time, at the end of which R was refused.

When it comes to continuous-time reactive processes, a number of alternatives exist, which we have considered. Timed CSP [9], for example, uses a model which records a time stamp, of type $\mathbb{R}_{\geq 0}$, alongside each event. Hybrid CSP [26, 57] ($\mathcal{HCSP}$) changes the observational tr to have type $\mathbb{R}_{\geq 0} \to \text{seq } Event$, such that at each instant a sequence

of events can be recorded, though with the restriction that tr takes the value of $\langle\rangle$ for all but a finite number of instants, thus ensuring the trace of events is finite. Clearly tr here can be seen as a continuous variable, a fact which is readily used in the Duration Calculus semantics [57] of $\mathcal{HCSP}$.

Within in the context of the UTP, continuous-time behaviour has been considered by Hayes in his version of the timed reactive design model [25]. Timed reactive designs model the time dependent behaviour of a system using a partial function $\sigma : \mathbb{T} \nrightarrow \Sigma$, for some suitable time domain $\mathbb{T}$ and model of the state Σ. A timed reactive design relates an initial trace σ to a final trace σ', encapsulating the changes to the state variables. There is also the requirement that $\sigma \subseteq \sigma'$, so that the timed trace can only be extended, analogous to healthiness condition ***R1***.

Similar to Hayes' work, Höfner derives an algebra for hybrid systems [31], which includes a model of continuous-time traces. He also defines a number of algebraic operators on timed traces, such as composition, which are then lifted to Kleene algebra operators to describe the behaviour hybrid programs and automata. This allows him to construct an algebraic verification technique for hybrid systems. The works of Hayes and Höfner leads us to the conclusion that there are common principles with reactive processes which we can exploit.

Thus, as illustrated in Figure 6, we will generalise reactive processes so that they can accommodate such continuous-time traces. We first create a theory called Generalised Reactive Processes (in Section 9.2) which substitues the concrete sequence-based trace model with an algebraic characterisation, resulting in an abstract notion of trace. As reactive processes yield reactive designs, likewise generalised reactive processes yield generalised reactive designs. This then allows us to create the theories of hybrid reactive processes and hybrid reactive designs (in Section 9.3), as specialisations using continuous timed traces, whilst retaining the standard laws of reactive designs. We finally show how the operator of the hybrid relational calculus can be embedded into this domain (illustrated by the red arrow of Figure 6).

9.2 Generalised Reactive Processes

Our approach to hybrid semantics is to generalise the theory of reactive designs to accommodate continuous time trajectories. In our new model, continuous variables are effectively embedded into the standard relational calculus through a generalised notion of trace, rather than added as a new core concept, which greatly simplifies reasoning. We first make the observation that traces need not be given a concrete model, as they are in the UTP book [30], but can be characterised algebraically. We introduce the following abstract operators: $\langle\rangle$, which denotes the empty trace, $x \frown y$, which denotes trace concatenation, $x \leq y$, a partial order on traces which denotes that x is a prefix of y, and finally $x - y$ which, when possible, removes the prefix y from x. The behaviour of these functions is characterised by the following axioms:

Definition 9.1 *Trace axioms*

$$(x \frown y) \frown z = x \frown (y \frown z) \quad \text{(tassoc)}$$
$$\langle\rangle \frown x = x \frown \langle\rangle = x \quad \text{(tident)}$$
$$(z \frown x) - (z \frown y) = x - y \quad \text{(tcancel)}$$
$$x - \langle\rangle = x \quad \text{(tminus)}$$
$$x \leq y \Leftrightarrow (\exists\, z \bullet y = x \frown z) \quad \text{(tprefix)}$$

Trace concatenation is associative and has $\langle\rangle$ as its identity, and thus traces form a monoid (tassoc, tident). Trace minus cancels concatenation (tcancel), and removing an empty prefix has no effect (tminus). Finally we require that if x is a prefix of y then this equates to there existing a suffix z such that $y = x \frown z$. A possible model for these axioms is the standard sequence model of the UTP book, which satisfies these axioms. Using these operators we can generalise the healthiness conditions of reactive processes as shown below. The definitions look unchanged from those traditionally adopted, but the sequence operators are those defined axiomatically above.

