Hybrid Relations in Isabelle/UTP

Simon Foster[ORCiD]

University of York
simon.foster@york.ac.uk

Abstract. We describe our UTP theory of hybrid relations, which extends the relational calculus with continuous variables and differential equations. This enables the use of UTP in modelling and verification of hybrid systems, supported by our mechanisation in Isabelle/UTP. The hybrid relational calculus is built upon the same foundation as the UTP's theory of reactive processes, which is accomplished through a generalised trace algebra and a model of piecewise-continuous functions. From this foundation, we give semantics to hybrid programs, including ordinary differential equations and preemption, and show how the theory can be used to reason about sequential hybrid systems.

1 Introduction

Cyber-Physical Systems (CPSs) use computation to monitor and control real-world phenomena, employing sensors and actuators. Autonomous mobile robots, for example, implement their goals by sensing the environment, updating an internal model of the real-world, using the model to plan and make decisions, and finally actuating. A common way of modelling, simulating, and verifying CPSs is with the use of a hybrid dynamical systems modelling language, such as Simulink[1], Modelica[2], hybrid programs[3] [1], and Hybrid CSP [2,3] (HCSP). Here, a system model is decomposed into two parts: (1) a digital controller, which is described used traditional programming constructs; and (2) a continuously evolving environment, which is described using differential equations.

Languages like Simulink and Modelica are used commercially for developing CPSs, since they are largely diagrammatic in nature and can be used to produce executable code. Typically, however, such tools support only simulation and testing, which limit their effectiveness for verification. On the other hand, tools like KeYmaera X [4] and HHL Prover [5] support rigorous formal verification, for differential dynamic logic [1] ($\mathsf{d}\mathcal{L}$) and HCSP [2,3], respectively, that can prove properties of the entire state space symbolically. However, the latter tools are hard to apply for non-academics, and there is need for greater integration with the commercial tools [6]. A precondition of this is that there are

[1] Simulink: https://uk.mathworks.com/products/simulink.html
[2] Modelica Language: https://www.modelica.org/modelicalanguage
[3] The modelling notation of differential dynamic logic ($\mathsf{d}\mathcal{L}$).

unified semantic foundations for hybrid systems that acknowledge both similarities and differences between the languages, and integrated mechanised reasoning to support comprehensive automated formal verification.

The goal of this paper is to make first steps towards this foundation with a mechanised UTP theory for hybrid systems. UTP [7] is concerned with establishing formal links between languages based on heterogeneous computational paradigms, and therefore it is wholly appropriate to apply it to study of hybrid computational models. Our contributions are: (1) a UTP theory that incorporates a piecewise continuous timed trace model, building on our previous theory of generalised reactive relations [8]; (2) denotational semantics for a simple imperative language for hybrid programs, inspired by $\mathsf{d}\mathcal{L}$ and HCSP; and (3) mechanised reasoning support in our UTP theorem prover, Isabelle/UTP [9,10,11,12]. Our hybrid theory represents a substantial overhaul of our previous results [13,14] by unifying it with our generalised UTP theory of reactive processes [8].

Most theorems and definitions in the paper are accompanied by a small Isabelle icon (). In the electronic version, each icon is hyperlinked to the corresponding mechanised artefact in our Isabelle/UTP GitHub repository[4]. This, we hope, will convince the reader of the level of rigour employed in this work.

Our paper is structured as follows. §2 gives an overview of our hybrid program notation. §3 gives an overview of Isabelle/UTP, and how it is used to model programs. §4 presents our foundational UTP theory of generalised reactive processes. §5 describes a model for piecewise continuous timed traces, which is the basis for modelling continuous state spaces and variables. §6 gives a comprehensive exposition of the UTP theory of hybrid relations. §7 illustrates a small verification example in Isabelle/UTP. §8 concludes and discusses our results.

2 Hybrid Systems and Programs

In this section, we briefly introduce the key concepts of hybrid systems and programs, to the set the technical work that follows in context.

Hybrid systems exhibit both continuous flows and discrete jumps in the values of their variables [3,15]. Typically, a hybrid system evolves according to a differential equation, until some condition is satisfied, at which point a discrete jump occurs. For example, in the classic bouncing ball example, the ball is dropped and falls until it impacts the floor. During flight, the height above the ground, h, and velocity, v, change continuously. However, once the ball impacts the floor, the velocity is instantaneously inverted and it begins to travel upwards.

In this paper, we model such systems with a form of hybrid program:

Definition 2.1 (Hybrid Programs).

$$\mathcal{H} ::= ?b \mid x := v \mid \langle\, \dot{x}(t) \bullet f(t,x) \,\rangle \mid \mathcal{H} \,;\, \mathcal{H} \mid \mathcal{H} \sqcap \mathcal{H} \mid \mathcal{H}^* \mid \mathcal{H} \triangle \langle b \mid c \rangle \mid \cdots$$

where x is a name, t is a time variable, b and c are predicates, v is an expression.

[4] Isabelle/UTP repository: https://github.com/isabelle-utp/utp-main

As is common in UTP, the syntax can be further extended, which is the reason for the ellipsis. The simple language contains a mixture of constructs adapted from $\mathsf{d}\mathcal{L}$ hybrid programs and HCSP. As usual, programs can be composed sequentially ($P\ ;\ Q$) and nondeterministically ($P \sqcap Q$). As in $\mathsf{d}\mathcal{L}$, we can also define tests, $?b$, which execute when assumption b is satisfied, and assignment $x := v$. Hybrid programs can be iterative, which is expressed by the Kleene star P^*.

We characterise ordinary differential equations (ODEs), $\langle\, \dot{x}(t) \bullet f(t, x)\, \rangle$, which express that the derivative of the variable vector x is given by f. Its behaviour is to evolve the variables, without terminating, according to a solution $x(t)$, such that $\dot{x}(t) = f(t, x(t))$. For example, we can model a real-time clock by creating a distinguished continuous variable called *time*, such that $\dot{time}(t) = 1$.

Finally, $P \triangle \langle b \mid c \rangle$ allows preemption of a continuous evolution. Evolution of P may continue whilst b, a condition on the continuous variables, is invariant, and can terminate once c becomes true. The reason for having both b and c is to allow nondeterminism around when an evolution is preempted. Such nondeterminism exists in languages like Modelica, where discrete jumps are implemented using "zero-crossing detection", such that a function goes from positive to negative or vice-versa. This is subject to numerical imprecision, and thus the point at which the event occurs is effectively nondeterministic.

To illustrate hybrid programs, we formalise the bouncing ball example below:

Example 2.2 (Bouncing Ball as a Hybrid Program).

$$\text{BBall} \triangleq h := 2.0\ ;\ v := 0\ ; \left(\begin{array}{l} \left\langle\, (\dot{h}(t), \dot{v}(t)) \bullet (v, -9.81)\, \right\rangle \\ \quad \triangle\ \langle h \geq 0 \mid v \leq 0 \wedge h < \epsilon \rangle\ ; \\ v := -0.8 \cdot v \end{array} \right)^*$$

Initially, the height of the ball is 2 meters, and the velocity is 0. Then, the main body of the system begins, by first evolving h and v according to a system of ODEs. The ODEs state that the derivative of h is v, and the derivative of v is -9.81, the standard gravity constant. Evolution continues whilst $h \geq 0$: the ball is above the ground, which gives a bound on the evolution. Once h falls below a constant $\epsilon > 0$, a small number which characterises numerical imprecision, and assuming $v \leq 0$ (flight is downwards), then the evolution can terminate. At this point, v is discontinuously inverted and a damping factor of .8 is applied. Through the Kleene star, the system is permitted to iterate zero or more times.

In the remainder of this paper, we show how such hybrid programs can be mechanically supported with our UTP theory of hybrid relations.

3 Isabelle/UTP

Isabelle/UTP [10,11] is a mechanisation of UTP in Isabelle/HOL, along with the main results from the UTP book [7] and related publications [16,17]. It provides an implementation of the alphabetised relational calculus, a model of imperative programs, a large library of algebraic laws, and several automated proof

tactics. It mechanically supports the following activities: (1) development of UTP theories for languages of various computational paradigms; (2) construction of denotational semantics for said languages; and (3) creation of proof strategies to support automated verification tools using UTP theories. Aside from UTP, our mechanisation also draws heavily on the work of Back and von Wright [18].

For the imperative program model, Isabelle/UTP supports several calculi, including Hoare logic, weakest (liberal) precondition, and structural operational semantics. These axiomatic semantics are defined denotationally, and the associated laws proved as theorems. Linking theorems can also show correspondences between different semantic presentations. For example, it is well known that

$$\{\, b \,\}\, P \,\{\, c \,\} = (b \Rightarrow P \mathbin{\textbf{wlp}} c)$$

is a theorem of Hoare calculus and weakest liberal preconditions [7,19]. Such a result can be harnessed to recast a verification theorem into a different form, which is potentially easier to prove. For each verification calculus, Isabelle/UTP also provides proof tactics, including deductive reasoning for Hoare logic, and equational reduction for ***wlp***. The output of these tactics is a set of verification conditions (VCs), to which Isabelle's automated proof strategies can be applied.