Definition 9.2 *Generalised reactive healthiness conditions*

$$\mathbf{R1}(P) \triangleq P \wedge tr \leq tr'$$
$$\mathbf{R2}(P) \triangleq P[\langle\rangle, tr' - tr/tr, tr'] \lhd tr \leq tr' \rhd P$$
$$\mathbf{R3}(P) \triangleq \pi_{\text{rea}} \lhd wait \rhd P$$
$$\mathbf{R} \triangleq \mathbf{R3} \circ \mathbf{R2} \circ \mathbf{R1}$$

The axioms of Definition 9.1 are sufficient to prove that each of these healthiness conditions is idempotent and monotone, as demonstrated in the companion Isabelle theory[9]. Moreover, they are also sufficient to prove a significant number of the standard laws of reactive designs [6, 41, 40], such as distribution through operators like internal choice and sequential composition, and that they induce a complete lattice. This then affords us a very general model of reactive programs with an abstract definition of trace, which we can now instantiate to accommodate continuous variables.

9.3 Timed Traces and Hybrid Reactive Designs

We adopt Hayes model of timed traces [23], which form the basis for timed reactive designs [25], to allow integration of continuous variables into our traces. We show that timed traces satisfy the axioms of reactive traces. This allows us to view timed reactive designs in the guise of normal UTP reactive processes. We formally define our adapted model of timed traces, $\mathbb{TT}$, below (cf. Definition 5.2):

[9]See https://github.com/isabelle-utp/utp-main/blob/shallow.2016/utp/utp_reactive.thy

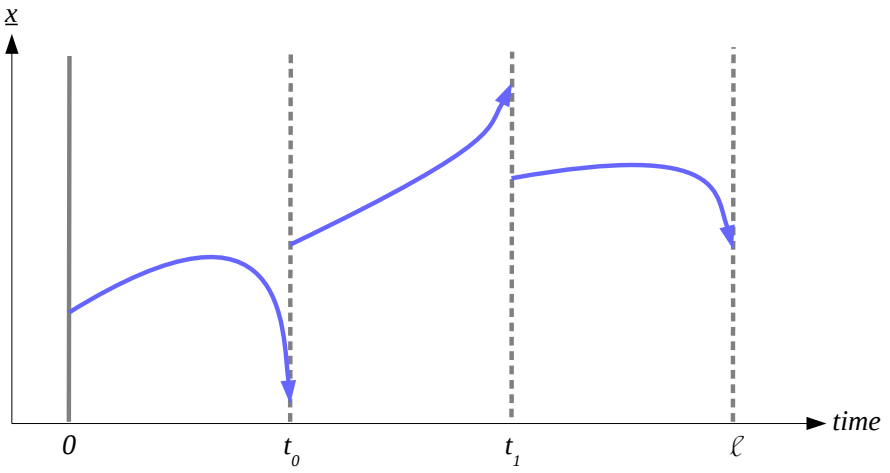


Figure 7: Example timed trace

Definition 9.3 *Timed traces*

$$\mathbb{TT} \triangleq \left\{ \begin{array}{l} f : \mathbb{R}_{\geq 0} \nrightarrow \Sigma \\ \mid \exists\, t \bullet \mathrm{dom}(f) = [0, t) \\ \qquad \wedge\ t > 0 \Rightarrow \exists\, I : \mathbb{R}_{\mathrm{oseq}} \bullet \left(\begin{array}{l} \mathrm{ran}(I) \subseteq \{0..t\} \\ \wedge\ \{0, t\} \subseteq \mathrm{ran}(I) \\ \wedge \left(\begin{array}{l} \forall\, n < \#I - 1 \bullet \\ f \ \textit{cont-on}\, [I_n, I_{n+1}) \end{array} \right) \end{array} \right) \end{array} \right\}$$

where

$$\mathbb{R}_{\mathrm{oseq}} \triangleq \{x : \mathrm{seq}\, \mathbb{R} \mid \forall\, n < \#x - 1 \bullet x_n < x_{n+1}\}$$

$$f \ \textit{cont-on}\, [m, n) \triangleq \forall\, t \in [m, n) \bullet \lim_{x \to t} f(x) = f(t)$$

An example of such a timed trace is shown in Figure 7. We require that timed traces ($\mathbb{TT}$) have the contiguous domain $[0, \ell)$, which is right-open to the end point ℓ[10]. Moreover, we require that the trace is piecewise continuous, meaning it has a finite set of discontinuities, similar to our definition of ***HCT2*** in Section 5.2. Since we need to talk about limits and continuity of the continuous state space Σ. We require that this be a topological space, such as $\mathbb{R}^n$, though it can also contain discrete topological entities, such as CSP events.

Next we can define the trace operators as below.

Definition 9.4 *Timed trace operators*

$$\begin{array}{rcl} \mathrm{end}(f) & \triangleq & \iota t \bullet \mathrm{dom}(f) = [0, t) \\ \langle\rangle & \triangleq & \emptyset \\ f \frown g & \triangleq & \lambda\, i \bullet \begin{cases} f(i) & \textit{if } i < \mathrm{end}(f) \\ g(i - \mathrm{end}(f)) & \textit{otherwise} \end{cases} \\ f \leq g & \Leftrightarrow & \mathrm{end}(f) \leq \mathrm{end}(g) \wedge (\forall\, i < \mathrm{end}(f) \bullet f(i) = g(i)) \\ g - f & \triangleq & \begin{cases} \lambda\, i \bullet f(i + \mathrm{end}(g)) & \textit{if } f \leq g \\ \bot & \textit{otherwise} \end{cases} \end{array}$$

[10] See mechanisation of $\mathbb{TT}$ in https://github.com/isabelle-utp/utp-main/blob/master/utils/ttrace.thy