A main objective in implementing Isabelle/UTP has been to harness the power of Isabelle's automated reasoning. Consequently, Isabelle/UTP follows in the tradition of shallow embeddings [20,21,22] in reusing as much as possible of the Isabelle technical infrastructure, such as its type system, parser, term language, and meta-logic, in defining the relational calculus. However, this objective must be reconciled with the need to provide a sufficiently expressive relation model to allow expression of UTP theories and the associated laws. The crucial artifact to get right here is the mechanisation of UTP variables and alphabets.

In Isabelle/UTP, state spaces are modelled as Isabelle types, and programs are parametric in their state space. Assuming a suitable state space Σ, a relational program is effectively modelled as a subset of $\mathbb{P}(\Sigma \times \Sigma)$, and an expression of type τ can be modelled as functions $\Sigma \to \tau$. This is consistent with most other works on verification using Isabelle/HOL [20,23], and allows us to obtain the UTP relational operators easily, such as disjunction ($P \vee Q$), relational composition ($P \,;\, Q$), tests ($?b$), and refinement ($P \sqsubseteq Q$). However, this does not in itself provide us with a variable model. Rather than modelling these syntactically using names, we treat them as algebraic objects called lenses [24,18,11]:

Definition 3.1. *A lens is a quadruple* $\langle \mathcal{V} \mid \mathcal{S} \mid \mathsf{get} : \mathcal{S} \to \mathcal{V} \mid \mathsf{put} : \mathcal{S} \to \mathcal{V} \to \mathcal{S} \rangle$, *where* $\mathcal{V}$ *and* $\mathcal{S}$ *are non-empty sets called the view and source, respectively, and* get *and* put *are total functions, such that the following equations hold:*

$$\mathsf{get}\,(\mathsf{put}\,s\,v) = v \qquad \mathsf{put}\,(\mathsf{put}\,s\,v')\,v = \mathsf{put}\,s\,v \qquad \mathsf{put}\,s\,(\mathsf{get}\,s) = s$$

We write $\mathcal{V} \Longrightarrow \mathcal{S}$ *to denote the type of lenses with source type* $\mathcal{S}$ *and view type* $\mathcal{V}$, *and subscript* get *and* put *with the name of a particular lens.*

Each variable $x : \tau$ in UTP is modelled using a lens $\tau \Longrightarrow \Sigma$, for some suitable state space type, and each $\mathsf{get}_x/\mathsf{put}_x$ pair is used to query and update its value.

The main advantage of this algebraic encoding is that we obtain an abstract representation that unifies several state space representations. We also note that a very similar concept to lenses exists in Back's refinement calculus [18, Chapter 5], which substantially predates the work on lenses [24].

With lenses, expressions can modelled as functions on the Σ; for example:

$$[\![x > (y + z)/2]\!] = (\lambda\, s : \Sigma \bullet \mathit{get}_x\ s > (\mathit{get}_y\ s + \mathit{get}_z\ s)/2)$$

In this way, each operator at the expression level corresponds to a point-wise lifting of the corresponding operator through the state s at the function level.

With lenses, we can also generically characterise several variable properties:

1. independence, $x \bowtie y$ — x and y refer to disjoint views of Σ;
2. inclusion partial order, $x \preceq y$ — the view of y contains the view of x;
3. equivalence, $x \approx y$ — the lenses x and y refer to identical views.

All of these properties reduce to properties of the corresponding *get* and *put* functions; for example independence is essentially commutativity of put_x and put_y. They allow us to effectively characterise meta-logic properties of variables, which are normally characteristic of a deep embedding.

Variable updates are described using substitutions $\sigma : \Sigma \to \Sigma$, which are total functions on the state space, and allow us to describe assignments, variables contexts, and substitutions. The most basic substitution is the identity function, ***id***, which effectively maps every variable to its present value. A substitution can be updated using the operator $\sigma(x \mapsto e)$, which associates x with an expression e over Σ. Then, we use the notation $[x_1 \mapsto e_1, \cdots, x_n \mapsto e_n]$ as a shorthand for $\mathit{id}(x_1 \mapsto e_1 \cdots x_n \mapsto e_n)$, which is a simultaneous substitution for n variables. Substitution update obeys several algebraic laws:

Theorem 3.2. *If x and y are lenses, then the following identities hold:*

$$\sigma(x \mapsto x) = \sigma \tag{3.2.1}$$

$$\sigma(x \mapsto e, y \mapsto f) = \sigma(y \mapsto f, x \mapsto e) \quad \text{if } x \bowtie y \tag{3.2.2}$$

$$\sigma(x \mapsto e, y \mapsto f) = \sigma(y \mapsto f) \quad \text{if } x \preceq y \tag{3.2.3}$$

(3.2.1) shows that a trivial update is ineffectual. (3.2.2) shows that two maplets may be commuted if the lenses are independent. (3.2.3) shows that a maplet for x is overridden by one assigning y when $x \preceq y$, and thus also when $x \approx y$.

Substitutions can be applied to expressions using $\sigma \dagger e$, which replaces all the variables in e with those assigned in σ. This is similar to syntactic substitution, with $e[v/x] = [x \mapsto v] \dagger e$, and obeys similar laws, but it is a semantic operator that composes σ with e (both are functions).

We also use substitutions to construct assignments, using Back's generalised operator [18]: $\langle\sigma\rangle$. This operator recasts the function σ as a relation. A singleton assignment, $x := v$, can be denoted using $\langle x \mapsto v\rangle$, and a simultaneous assignment by $\langle x_1 \mapsto v_1, x_2 \mapsto v_2, \cdots\rangle$. We can prove several familiar assignment laws:

Theorem 3.3 (Assignment Laws).

$$\langle \sigma \rangle \,;\, \langle \rho \rangle = \langle \rho \circ \sigma \rangle \tag{3.3.1}$$

$$x := x = \langle \mathit{id} \rangle \tag{3.3.2}$$

$$x := e \,;\, y := f = y := f \,;\, x := e \qquad x \bowtie y, x \sharp f, y \sharp e \tag{3.3.3}$$

$$x := e \,;\, x := f = x := f[e/x] \tag{3.3.4}$$

The first law is a homomorphism law for assignments. The other laws are corollaries of it and the laws of Theorem 3.2. The third law, showing that assignments commute, requires an extra side condition that f does not mention x, and e does not mention y. These are both formulated using a semantic operator called unrestriction, $x \sharp f$, which means that f does not depend on the state space region characterised by x for its valuation, and is denoted using the lens operators [10].

Thus we have demonstrated the ubiquity of lenses in capturing the UTP relational calculus. In the next section we describe of theory of generalised reactive processes that is the foundation of the hybrid relational calculus. Later, we show how lenses are used to characterise continuous variables.

4 Trace Algebra and Generalised Reactive Relations

In this section, we describe our theory of generalised reactive relations. This UTP theory provides the foundation for our theory of hybrid relations using an abstract trace model. This, in particular, can be instantiated with piecewise continuous functions, which are often used to semantically capture the behaviour of hybrid systems [3,15].

The UTP theory of reactive processes [7,16] provides a generic foundation for trace-based reactive languages. Originally the trace model was fixed to discrete sequences, to support the semantic models of CSP and ACP [7]. In previous work [8], we generalised this theory to characterise traces abstractly with a *trace algebra*. We characterise traces with a set $\mathcal{T}$ and two operators: concatenation $\frown : \mathcal{T} \to \mathcal{T} \to \mathcal{T}$, and the empty trace $\varepsilon : \mathcal{T}$, which obey the following axioms [8].

Definition 4.1. *A trace algebra $(\mathcal{T}, \frown, \varepsilon)$ satisfies the following axioms:*

$$x \frown (y \frown z) = (x \frown y) \frown z \quad \text{(TA1)}$$

$$\varepsilon \frown x = x \frown \varepsilon = x \quad \text{(TA2)}$$

$$x \frown y = x \frown z \;\Rightarrow\; y = z \quad \text{(TA3)}$$

$$x \frown z = y \frown z \;\Rightarrow\; x = y \quad \text{(TA4)}$$

$$x \frown y = \varepsilon \;\Rightarrow\; x = \varepsilon \quad \text{(TA5)}$$

TA5 ensures that every trace is positive ($x \geq 0$); its lefthand dual is a theorem of these axioms. An example model is formed by finite sequences, $\langle a, b, \cdots, z \rangle$, that is $(\mathrm{seq}\, A, \frown, \langle\rangle)$ forms a trace algebra, where $\frown$ is concatenation. Using the trace algebra operators, we can define trace prefix ($x \leq y$), which partially orders traces, and trace difference ($x - y$), which removes a prefix y from x [8].

From these algebraic foundations, we reconstruct the complete UTP theory of reactive processes [7,16], including its healthiness conditions and associated laws,

in particular those for sequential and parallel composition [8]. For our version of the theory, the alphabet includes the following observational variables:

1. $ok, ok' : \mathbb{B}$ – to indicate whether there is divergence;
2. $wait, wait' : \mathbb{B}$ – to indicate whether a process is intermediate;
3. $tr, tr' : \mathcal{T}$ – to represent the trace, using a suitable trace algebra;
4. $\mathit{st}, \mathit{st}' : \Sigma$ – to represent the state, for some non-empty state space Σ.