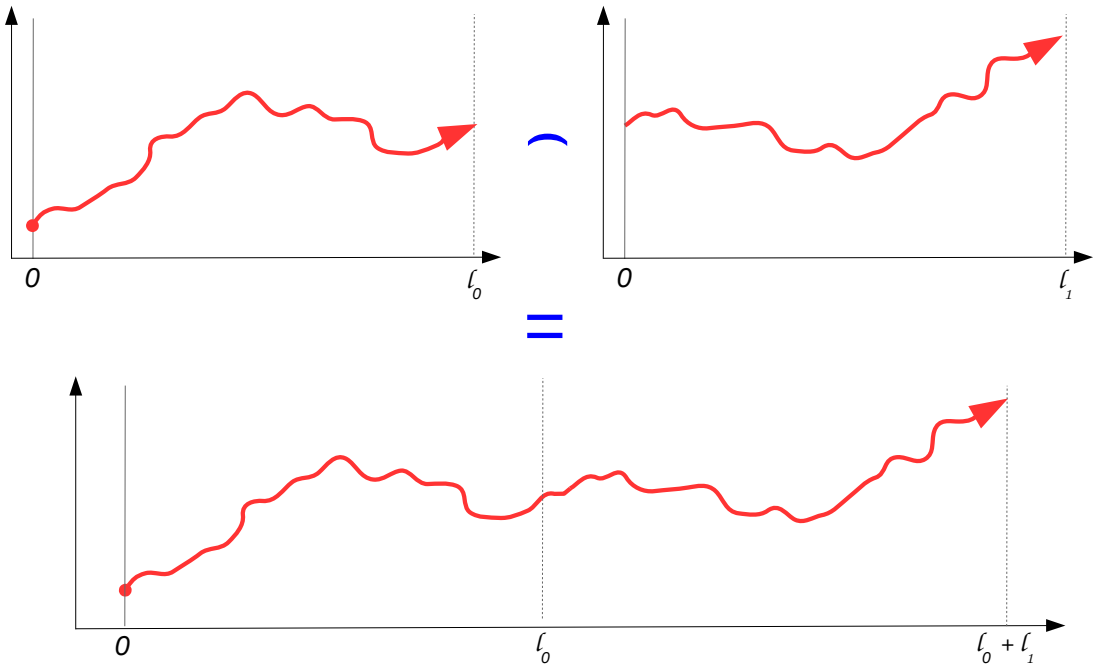


Figure 8: Timed trace concatenation

Function end(f) which gives the end time of a timed trace f through application of the definite description operator. $\langle\rangle$ denotes the empty timed trace, which is simply the empty partial function. $f \frown g$, as illustrated by Figure 8, concatenates timed traces f and g by shifting the trace domain index i when it goes beyond the end of f. $f \leq g$ is a prefix operator for timed traces: it requires that f be no longer than g, and that the traces agree on values up to the end of f. Thus $f \leq g$ corresponds to $f \subseteq g$. Finally, $g - f$ removes the initial trace of g contained in f, assuming that f is a prefix of g.

Trace concatenation and subtraction are both closed under $\mathbb{TT}$. These definitions also satisfy our trace axioms[11], and thus one can integrate continuous-time traces into generalised reactive designs with $tr, tr' : \mathbb{TT}$, which equips us with a theory of hybrid reactive designs, as illustrated in Figure 6. In this context, for instance, ***R1*** ensures that the continuous trace must grow monotonically, and thus that each behaviour must have a positive duration: $\ell \geq 0$. This instantiation means we automatically obtain the laws of reactive processes and designs. In particular, the following law shows how a continuous trace can be decomposed for a hybrid reactive process:

Theorem 9.1 (Sequential trace decomposition) *Assume that P, Q are both **R1-R2** hybrid reactive processes, then their sequential composition can be rewritten as follows.*

$$(P \,;\, Q) \;=\; \exists\, tt_1\; tt_2 \bullet ((P[\langle\rangle, tt_1/tr, tr'] \,;\, Q[\langle\rangle, tt_2/tr, tr']) \wedge tr' = tr \frown tt_1 \frown tt_2)$$

As in Section 5, we assume that the alphabet of a hybrid relation P can be subdivided into con$\alpha(P)$ and dis$\alpha(P)$, that is, the continuous and discrete variables. We then set Σ, the continuous state, to $\{f : Var \nrightarrow \mathbb{U} \mid (\forall\, x \in \mathrm{dom}(f) \bullet f(x) : x_\tau) \wedge \mathrm{dom}(f) = \mathrm{con}\alpha(P)\}$, that is a partial mapping from continuous variables to values of the correct type. This

[11] See https://github.com/isabelle-utp/utp-main/blob/8d62a2e1da72e4076e00118c8f6c317420fdf335/utils/ttrace.thy#L390

new model then allows us to recreate the core operators of the hybrid relational calculus.

Definition 9.5 *Hybrid relational calculus using timed traces*

$$
\begin{aligned}
\mathbf{tt} &\triangleq tr' - tr \\
\underline{x}(t) &\triangleq \mathbf{tt}(t)(x) \\
\ell &\triangleq \mathrm{end}(\mathbf{tt}) \\
P \mathbin{@} \tau &\triangleq \{\underline{x} \mapsto \mathbf{tt}(t)(x) \mid \underline{x} \in \mathrm{con}\alpha(P) \setminus \{\underline{t}\}\} \dagger P \\
\lceil P \rceil &\triangleq tr' > tr \wedge (\forall\, t \in [0, \ell) \bullet P \mathbin{@} t) \\
[\![P]\!] &\triangleq \lceil P \rceil \wedge \bigwedge_{\underline{v} \in \mathrm{con}\alpha(P)} \Big(v = \underline{v}(0) \wedge v' = \lim_{t \to \ell}(\underline{v}(t)) \Big) \wedge \mathit{II}_{\mathrm{dis}\alpha(P)}
\end{aligned}
$$

Here, $\mathbf{tt}$ denotes the portion of the trace that the current process contributes to. A continuous variable $\underline{x}(t)$ takes its value from the continuous state at t by selecting the appropriate variable in the domain. The definitions then broadly follow those given in Section 5. As before, all of the core relational calculus operators retain their standard definitions, which is clearly the case since we are within the theory of reactive designs.

A notable difference with this version of the calculus is its modelling of time. A hybrid relation's continuous behaviour takes place in the interval $[ti, ti')$, whereas here the interval is $[0, \ell)$, and there is no observational variable specifically for global time; as in [25] time is just a property of the trace. This means that hybrid reactive designs can only observe time from when they are started, and not at arbitrary times, as ensured by healthiness condition ***R2***.

In terms of the CSP style communication, we aim to follow a similar approach to Hybrid CSP [26] and introduce additional continuous variables into Σ, namely $\underline{tr} : \mathit{Event}$ and $\underline{ref} : \mathbb{P}\, \mathit{Event}$ that denote the events that are accepted and refused at each given time instant. This will then complete our integration of reactive processes and hybrid relations, and provide the semantic model for *CyPhyCircus*. Most likely, the model for *CyPhyCircus* will be a hybrid reactive design

$$\boldsymbol{R}(P \wedge [\![R]\!] \vdash Q \wedge [\![G]\!])$$

where P and Q are the precondition and postcondition on the discrete state, and R and G are assumptions and commitments on the continuous variables. Such a construction will enable us to apply contractual-style program construction and reasoning to concurrent Cyber-Physical Systems.

10 Conclusion

We have constructed a UTP theory of hybrid relations, which extends the alphabetised relational calculus with continuous variables whose behavioural evolution is constrained by healthiness conditions. The signature of this theory is the hybrid relational calculus enables modelling hybrid systems using imperative programming constructs, differential-algebraic equations, and continuous event preemption. We have then shown how this

theory can be used to give an initial semantics to the Modelica language, and in particular the latter's event preemption cycle. Finally, we have shown the way forward in integrating the hybrid relational calculus with concurrent and reactive behaviour through a generalised model of UTP reactive processes, which equips us with hybrid reactive designs.

There are three main strands of work to be followed in the final year of INTO-CPS in this task, some of which involves collaboration with Task T2.4. The first is to develop our theory of hybrid reactive designs further, prove its fundamental laws in Isabelle/UTP, and use it to give a semantics to our proposed lingua franca, *CyPhyCircus*. The second strand of work is to use *CyPhyCircus* to give a more comprehensive semantics for Modelica, including the description of block diagrams. We will also develop further examples which illustrate the Modelica semantics, and enable us to tackle more complex models. The third strand, in collaboration with Task 2.4, is to integrate this semantic model with the FMI semantics described in [8], to enable semantic integration of multi-models involving Modelica, VDM-RT, and other modelling languages. This will involve giving a Modelica model an interface using the FMI *get*, *set*, and *doStep* events to allow its orchestration by a master algorithm.

References

[1] 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).

[2] Bernhard Bachmann, Peter Aronsson, and Peter Fritzson. Robust initialization of differential algebraic equation. In 5th *Int. Modelica Conference*, Vienna, Austria, September 2006.

[3] A. Beg and A. Butterfield. Development of a prototype translator from Circus to CSPm. In *Proc. 9th Intl. Conf. on Open Source Systems and Technologies (ICOSST)*, pages 16–23. IEEE, December 2015.

[4] M. M. A. Beg. *Translating from "State-Rich" to "State-Poor" Process Algebras*. PhD thesis, Department of Computer Science, Trinity College Dublin, April 2016.

[5] 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.

[6] 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.

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

[8] Ana Cavalcanti and Jim Woodcock. Foundations for FMI comodelling. Technical report, INTO-CPS Deliverable, D2.2d, December 2016.

[9] J. Davies and S. Schneider. A brief history of Timed CSP. *Theoretical Computer Science*, 138(2):243–271, 1995.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] S. Foster, B. Thiele, A. Cavalcanti, and J. Woodcock. Towards a UTP semantics for Modelica. In *Proc. 6th Intl. Symp. on Unifying Theories of Programming*, volume 10134 of *LNCS*. Springer, June 2016. To appear.

[15] 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.