We then define the following reactive healthiness conditions [7,16]:

Definition 4.2 (Reactive Relations Healthiness Conditions).

$$\begin{aligned}
\mathbf{R1}(P) &\triangleq P \wedge tr \leq tr' \\
\mathbf{R2}(P) &\triangleq P[\langle\rangle, tr' - tr/tr, tr'] \lhd tr \leq tr' \rhd P \\
\mathbf{RR}(P) &\triangleq (\exists (ok, ok', wait, wait') \bullet \mathbf{R1}(\mathbf{R2}(P))) \\
\mathbf{RC}(P) &\triangleq \mathbf{R1}(\mathbf{RR}(P) \; ; \; tr' \leq tr) \\
\mathit{tt} &\triangleq (tr' - tr)
\end{aligned}$$

The main healthiness conditions are **RR**, which describes reactive relations, and **RC**, which describes a subset of **RR** called reactive conditions. For our purposes, a reactive relation is, intuitively, a relation that refers to the initial and final values of state variables (x and x', where $x \preceq \mathit{st}$), and a special variable tt, that denotes a trace contribution. Technically, tt is an expression that denotes the difference $tr' - tr$, whose well-formedness is ensured by the commuting reactive healthiness conditions **R1** and **R2**. This is reflected by the following theorem:

Theorem 4.3. *If P is **RR** healthy then* $P = (\exists\, t \bullet P[\varepsilon, t/tr, tr'] \wedge tr' = tr \frown t)$

Any observation of a reactive relation P characterises a trace extension t which can be observed using tt. Reactive relations are closed under most relational operators; the exceptions are the universal relation (**true**), complement ($\neg$), and implication ($\Rightarrow$). These all require imposition of **R1**, and so we recast them as $\mathbf{true}_\mathsf{r}$, $\neg_\mathsf{r}$, and $\Rightarrow_\mathsf{r}$, respectively. Reactive relations form several algebras, including (1) a Boolean algebra [25], (2) a complete lattice [25], and (3) a Kleene algebra [26], which allows us to reason about iterative reactive programs (P^*).

The generalised assignment operator, $\langle\sigma\rangle$, is not in general healthy as it permits assignment of any variable, including ok, $wait$, and tr, which can violate **RR**. Consequently, we recast the operator $\langle\sigma\rangle_\mathsf{r}$, where $\sigma : \Sigma \to \Sigma$ operates on the program state in st only. It obeys analogous laws to those in Theorem 3.3.

The second main healthiness condition in Definition 4.2, **RC**, characterises reactive conditions. A reactive condition is a reactive relation which (1) does not refer the final value of the state variables (st'), and (2) characterises a set of traces that is prefix closed. Reactive conditions are analogous to relational conditions, which refer to the initial state only, but can refer to both tr and tr', provided that tt is prefix closed. For example, if we apply **RC** to the non prefix-closed relation $tr' = tr \frown \langle a \rangle$, then we obtain $\mathbf{R1}(tr' \leq tr \frown \langle a \rangle) = \mathbf{R1}(\mathit{tt} \in \{\langle\rangle, \langle a \rangle\})$, which is

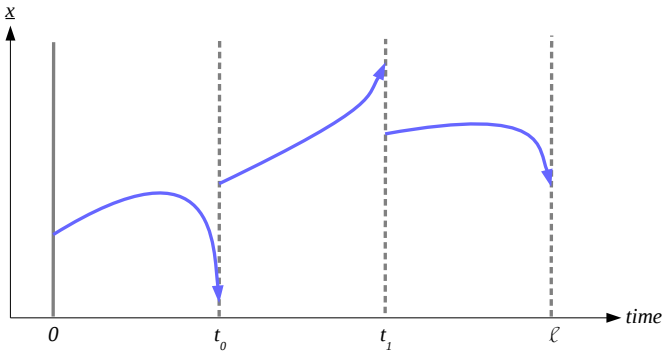

Fig. 1. Piecewise continuous timed traces

prefix closed. The intuition is that when a reactive condition is satisfied, it should also be satisfied by any prefix of the trace. Reactive conditions are particularly useful to characterise assumptions in our theory of reactive contracts [25], which extends reactive relations with assume/guarantee reasoning (see §6.5).

We have outlined our theory of reactive relations. In the next section we construct a trace algebra model that allows us to specialise to hybrid relations.

5 Continuous State and Timed Traces

In this section we describe how continuous state is modelled using a timed trace model, which characterises piecewise continuous trajectories [15]. Our model refines our previous work [8] by requiring that each continuous segment also converges, ensuring that its final value can be obtained. This requirement is always satisfied by, for example, linear ODEs. The state space (Σ) of a hybrid system consists of discrete variables, which exhibit only jumps at certain instants, and continuous variables, which change constantly with respect to time. Consequently, we subdivide the state space Σ into a discrete state space (Σ_d) and a continuous state space (Σ_c), both of which are non-empty, and then $\Sigma \triangleq \Sigma_d \times \Sigma_c$.

We require that Σ_c, minimally, forms a topological (Hausdorff) space, so that we can describe limits of a function over Σ_c. A special case is when $\Sigma_c = \mathbb{R}^n$, for some $n : \mathbb{N}$, that is a Euclidean state space. We impose no additional constraints on Σ_d which characterises variables that only exhibit discontinuous changes.

We now define our model of timed traces, which refines our previous model [8] by adding a convergence requirement:

Definition 5.1 (Timed Traces).

$$\mathbb{TT}_{\Sigma_c} \triangleq \left\{ f : \mathbb{R}_{\geq 0} \nrightarrow \Sigma_c \;\middle|\; \begin{array}{l} \exists\, t : \mathbb{R}_{\geq 0} \bullet \mathrm{dom}(f) = [0, t) \wedge \\ \left(t > 0 \Rightarrow \exists\, I : \mathbb{R}_{\mathsf{osq}} \bullet \left(\begin{array}{l} \mathrm{ran}(I) \subseteq [0, t] \wedge \\ \{0, t\} \subseteq \mathrm{ran}(I) \wedge \\ \forall\, n \in [0, \#I - 2] \\ \quad \bullet \left(\begin{array}{l} f \ \mathit{cont\text{-}on}\ [I_n, I_{n+1}) \\ \wedge\ f \ \mathit{has\text{-}limit}\ I_{n+1} \end{array} \right) \end{array} \right) \right) \end{array} \right\}$$

where
$$\mathbb{R}_{\mathsf{osq}} \triangleq \{x : \mathrm{seq}\,\mathbb{R} \mid \forall\, n < \#x - 1 \bullet x_n < x_{n+1}\}$$
$$f \textit{ cont-on } A \triangleq \forall\, t \in A \bullet \lim_{x \downarrow t} f(x) = f(t)$$
$$f \textit{ has-limit } k \triangleq (\exists\, l : \mathbb{R} \bullet \lim_{x \uparrow k} f(x) = l)$$

A timed trace is a partial function from positive real numbers ($\mathbb{R}_{\geq 0}$) to the continuous state space, Σ_c, that satisfies certain constraints. Firstly, we require that the domain is a right open interval from 0 to some positive real t. Secondly, if t is non-zero, we require that the function is composed of a sequence of continuous segments, each of which converges to a limit. The intuition is sketched in Figure 1 for a state space with a single variable x. There are three continuous segments, with domains $[0, t_0)$, $[t_0, t_1)$, and $[t_1, \ell)$, where ℓ is the end of timed trace. At each segment end point, such as t_0 and t_1, the trajectory may make a discontinuous jump, following the standard piecewise continuous trajectory model [15].

We specify this in Definition 5.1 by requiring that there is a strictly ordered sequence of real numbers I, that give the start and end point of each segment. $\mathbb{R}_{\mathsf{osq}}$ is the subset of finite real sequences such that for every index n in the sequence less than its length minus one ($\#x - 1$), $x_n < x_{n+1}$. I must contain at least 0 and t, such that at least one segment is present, and only values between these two extremes. The timed trace f is required to be continuous on each interval $[I_n, I_{n+1})$, and convergent to a limit at I_{n+1}. The operator f *cont-on* A specifies that f is continuous on the range given by A, by requiring that, each point $t \in A$, the limit of $f(x)$ as x approaches t from above equals $f(t)$. We use the upper limit as the lower limit may be different, for example at the discontinuous jumps t_0 and t_1 in Figure 1. The operator f *has-limit* k requires that there is a limit point l such that f converges toward l as it approaches k from below. Due, to discontinuity, the value at k may be different.

From this model, we can now introduce the core operators on timed traces, which we previously defined in [8], inspired by [27]. We reproduce them here for completeness and because our timed trace model has further constraints.

Definition 5.2 (Timed-trace Operators).

$$f \gg n \triangleq \lambda\, x \bullet f(x - n) \qquad \varepsilon \triangleq \emptyset$$
$$\mathrm{end}(f) \triangleq \min(\mathbb{R}_{\geq 0} \setminus \mathrm{dom}(f)) \qquad f \frown g \triangleq f \cup (g \gg \mathrm{end}(f))$$

Function $f \gg n$ shifts the indices of a partial function $f : \mathbb{R}_{\geq 0} \nrightarrow A$ to the right by $n : \mathbb{R}_{\geq 0}$. The operator $\mathrm{end}(f)$ gives the end time of a trace $f : \mathbb{TT}_{\Sigma_c}$ by taking the infimum of the real numbers excluding the domain of f. The empty trace ε is the empty function. Finally, $f \frown g$ shifts the domain of g to start at the end of f, and takes the union. From these definitions, we prove the following theorem.

Theorem 5.3. *For any Σ_c, $(\mathbb{TT}_{\Sigma_c}, \frown, \varepsilon)$ forms a trace algebra.*

This model is the foundation for hybrid relations, which we now describe.

6 Hybrid Relations

In this section we describe our hybrid relational calculus, which specialises our theory of reactive relations with our timed trace model. We use this to give a denotational semantics to the hybrid programming language described in §2, including continuous variables, continuous specifications, and systems of ordinary differential equations (ODEs). A preliminary presentation of the materials in this section can be found in a previous technical report [28].

6.1 Continuous Variables

A hybrid relation is a specialised reactive relation where the underlying trace model is $(\mathbb{TT}_{\Sigma_c}, ^\frown, \varepsilon)$, with $tr, tr' : \mathbb{TT}_{\Sigma_c}$ and $\mathit{st}, \mathit{st}' : \Sigma_d \times \Sigma_c$. Intuitively, a hybrid relation describes a set of trajectories that characterise the possible behaviours of the continuous variables. The trace contribution (tt) refers to a particular evolution of the continuous state space, Σ_c. We introduce the syntax $\ell \triangleq \mathrm{end}(\mathit{tt})$, which refers to the length of the present evolution in a hybrid relation [29].

As outlined in §4, our theory provides us with the operators of an imperative programming language. Consequently, we do not redefine them here, but reuse their existing definitions and laws. This is a key contribution of our approach – we need now only consider the specialised continuous evolution operators.

The hybrid state in st consists of the discrete and continuous state. We introduce independent lenses $\mathbf{d} : \Sigma_d \Longrightarrow \Sigma$ and $\mathbf{c} : \Sigma_c \Longrightarrow \Sigma$ for these subregions, respectively, which are the first and second projections. We introduce the syntax $s{:}x$ to project the part of state space s described by lens x, and then refer to discrete-state variables using $\mathbf{d}{:}x$ and continuous-state variables using $\mathbf{c}{:}y$.

Continuous variables are modelled as projections from the state space, Σ_c, over time. We likewise use lenses to model these projections, so that each variable x identifies a region of Σ_c, such as $x : \mathbb{R} \Longrightarrow \Sigma_c$. The source type of each lens, that is the type of data it refers to, is not limited to $\mathbb{R}$ but can be any topological space. A trajectory variable expression, $\widetilde{x}(t)$, can then be defined as follows:

Definition 6.1 (Trajectory Variables). $\widetilde{x}(t) \triangleq \mathit{tt}(t){:}x$ 

A trajectory variable, $\widetilde{x}$, is a function that obtains the continuous state space from the timed trace, recorded in tt, at time $t : \mathbb{R}_{\geq 0}$, and then projects the corresponding region using lens x. Here, t denotes time relative to the start of an evolution, and not absolute time, a property imposed by healthiness condition **R2**. Absolute time should instead be modelled as a distinguished continuous variable (cf. §2). It is also important to distinguish these trajectory variables, which are functions on the timed trace, from **state variables**, that is the valuation of the continuous variables at the start or end of a computation, characterised by st. These quantities are related, but are not identical. The value of $\widetilde{x}(t)$ is not *a priori* the same as the corresponding variable $\mathbf{c}{:}x'$, for example, because the former is part of tt, but the the latter is part of st, and $\mathit{st} \bowtie \mathit{tt}$. Later in this section, we will introduce coupling invariants to link these quantities.

Next, we describe instant relations, which lift relational predicates on the continuous state to hybrid relations:

Definition 6.2 (Instant Relations). $P \mathbin{@} t \triangleq [\boldsymbol{c}' \mapsto \mathit{tt}(t)] \dagger P$ 

An instant predicate expression, $P \mathbin{@} t$, lifts primed continuous state variables, referred to in P, to continuous trajectory variables. P is a relation over the discrete and continuous state variables, that is a subset of $\mathbb{P}(\Sigma \times \Sigma)$.

To exemplify, the expression $(x' > 7.5) \mathbin{@} t$ is equivalent to $\widetilde{x}(t) > 7.5$, that is, the predicate that asserts that continuous state variable x is greater than 7.5 at time t. Instant relations can also refer to the initial values of continuous variables: primed variables (x') are used to denote the valuation of x at t, whereas its unprimed variant (x) simply refers to the initial value. Thus, the relation $(x > x') \mathbin{@} t$ is equivalent to $\widetilde{x}(t) > x$ — the value of x in the trajectory at time t is greater than it was initially. Effectively P is a relation between initial values of continuous variables, alternatively written as x_0, and the valuation of the variables at t. The definition of $P \mathbin{@} t$ simply substitutes the valuation of the continuous state variable $\boldsymbol{c}'$ for $\mathit{tt}(t)$: the trajectory state at t.

We next define an interval operator, inspired by Duration Calculus [30,29]:

Definition 6.3 (Interval). $\mathit{dur}\,[P(t)] \triangleq \textbf{\textit{R1}}\,(\forall\, \tau \in [0, \ell) \bullet P(\tau) \mathbin{@} \tau)$ 

An interval specification $\mathit{dur}\,[P(t)]$ states that such a relation P holds over the entire evolution of the trajectory. Here, P is also parametrised by the current time t, which allows continuous variables to also depend on time. Technically, $t : \mathbb{R}_{\geq 0}$ is distinguished variable that is often used in continuous time predicates. We can obtain a Duration Calculus style specification operator with $\lceil p \rceil \triangleq \mathit{dur}\,[p']$, where p is a predicate on undashed variables only that does not mention t. This simplified operator states that the invariant p holds over the evolution.

The definition of $\mathit{dur}\,[P(t)]$ states that P holds at every instant τ between 0 and ℓ, and additionally enforces ***R1*** to ensure only healthy timed traces are permitted. The construction is automatically ***R2*** since it only refers to tt and not tr or tr' explicitly. Thus, since neither ok nor $wait$ are mentioned, $\mathit{dur}\,[P(\tau)]$ is an ***RR*** healthy reactive relation. Moreover, it is also ***RC*** healthy, since the set of timed traces specified is prefix-closed. The derived duration operator has the following laws, adapted from Duration Calculus, as theorems:

Theorem 6.4 (Interval Laws). 

$$\lceil p \wedge q \rceil = (\lceil p \rceil \wedge \lceil q \rceil) \qquad \lceil \textbf{\textit{false}} \rceil = (tr' = tr)$$
$$\lceil p \vee q \rceil \sqsubseteq (\lceil p \rceil \vee \lceil q \rceil) \qquad \lceil \textbf{\textit{true}} \rceil = \textbf{\textit{true}}_r$$
$$\lceil p \rceil \,;\, \lceil p \rceil = \lceil p \rceil$$

6.2 Continuous Function Evolution

The operators defined so far only permit specification of trajectory variables. In order to link these to continuous state variables so that, for instance, we can assign continuous variables, we define two coupling invariant operators.

Definition 6.5 (Continuous Coupling Invariants). 

$$\mathit{II}_x \triangleq \mathit{R1}\,(\mathbf{c}{:}x = \widetilde{x}(0)) \qquad \mathit{rI}_x \triangleq \mathit{R1}\left(\mathbf{c}{:}x' = \lim_{t \uparrow \ell} \widetilde{x}(t)\right)$$

The first coupling invariant, II_x, links together the initial value of continuous state variable x with the corresponding trajectory variable at time 0. The second, rI_x, links the final value of continuous state variable x (that is, x') with the limit of the corresponding trajectory variable as it approaches the duration of the evolution (ℓ) from the left. By Definition 5.1, we know that the latter limit must exist, since our timed traces are piecewise convergent.

The asymmetry of the two invariants is important. Whilst the trajectory explicitly defines a value at time 0, as invoked by II, it does not define one at ℓ since the domain is the right-open interval $[0, \ell)$. The final value exists, however, because the timed trace converges to a limit. However, when sequentially composing hybrid relations, and thus composing the two trajectories, a discrete jump is permitted so that the value at t and the left limit at t need not be the same. Both II_x and rI_x are healthy reactive relations.