[16] S. Foster, F. Zeyda, and J. Woodcock. Unifying heterogeneous state-spaces with lenses. In *Proc. 13th Intl. Conf. on Theoretical Aspects of Computing (ICTAC)*, volume 9965 of *LNCS*. Springer, 2016.

[17] Simon Foster, Ana Cavalcanti, Samuel Canham, Ken Pierce, and Jim Woodcock. Final Semantics of VDM-RT. Technical report, INTO-CPS Deliverable, D2.2b, December 2016.

[18] Simon Foster, Ana Cavalcanti, Kenneth Lausdahl, Ken Pierce, and Jim Woodcock. Initial Semantics of VDM-RT. Technical report, INTO-CPS Deliverable, D2.1b, December 2015.

[19] Simon Foster, Bernhard Thiele, and Jim Woodcock. Differential Equations in the Unifying Theories of Programming. Technical report, INTO-CPS Deliverable, D2.1c, December 2015.

[20] 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.

[21] 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.

[22] I. Hayes. A predicative semantics for real-time refinement. In A. McIver and C. Morgan, editors, *Programming Methodology*. Springer, 2003.

[23] I. Hayes. Termination of real-time programs: Definitely, definitely not, or maybe. In S. Dunne and B. Stoddart, editors, *Proc. 1st Intl. Symp. on Unifying Theories of Programming*, volume 4010 of *LNCS*, pages 141–154. Springer, February 2006.

[24] I. Hayes and M. Utting. A sequential real-time refinement calculus. *Acta Informatica*, 37(6):385–448, February 2001.

[25] 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.

[26] 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.

[27] Jifeng He. HRML: a hybrid relational modelling language. In *IEEE International Conference on Software Quality, Reliability and Security (QRS 2015)*, August 2015.

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

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

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

[31] Peter Höfner and Bernhard Möller. An algebra of hybrid systems. *Journal of Logic and Algebraic Programming*, 78(2):74–97, 2009.

[32] 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.

[33] 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.

[34] Edward Lee. Constructive models of discrete and continuous physical phenomena. *IEEE Access*, 2014.

[35] 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.

[36] Modelica Association. Modelica - A Unified Object-Oriented Language for Systems Modeling - Version 3.3 Revision 1. Standard Specification, July 2014.

[37] Carroll Morgan. *Programming from Specifications*. Prentice-Hall, London, UK, 1990.

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

[39] 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.

[40] M. V. M. Oliveira. *Formal Derivation of State-Rich Reactive Programs using Circus*. PhD thesis, Department of Computer Science - University of York, UK, 2006. YCST-2006-02.

[41] 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.

[42] M. V. M Oliveira, A. C. A. Sampaio, and M. S. C. Filho. Model-checking Circus state-rich specifications. In *11th Intl. Conf. on Integrated Formal Methods*, volume 8739 of *LNCS*, pages 39–54. Springer, 2014.

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

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

[45] A. W. Roscoe, C. A. R. Hoare, and Richard Bird. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[46] Lucas Satabin, Jean-Louis Colaco, Olivier Andrieu, and Bruno Pagano. Towards a Formalized Modelica Subset. In Hilding Elmqvist Peter Fritzson, editor, 11th *Int. Modelica Conference*, Versailles, France, September 2015.

[47] Adnan Sherif, Ana Cavalcanti, He Jifeng, and Augusto Sampaio. A process algebraic framework for specification and validation of real-time systems. *Formal Aspects of Computing*, 22:153–191, 2010.

[48] 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.

[49] 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.

[50] J. C. P. Woodcock and A. L. C. Cavalcanti. A Concurrent Language for Refinement. In A. Butterfield and C. Pahl, editors, *IWFM'01: 5th Irish Workshop in Formal Methods*, BCS Electronic Workshops in Computing, Dublin, Ireland, July 2001.

[51] 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.

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

[53] F. Zeyda, S. Foster, and L. Freitas. An axiomatic value model for Isabelle/UTP. In *Proc. 6th Intl. Symp. on Unifying Theories of Programming*, volume 10134 of *LNCS*. Springer, 2016. To appear.

[54] 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.

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

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

[57] 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.

[58] 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.

[59] 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.