We can now define operators for continuous function evolution:

Definition 6.6 (Function Evolution). 

$$\widetilde{x}(t) \leftarrow f(t) \triangleq (\mathit{dur}\,[x' = f(t)] \wedge \ell > 0)$$

$$\widetilde{x}(t) \underset{\leq d}{\leftarrow} f(t) \triangleq (\widetilde{x}(t) \leftarrow f(t) \wedge \ell \leq d)$$

$$\widetilde{x}(t) \underset{[s,d]}{\leftarrow} f(t) \triangleq \left(\widetilde{x}(t) \leftarrow f(t) \wedge \ell \in [s, d] \wedge \mathbf{d}' = \mathbf{d} \wedge \mathit{rI}_{\mathbf{v}}\right)$$

where $x : \mathbb{R}^n \Longrightarrow \Sigma_c$, $f : \mathbb{R}_{\geq 0} \to \mathbb{R}^n$, *and* $d : \mathbb{R}_{\geq 0}$.

Here, $\mathbf{v}$ is a special variable that denotes the entirety of the state space. The first operator, $\widetilde{x}(t) \leftarrow f(t)$, states that the trajectory variable x evolves according to continuous function f in n variables. We require that such an evolution have non-zero duration, as otherwise the function's behaviour cannot be observed. Lens x can consist of several continuous variables, and thus a function evolution can be used to encode a system of simultaneous algebraic equations, as present, for example, in Modelica. It is also worth noting that other continuous variables not mentioned in such a statement are unconstrained and thus behave nondeterministically. This is an important feature of the model as it allows the use of nondeterminism to model concurrency of parallel hybrid processes.

We exemplify the semantics of function evolution with the calculation below:

Example 6.7 (Evolution Calculation).

$$\begin{aligned}
& \widetilde{v}(t) \leftarrow v - 9.81 \cdot t \\
= \; & (\mathit{dur}\,[v' = v - 9.81 \cdot t] \wedge \ell > 0) \\
= \; & (\mathit{R1}\,(\forall\, t \in [0, \ell) \bullet \widetilde{v}(t) = v - 9.81 \cdot t) \wedge \ell > 0) \\
= \; & (tr \leq tr' \wedge (\forall\, t \in [0, \ell) \bullet \mathit{tt}(t){:}v = v - 9.81 \cdot t) \wedge \ell > 0)
\end{aligned}$$

The evolution is first mapped to a duration with the equation $v' = v - 9.81 \cdot t$, meaning that v is assigned the initial value of v minus $9.81 \cdot t$, at each instant. In the second and third steps, the interval operator is expanded to a predicate that assigns a value to $\widetilde{v}$ over the whole duration. Ultimately, this continuous variable is simply a reference to tt, which shows how the semantics builds on generalised reactive relations. Function evolution also admits the following valuable theorem:

Theorem 6.8. $\boldsymbol{c}{:}y := v \,;\, \widetilde{x}(t) \leftarrow f(t) \;=\; \widetilde{x}(t) \leftarrow (f(t))[v/y]$

This law shows the effect of pushing a leading assignment to y into a function evolution. Any instance of y in the continuous function expression $f(t)$ is replaced by v. This allows evaluation of any expressions that depend upon the initial state.

The second operator in Definition 6.6, $\widetilde{x}(t) \leftarrow_{\leq d} f(t)$, is the same as the above, but adds the requirement that the duration be at most d. The third and final operator, $\widetilde{x}(t) \leftarrow_{[s,d]} f(t)$, states that the evolution terminates non-deterministically in the interval $s \leq t \leq d$. This operator explicitly terminates the function's evolution and thus additionally states that all discrete variables should remain the same as they were at the start, and applies coupling invariant $\boldsymbol{rl_V}$ to set the final state of all continuous variables. All the function evolution operators in Definition 6.6 form healthy reactive relations.

6.3 Preemption of Evolution

We next define the preemption operator:

Definition 6.9. $P \triangle \langle b \mid c \rangle \;\triangleq\; (P \wedge \mathit{dur}[b] \wedge \ell > 0 \wedge \boldsymbol{rl_V} \wedge c' \wedge \boldsymbol{d'} = \boldsymbol{d})$

The preemption operator $(P \triangle \langle b \mid c \rangle)$ states that P evolves for some non-zero duration, while condition b holds. At some undetermined point, c should become true finally, and at this point the operator can terminate. This yields final values for all variables, obtained using the right limit, and requires that all discrete variables remain unchanged over the evolution.

Intuitively, the first condition, b, is similar to the invariants present in hybrid automata [31]. Evolution of P can continue while b remains true, which is ensured by conjunction with the interval specification $\mathit{dur}[b]$. On the other hand, evolution of P can terminate whenever the final continuous state satisfies c. Since b and c can overlap there is potential nondeterminism as to when P terminates, which is necessary when handling numerical imprecision. In the special case that $b = (\neg c)$, there is at most one instant at which P terminates, leading to a precise and purely deterministic preemption. If c never becomes true then this operator evaluates to false, that is, a non-terminating reactive relation.

We give an important theorem regarding termination of a function evolution:

Theorem 6.10. *We assume that f is a continuous function on the domain $[0, l]$, where $l > k$ and $k > 0$, and the following conditions hold:*

1. *b is satisfied for all instants $t \in [0, l)$: $\forall\, t \in [0, l) \bullet b[f(t)/x']$;*
2. *b becomes false at l: $\neg b[f(l)/x']$;*

3. *c is not satisfied for all instants* $t \in [0, k)$*:* $\forall\, t \in [0, k) \bullet \neg c[f(t)/x']$*;*
4. *c becomes true at k and stays true until l:* $\forall\, t \in [k, l) \bullet c[f(t)/x']$.

Then the following equality holds:

$$(\widetilde{x}(t) \leftarrow f(t) \triangle \langle b \mid c \rangle) = \left(\widetilde{x}(t) \underset{[k..l]}{\leftarrow} f(t)\right)$$

This theorem shows the conditions under which a function evolution, with a given invariant and preemption condition, will terminate. The first two assumptions ensure that the invariant b is true initially, and remains true until l. The remaining two assumptions state that c was not true for some period, until k at which point it becomes true and stays true until l. This being the case, the preemption will occur nondeterministically at some point between k and l. A special case is when $k = l$, in which case there is precisely one instant when this occurs. This theorem is useful in languages like Modelica where the evolution of a differential equation can be halted when a specific condition is reached.

6.4 Derivatives and Ordinary Differential Equations

The ability to express derivatives of continuous variables is central to hybrid system modelling. In the hybrid relational calculus we introduce the notation x ***has-der*** $f(t)$ which states that the derivative of continuous variable x is determined by expression f, which is parametrised over time t. This is equivalent to the usual calculus notation $\dot{x}(t) = f(t, x)$. For example, we can write constraints like x ***has-der*** $2 \cdot x$, which states that x is changing at the rate of $2 \cdot x$.

A system of ODEs, $\dot{x}(t) = f(t, x(t))$, specifies a family of continuous solution functions, $x : \mathbb{R}_{\geq 0} \to \mathbb{R}^n$, that specify the value for the n variables at each instant. The system is defined by function $f : \mathbb{R}_{\geq 0} \times \mathbb{R}^n \to \mathbb{R}^n$ that gives the derivative of each variable at time t, and depends on the initial value, that is $x(t)$. A solution is any function x that changes at the rate specified by f.

Naturally, when animating or verifying a system, a single solution is normally desired. For this, it is necessary to construct an initial value problem (IVP) that supplements the system of ODEs with initial values for all continuous variables. Then the Picard-Lindelöf theorem [32] can be applied to show that, provided f is Lipschitz continuous, a unique solution exists to the initial value problem [33]. Lipschitz continuity essentially limits the rate at which a continuous function can change. We now describe our operator for systems of ODEs:

Definition 6.11. $\langle\, \dot{x}(t) \bullet f(t, x) \,\rangle \triangleq (\textbf{\textit{II}}_x \wedge x$ ***has-der*** $(f(t)(x)))$ 

The operator takes two parameters: $x : \mathbb{R}^n \Longrightarrow \Sigma_c$, which is a lens projecting a vector of reals from the continuous state; and f, the ODE specification function described above. The definition applies the initial value coupling invariant, and asserts that lens x has the derivative given by the characteristic ODE function f. It does not apply the final state coupling invariant, ***rI***, as a system of ODEs only produces a final value when it is preempted. Usually, though not necessarily,

ODEs are guarded by the $\triangle \langle b \mid c \rangle$ operator. Every operator of which the ODE operator is composed is ***R1*** and ***R2***, and thus it is a healthy reactive relation.

In order to solve differential equations, it is necessary to set up an IVP. The following theorem shows how a solution may be used to transform an ODE to symbolic solution function evolution.

Theorem 6.12. *If, for any $v : \mathbb{R}^n$ and $l > 0$, $g(v)$ is the unique solution to f on the interval $[0, l]$, and $g(v)(0) = v$ then*

$$\langle\, \dot{x}(t) \bullet f(t, x) \,\rangle = \widetilde{x}(t) \leftarrow g(x)(t)$$

This theorem allows us to transform a differential equation into a solution function evolution. It has some subtleties that require further explanation. Function $g : \mathbb{R}^n \to \mathbb{R} \to \mathbb{R}^n$ is the solution function, but it depends on the initial value for variables which is why it has two inputs. This allows us to abstract from IVPs when symbolically solving an ODE. Thus, we require that for any given initial valuation of the continuous state v, $g(v)$ is the unique solution to f. Moreover, we require that the function's value at time 0 be the initial value we have supplied; a kind of sanity check for the function. If all these conditions are satisfied then the ODE can be rewritten to $\widetilde{x}(t) \leftarrow g(x, t)$. The x on the right hand side of the arrow is the initial value of x, as usual for the relational calculus. Thus, the solution function is fully described when an initial value is supplied by a preceding assignment, for example by use of Theorem 6.8.

In terms of showing that a function is a unique solution, it suffices to show that the function is a solution and then to exhibit an appropriate Lipschitz constant. In Isabelle/HOL the former of these two can be accomplished through a tactic we have written called **ode-cert** that certifies a solution to an ODE by applying derivative introduction rules.

To exemplify, we give the following calculation of the first step of Example 2.2:

Example 6.13 (ODE Calculation).

$$\begin{aligned}
& h, v := 2, 0 \,;\, \left\langle (\dot{h}(t), \dot{v}(t)) \bullet (v, -g) \right\rangle \\
= {} & h, v := 2, 0 \,;\, \begin{pmatrix} h(t) \\ v(t) \end{pmatrix} \leftarrow \begin{pmatrix} v \cdot t - g \cdot t^2/2 + h \\ v - g \cdot t \end{pmatrix} && (6.12) \\
= {} & \begin{pmatrix} h(t) \\ v(t) \end{pmatrix} \leftarrow \begin{pmatrix} 0 \cdot t - g \cdot t^2/2 + 2 \\ 0 - g \cdot t \end{pmatrix} && (6.8)
\end{aligned}$$

We first obtain the unique solution to the ODEs, which can be done using a typical computer algebra tool like Mathematica, and then rewrite this to a function evolution. We also push forward the assignment using Theorem 6.8 to set initial values for continuous variables.

6.5 Hybrid Reactive Contracts

Whilst hybrid relations can be used to model programs, they do not allow us to distinguish terminating, non-terminating, and divergent behaviours[5]. Specif-

[5] A concept capturing erroneous behaviours such as unproductive non-termination.

ically, a dynamically evolving ODE, $\langle\, \dot{x}(t) \bullet f(t) \,\rangle$ can continue indefinitely. In spite of this, it does not satisfy the following theorem [2]:

$$\langle\, \dot{x}(t, x) \bullet f(t) \,\rangle \,;\, P \;=\; \langle\, \dot{x}(t, x) \bullet f(t) \,\rangle$$

For example, if P is an assignment, $x := v$, then the results of it are observable in such a composition. This is the reason that we often place ODEs in the context of a preemption operator[6], which correctly handles termination. This issue is analogous to the well-known problem with basic relational model of programs, which motivated the UTP theory of designs [7,34].

Our solution, similarly, is to introduce a UTP theory of reactive contractual specifications, and use the *ok* and *wait* observational variables to distinguish divergent and intermediate observations. This approach captures non-termination in reactive systems in a way that avoids complex reasoning associated with infinite traces. In previous work [25,26,35] we developed a UTP theory of generalised reactive designs, building on our theory of reactive relations [8] (§4) and prior work with *Circus* [17,16]. We use this to develop a contract notation, and a method for automatically calculating the semantics of reactive programs for the purpose of verification [26]. As for designs, our reactive contracts also support refinement with assume/guarantee style reasoning [36,37]. It allows a unified set of laws for a diverse set of languages, including CSP [7], *Circus* [17], timed extensions [38], and of course our hybrid relations.

A reactive contract, $[\, P_1 \vdash P_2 \mid P_3 \,]$, consists of three reactive relations that specify the (1) assumption, P_1, and (2) guarantee for intermediate observations, P_2, and terminating observations, P_3. Assumption P_1 is a reactive condition (Definition 4.2): it can refer only the initial state (st) and trace (tt), and the characterised set of traces must be prefix closed. If the assumption of a contract is violated, then the result is the most nondeterministic reactive designs, called $\textbf{\textit{Chaos}} \triangleq [\, \textbf{\textit{false}} \vdash \textbf{\textit{false}} \mid \textbf{\textit{false}} \,]$, which corresponds to divergent behaviour. P_2 characterises the intermediate or waiting observations of the reactive program; consequently it is a reactive relation that, like P_1, refers only to st and tt. P_3 characterises terminating observations, and so can refer to st, tt, and also st'.

We can now lift our ODE operator so that it is correctly non-terminating:

Definition 6.14. $\langle\!\langle\, \dot{x}(t) \bullet f(t, x) \,\rangle\!\rangle \triangleq [\, \textbf{\textit{true}}_r \vdash \langle\, \dot{x}(t) \bullet f(t, x) \,\rangle \mid \textbf{\textit{false}} \,]$

The assumption of the lifted ODE is true, since there is no divergent behaviour. The terminating guarantee is ***false***, since this is a non-terminating operator. The intermediate guarantee is simply our hybrid relational ODE operator, so that evolutions of the ODE are flagged as intermediate obervations. Then, we can use the following contract theorems for reasoning about compositions [25,26]:

Theorem 6.15 (Reactive Design Laws).

$$[\, P_1 \vdash P_2 \mid P_3 \,] \,;\, [\, \textbf{\textit{true}}_r \vdash Q_2 \mid Q_3 \,] \;=\; [\, P_1 \vdash P_2 \vee P_3 \,;\, Q_2 \mid P_3 \,;\, Q_3 \,] \quad (6.15.1)$$

$$\textbf{\textit{Chaos}} \sqcap [\, P_1 \vdash P_2 \mid P_3 \,] \;=\; \textbf{\textit{Chaos}} \quad (6.15.2)$$

[6] This is also true of dℒ hybrid programs, which are modelled similarly.

$$[\, P_1 \vdash P_2 \mid \textbf{\textit{false}} \,] \,;\, [\, Q_1 \vdash Q_2 \mid Q_3 \,] \;=\; [\, P_1 \vdash P_2 \mid \textbf{\textit{false}} \,] \tag{6.15.3}$$

$$\textbf{\textit{Chaos}} \,;\, [\, P_1 \vdash P_2 \mid P_3 \,] \;=\; \textbf{\textit{Chaos}} \tag{6.15.4}$$

$$[\, \textbf{\textit{false}} \vdash P_2 \mid P_3 \,] \;=\; \textbf{\textit{Chaos}} \tag{6.15.5}$$

(6.15.1) is the basic law for sequential composition[7]. When composing contracts in the sequence, an intermediate observation is either an intermediate observation of the first contract (P_2), or a terminating observation of the first, followed by an intermediate observation of the second (P_3 ; Q_2). A terminating observation requires that both contracts can terminate P_3 ; Q_3. (6.15.2) show that ***Chaos*** is indeed the most nondeterministic reactive contract. The remaining laws are essentially corollaries of (6.15.1). Of particular interest for ODEs is (6.15.3), which has the following law as a consequence:

Theorem 6.16. $\langle\!\langle\, \dot{x}(t) \bullet f(t) \,\rangle\!\rangle \,;\, P \;=\; \langle\!\langle\, \dot{x}(t) \bullet f(t) \,\rangle\!\rangle$ 

Similar results can be achieved for the function evolution operators.

A further advantage of hybrid reactive contracts is to encode assumptions about continuous variables outside of the system's control (e.g. monitored variables). Assumptions can, for example, be specified using the interval operator of Definition 6.3, since this constructs reactive conditions. For example, we can specify a division block for the control law languages of Modelica or Simulink:

Example 6.17. $\mathsf{Div}(x, y, z) \triangleq [\, \lceil y \neq 0 \rceil \vdash \lceil z = x/y \rceil \mid \textbf{\textit{false}} \,]$

We encode a division block with two inputs, x and y, and a single output z. These are all modelled as lenses into the continuous state ($\mathbb{R} \Longrightarrow \Sigma_c$), that correspond to connections in a block diagram, and are given as parameters. The intuition here is that every wire in a control law diagram is modelled as a lens. The divison block is a non-terminating hybrid process that in every intermediate state requires that the continuous variable z take the value of x/y. The assumption requires that $y \neq 0$, to ensure that division by zero cannot occur. Using a pattern like this, we can give semantics to a large number of blocks in the Simulink and Modelica block libraries[8]. This allows us to use hybrid reactive designs to reason about control law diagrams.

7 Mechanisation and Example

The hybrid relational calculus, and the theorems described in §6 are mechanised in Isabelle/UTP. For this, we employ Isabelle's implementation of multivariate analysis [39], including its symbolic real numbers, Euclidean spaces, limits, and derivatives. We also utilise Immler's library for ODEs and IVPs [33,40], which allows us to certify that a function is the solution to a system of ODEs.

In order to exemplify the use of the mechanisation, we describe part of a tram model, which is part of a previous industrial case study [14]. We reproduce it here

[7] For brevity, we present a simplified law where the second assumption is $\textbf{\textit{true}}_{\mathsf{r}}$.

[8] See https://build.openmodelica.org/Documentation/Modelica.Blocks.html

for the purposes of illustration, with adaptation for our new hybrid relational model. We focus on the situation when the tram is slowing due to an approaching red signal, and formalise this using variables for acceleration *acc*, velocity *vel*, and track position *pos*. We note that *normal-deceleration* below is negative and determines the rate at which the tram reduces its speed as the brakes are applied.

Definition 7.1 (Braking Tram in Hybrid Relational Calculus).

$$BrakingTrain \triangleq \left(\begin{array}{l} acc, vel, pos := normal\text{-}deceleration, max\text{-}speed, 0 \ ; \\ \left\langle \begin{pmatrix} \dot{acc} \\ \dot{vel} \\ \dot{pos} \end{pmatrix} \bullet \begin{pmatrix} 0 \\ \dot{acc} \\ \dot{vel} \end{pmatrix} \right\rangle \triangle \langle vel > 0 \mid vel \leq 0 \rangle \ ; \\ acc := 0 \end{array} \right)$$

We assign initial values to the continuous variables, and then evolve them until the velocity reaches 0. In this instance, we do not allow non-determinism here, but record the precise instant that the velocity is 0. Thus, the evolution invariant is $vel > 0$, and the preemption condition is $vel \leq 0$. After this, we set the acceleration to 0, so that the tram halts and does not start moving backwards. Though this model is highly idealised, a more realistic model, which, for example, introduces pertubations into the acceleration due to external influences like weather, can be described by adding periodic preemption conditions and non-deterministic assignments to corresponding variables.

This example is encoded in Isabelle/UTP, as shown in Figure 2, where the preemption operator has the syntax $P \ \textbf{\textit{inv}} \ b \ \textbf{\textit{until}}_h \ c$. We also mechanise a proof that the train stops before the end of the track, that is,

Theorem 7.2. $(accl' = 0 \wedge \mathit{dur}[pos < 44]) \sqsubseteq BrakingTrain$ 

holds, where $44m$ is the track length. The specification to the left states that, for all possible evolutions, the final value of the acceleration is 0 and *pos* is always less than 44. This should then be refined by our hybrid relation, *BrakingTrain*. For the sake of brevity, we elide details of the proof in Isabelle, other than the first four steps. The proof proceeds as follows:

1. Solve the ODE to obtain a function evolution statement (Theorem 6.12);
2. Use the assigned values to obtain the initial conditions (Theorem 6.8);
3. Calculate the time at which the velocity reaches zero (Theorem 6.10);
4. Finally, prove that the position at every earlier instant is less than 44 metres.

The final step requires that we solve a polynomial inequality:

$$(104/25) \cdot t - (7/10) \cdot t^2 < 44$$

which includes the position derivative solution. In Isabelle, this can be done using the approximate tactic [41], which applies floating-point computation.

```
definition "BrakingTrain =
   (c:accel, c:vel, c:pos) :=ᵣ («normal_deceleration», «max_speed», «0») ;;
   ⟨{&accel,&vel,&pos} • train_ode(ti)⟩ₕ inv ¬$vel´≤ᵤ0 untilₕ ($vel´≤ᵤ0) ;; c:accel :=ᵣ 0"

theorem braking_train_pos_le:
 "($st:c:accel´ =ᵤ 0 ∧ ⌈$pos´ <ᵤ 44⌉ₕ) ⊑ BrakingTrain" (is "?lhs ⊑ ?rhs")
proof -
  — ‹ Solve ODE, replacing it with an explicit solution: @{term train_sol}. ›
  have "?rhs =
    (c:accel, c:vel, c:pos) :=ᵣ («-1.4», «4.16», «0») ;;
    {&accel,&vel,&pos} ←ₕ «train_sol»($accel,$vel,$pos)ₐ(«ti»)ₐ untilₕ ($vel´ ≤ᵤ 0) ;;
    c:accel :=ᵣ 0"
  by (simp only: BrakingTrain_def train_sol)
  — ‹ Set up initial values for the ODE solution using assigned variables. ›
  also have "... =
    {&accel,&vel,&pos}←ₕ«train_sol(-1.4,4.16,0)(ti)» untilₕ ($vel´≤ᵤ0) ;; c:accel :=ᵣ 0"
    by (rel_auto)
  — ‹ Find the point at which the train stops ›
  also have "... =
    (({&accel,&vel,&pos} ←ₕ(«416/140») «train_sol(-1.4,4.16,0)(ti)»)) ;; c:accel :=ᵣ 0"
```

Fig. 2. The braking tram in Isabelle/UTP

8 Conclusions and Discussion

We have described our UTP theory of hybrid relations that specialises reactive relations with a continuous timed trace model. A key result is the unification of hybrid models [2,1,13] and reactive programs [26], through our generalised theories of reactive relations and reactive designs. In a parallel development, we have used the generalised theory to mechanise a semantics and verification tool for *Circus* [26,17] (and thus CSP [42]), and our hybrid theory shares many of the laws, such as those in Theorem 6.15. This, we believe, shows the immense and practical value of unification. Our theory can also be used as a foundation for automated verification tools for hybrid programs in Isabelle/UTP [10], and we plan to apply it to verification of Modelica dynamical models, by extending our previous semantics [13] that used an early version of our UTP hybrid theory.

The two most related works are differential dynamic logic [1,4] (d$\mathcal{L}$), and HCSP [2,3,5], both of which have substantially influenced our direction.

Our model is more expressive than standard d$\mathcal{L}$ hybrid programs, since we encode an explicit trajectory, whilst d$\mathcal{L}$ encodes the initial/intermediate value pairs for each variable in a binary relation. This allows us to separate ODEs from preemption, which in d$\mathcal{L}$ are combined in a single operator, $\{x' = \theta \,\&\, b\}$, where b is the boundary condition. This can be useful when constructing systems by composition of continuous and discrete components, where b is not known *a priori*. An explicit trace model is also a prerequisite for modelling networks of communicating hybrid systems [2]. There is also a d$\mathcal{L}$ extension called dTL^2 [43] that also employs an explicit trajectory and is similar to our model.

Proof support in d$\mathcal{L}$'s tool, KeYmaera X [4], is cleary far more advanced than our implementation in Isabelle/UTP. Nevertheless, we are currently working on implementing d$\mathcal{L}$ in Isabelle/UTP[9], based on a recent implementation of d$\mathcal{L}$ in

[9] Differential dynamic logic in Isabelle/UTP

Isabelle/HOL [44], and hope to report on this soon. This will allow to formally link the two theories, and also extensions like [43], via Galois connections, and harness the differential induction reasoning technique.

HCSP [2,3] models communicating hybrid systems using CSP-style process algebraic operators. There are two main denotational semantic models, the original one by He [2], which employs a UTP-style relational calculus, and a later one by Zhou [3], that employs Duration Calculus [30,29]. Our model is comparable to, though less expressive than [2] — since [2] models a more sophisticated form of trajectory based on super-dense time [45,15] — and is likely of equivalent expressivity with [3]. The semantics and algebraic laws in [2] are a strong inspiration for our work, and we believe that [2] is very similar to our reactive designs. For super-dense time [45,15], the trajectory has type $\mathbb{R}_{\geq 0} \times \mathbb{N} \nrightarrow \Sigma_c$ – the time domain is extended with a natural number that allows state changes that are "simultaneous-but-ordered". This is, arguably, needed to allow CSP-style events that are often interpreted to take a zero time duration. We hope in the future to explore whether such a trajectory model forms a trace algebra [8], so that our reactive designs hierarchy can be reused. Moreover, we will also explore weakening the trace algebra to support infinite traces which are at present forbidden.

In conclusion, the UTP approach has been an invaluable tool in this development. Whilst several hybrid computational theories exist, there are links between them, which UTP theories allow us to explore. Moreover, UTP allows us to link to theories that at first sight seem unrelated, such as *Circus* [17], as our reactive design theory shows. Our overarching message is this: *the UTP works* — it can capture languages of differing and heterogeneous paradigms and use the associated theories to develop and integrate verification tools. As Hoare and He reflected in the first chapter of the UTP book, when considering all the tools and artefacts that software engineering research is producing:

> "...to ensure that [analysis] tools may be safely used in combination, it is essential that these [underlying] theories be unified..." [7, page 21]

We believe that our hierarchy of theories and verification tools in Isabelle/UTP is evidence that UTP supports a practical approach for integration of formal analysis tools [46]. As systems become more complex in nature, as is the case with cyber-physical systems and autonomous robots, there is an even greater need to consider integration of heterogeneous computational paradigms [6]. The UTP allows us to approach one of the grand challenges for software engineering: integration of formal methods [6,46,47] for assurance of large-scale systems.

Acknowledgments

This work is funded by the CyPhyAssure project[10], EPSRC grant EP/S001190/1. We would like to thank the anonymous reviewers for their thorough and helpful input, which has improved the presentation of our work.

[10] CyPhyAssure Project: https://www.cs.york.ac.uk/circus/CyPhyAssure/

References

1. Platzer, A.: Differential dynamic logic for hybrid systems. J. Autom Reasoning **41** (June 2008) 143–189
2. He, J.: From CSP to hybrid systems. In Roscoe, A.W., ed.: A classical mind: essays in honour of C. A. R. Hoare. Prentice Hall (1994) 171–189
3. Zhou, C., Ji, W., Ravn, A.P.: A formal description of hybrid systems. In Alur, R., Henzinger, T.A., Sontag, E.D., eds.: Hybrid Systems III. Volume 1066 of LNCS. Springer (1996) 511–530
4. Fulton, N., Mitsch, S., Quesel, J.D., Völp, M., Platzer, A.: KeYmaera X: An axiomatic tactical theorem prover for hybrid systems. In: CADE-25. Volume 9195 of LNCS., Springer (2015) 527–538
5. Wang, S., Zhan, N., Zou, L.: An improved HHL prover: An interactive theorem prover for hybrid systems. In: ICFEM. Volume 9407 of LNCS., Springer (2015) 382–399
6. Gleirscher, M., Foster, S., Woodcock, J.: New opportunities for integrated formal methods. ACM Computing Surveys (2019) Accepted subject to minor revision. Preprint: https://arxiv.org/abs/1812.10103.
7. Hoare, C.A.R., He, J.: Unifying Theories of Programming. Prentice-Hall (1998)
8. Foster, S., Cavalcanti, A., Woodcock, J., Zeyda, F.: Unifying theories of time with generalised reactive processes. Information Processing Letters **135** (2018) 47–52
9. Foster, S., Zeyda, F., Nemouchi, Y., Ribeiro, P., Wolff, B.: Isabelle/UTP: Mechanised Theory Engineering for Unifying Theories of Programming. Archive of Formal Proofs (2019) https://www.isa-afp.org/entries/UTP.html.
10. Foster, S., Baxter, J., Cavalcanti, A., Woodcock, J., Zeyda, F.: Unifying semantic foundations for automated verification tools in Isabelle/UTP. Submitted to Science of Computer Programming (March 2019) Preprint: https://arxiv.org/abs/1905.05500.
11. Foster, S., Zeyda, F., Woodcock, J.: Unifying heterogeneous state-spaces with lenses. In: ICTAC. LNCS 9965, Springer (2016)
12. Foster, S., Zeyda, F., Woodcock, J.: Isabelle/UTP: A mechanised theory engineering framework. In: UTP. Volume 8963 of LNCS., Springer (2014) 21–41
13. Foster, S., Thiele, B., Cavalcanti, A., Woodcock, J.: Towards a UTP semantics for Modelica. In: UTP. LNCS 10134, Springer (2016)
14. Zeyda, F., Ouy, J., Foster, S., Cavalcanti, A.: Formalising Cosimulation Models. In: Proc. CoSim-CPS 2017. Volume 10729 of LNCS., Springer (2017) 453–468
15. Lee, E.A.: Constructive models of discrete and continuous physical phenomena. IEEE Access **2** (August 2014) 797–821
16. Cavalcanti, A., Woodcock, J.: A tutorial introduction to CSP in unifying theories of programming. In: PSSE. Volume 3167 of LNCS. Springer (2006) 220–268
17. Oliveira, M., Cavalcanti, A., Woodcock, J.: A UTP semantics for Circus. Formal Aspects of Computing **21** (2009) 3–32
18. Back, R.J., Wright, J.: Refinement Calculus: A Systematic Introduction. Springer (1998)
19. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Communications of the ACM **18**(8) (1975) 453–457
20. Feliachi, A., Gaudel, M.C., Wolff, B.: Unifying theories in Isabelle/HOL. In: UTP 2010. Volume 6445 of LNCS., Springer (2010) 188–206
21. Feliachi, A., Gaudel, M.C., Wolff, B.: Isabelle/Circus: a process specification and verification environment. In: VSTTE 2012. Volume 7152 of LNCS., Springer (2012) 243–260

22. Boulton, R., Gordon, A., Gordon, M., Harrison, J., Herbert, J., van Tassel, J.: Experience with embedding hardware description languages in HOL. In: Proc. IFIP Intl. Conf. on Theorem Provers in Circuit Design. (1993) 129–156
23. Gomes, V.B.F., Struth, G.: Modal Kleene algebra applied to program correctness. In: Formal Methods. Volume 9995 of LNCS., Springer (2016) 310–325
24. Foster, J.: Bidirectional programming languages. PhD thesis, University of Pennsylvania (2009)
25. Foster, S., Cavalcanti, A., Canham, S., Woodcock, J., Zeyda, F.: Unifying theories of reactive design contracts. Under revision for Theoretical Computer Science (Dec 2017) Preprint: https://arxiv.org/abs/1712.10233.
26. Foster, S., Ye, K., Cavalcanti, A., Woodcock, J.: Calculational verification of reactive programs with reactive relations and Kleene algebra. In: Proc. 17th Intl. Conf. on Relational and Algebraic Methods in Computer Science (RAMICS). Volume 11194 of LNCS., Springer (October 2018)
27. Höfner, P., Möller, B.: An algebra of hybrid systems. Journal of Logic and Algebraic Programming **78**(2) (2009) 74–97
28. Cavalcanti, A., Foster, S., Thiele, B., Woodcock, J., Zeyda, F.: Final Semantics of Modelica. Technical report, INTO-CPS Deliverable, D2.3b (December 2017)
29. Zhou, C., Ravn, A.P., Hansen, M.R.: An extended Duration Calculus for hybrid real-time systems. In Grossman, R.L., Nerode, A., Ravn, A.P., Rischel, H., eds.: Hybrid Systems. Volume 736 of LNCS. Springer (1993) 36–59
30. Zhou, C., Hoare, C.A.R., Ravn, A.P.: A calculus of durations. Information Processing Letters **40**(5) (1991) 269–276
31. Henzinger, T.A. In: The theory of hybrid automata. IEEE (1996) 278–292
32. Coddington, E.A., Levinson, N.: Theory of Ordinary Differential Equations. McGraw-Hill (1955)
33. Immler, F., Hölzl, J.: Numerical analysis of Ordinary Differential Equations in Isabelle/HOL. In: 3rd Intl. Conf. on Interactive Theorem Proving (ITP). Volume 7406 of LNCS., Springer (2012) 377 – 392
34. Cavalcanti, A., Woodcock, J.: A tutorial introduction to designs in unifying theories of programming. In: Proc. 4th Intl. Conf. on Integrated Formal Methods (IFM). Volume 2999 of LNCS., Springer (2004) 40–66
35. Foster, S., Baxter, J., Cavalcanti, A., Miyazawa, A., Woodcock, J.: Automating verification of state machines with reactive designs and Isabelle/UTP. In: Proc. 15th. Intl. Conf. on Formal Aspects of Component Software. Volume 11222 of LNCS., Springer (October 2018)
36. Meyer, B.: Applying "design by contract". IEEE Computer **25**(10) (1992) 40–51
37. Benveniste, A., Caillaud, B., Ferrari, A., Mangeruca, L., Passerone, R., Sofronis, C.: Multiple viewpoint contract-based specification and design. In: 6th Intl. Symp. on Formal Methods for Components and Objects (FMCO). Volume 5382 of LNCS., Springer (2007) 200–225
38. Sherif, A., Cavalcanti, A., He, J., Sampaio, A.: A process algebraic framework for specification and validation of real-time systems. Formal Aspects of Computing **22**(2) (2010) 153–191
39. Harrison, J.: A HOL theory of Euclidean space. In Hurd, J., Melham, T., eds.: Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005. Volume 3603 of LNCS., Oxford, UK, Springer (2005)
40. Immler, F.: Formally verified computation of enclosures of solutions of Ordinary Differential Equations. In: Proc. 6th NASA Formal Methods Symposium (NFM). Volume 8430 of LNCS., Springer (2014)

41. Hölzl, J.: Proving inequalities over reals with computation in isabelle/hol. In: Proc. 2009 Intl. Workshop on Programming Languages for Mechanized Mathematics Systems (PLMMS), ACM (August 2009) 38–45
42. Brookes, S.D., Hoare, C.A.R., Roscoe, A.W.: A theory of communicating sequential processes. Journal of the ACM **31**(3) (1984) 560–599
43. Jeannin, J.B., Platzer, A.: dTL^2: Differential Temporal Dynamic Logic with Nested Temporalities for Hybrid Systems. In: IJCAR. Volume 8562 of LNAI., Springer (2014)
44. Huerta y Munive, J.J., Struth, G.: Verifying hybrid systems with modal Kleene algebra. In: Proc. 17th Intl. Conf. on Relational and Algebraic Methods in Computer Science (RAMICS). Volume 11194 of LNCS., Springer (October 2018)
45. Manna, Z., Pneuli, A.: Verifying hybrid systems. In: Hybrid Systems. Volume 736 of LNCS., Springer (1993)
46. Paige, R.F.: A meta-method for formal method integration. In: Proc. 4th. Intl. Symp. on Formal Methods Europe (FME). Volume 1313 of LNCS., Springer (1997) 473–494
47. Galloway, A.J., Stoddart, B.: Integrated formal methods. In: Proc. INFORSID, INFORSID (1997)