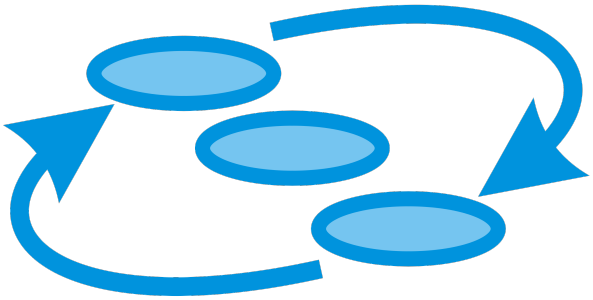
Grant Agreement: 644047

INtegrated TOol chain for model-based design of CPSs



INTO-CPS

Final Semantics of Modelica

Technical Note Number: D2.3b

Version: 1.2

Date: December 2017

Public Document

http://into-cps.au.dk

Contributors:

Ana Cavalcanti, UY
Simon Foster, UY
Jim Woodcock, UY
Frank Zeyda, UY

Editors:

Simon Foster, UY

Reviewers:

Ken Pierce, UNEW
Hassan Ridouane, UTRC
Bernhard Thiele, LIU

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	15-05-2017	Simon Foster	Initial document version
1.0	01-11-2017	Simon Foster	Internal review version completed
1.1	11-12-2017	Simon Foster	Addressed comments from all three reviewers
1.2	14-12-2017	Simon Foster	Final version for submission

Abstract

This deliverable reports on the final version of our continuous-time semantics for the dynamical systems modelling language, Modelica, in the context of Hoare and He's Unifying Theories of Programming (UTP). Modelica is a language for modelling a system's continuous behaviour using a combination of differential-algebraic equations and an event-handling system. We have previously given a high-level semantics to the Modelica event handling mechanism in terms of the hybrid relational calculus. In this deliverable, we provide an improved semantic model for the calculus, based on the theory of reactive processes that is typically used to give semantics to concurrent systems. We provide a generalised trace model for reactive processes, and show how it can encompass both discrete and continuous traces. We then reconstruct the hybrid relational calculus, which has also been mechanised in Isabelle/UTP, along with a number of additional operators, and new facilities for reasoning about differential equations. We also present a theory of reactive design contracts, which is integrated with our hybrid relational calculus to give a semantic model for concurrent and reactive hybrid systems. Finally, we bring all the theory developed to construct a more detailed semantics for Modelica, with a particular focus on the Modelica block language.

Contents

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 [29] (UTP) has been designed as a framework in which the integration of languages, through the common semantic domain of the alphabetised relational 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. We have previously shown how to give a high-level denotational semantics to Modelica using a UTP theory of hybrid relations [10, 19]. Our hybrid relational calculus provides operators for describing the behaviour of continuous variables in continuous and hybrid systems.

In this deliverable we provide a substantial upgrade to the hybrid relational calculus, and then use this foundation to construct a more detailed semantics of Modelica. This upgrade involves, on the one hand, a more general semantic model that is better integrated with other UTP theories of concurrency and reactivity. On the other hand, it is also combined with a novel theory of reactive design contracts, which we also construct in this deliverable. Whereas the hybrid relational calculus is insensitive to non-termination, our theory of contracts allows the description of reactive hybrid systems. Moreover, these contracts allow us to specify hybrid behaviour using an assertional reasoning style notation, commonly found in formalisms like Hoare logic and refinement calculus. Thus, our contract theory provides a way in to formal verification of hybrid systems.

Construction of the semantics of Modelica in UTP, thus, requires us to construct a number of foundational layers, which we enumerate below. The dependencies between these constituent UTP theories is also illustrated in Figure 1.

1. In Section 2 we review preliminary material necessary for the remainder of the deliverable, specifically the various key theories of the UTP and proof environment Isabelle/UTP [21].
2. In Section 3 we introduce our UTP theory of generalised reactive processes (GRP), which enriches the semantic model of concurrent and reactive systems present in the UTP to support continuous variables. This includes a language of relations and conditions which are a precursor of our contract notation.
3. In Section 4 we describe our UTP theory of hybrid relations (HRC), which is a substantial upgrade of the theory we delivered in D2.2c [10]. This is now founded entirely on our theory of generalised reactive processes, and is fully mechanised in Isabelle/HOL. The latter enables theorem proving support for hybrid systems.

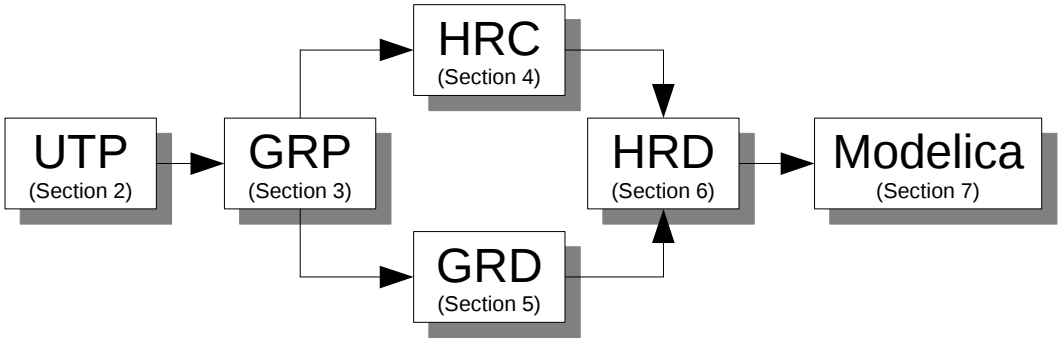


Figure 1: D2.3b Semantics Dependencies

4. In Section 5 we describe our theory of reactive design contracts (GRD), which further extend the theory of generalised reactive processes to support contracting with preconditions, postconditions, and also "pericondition" that describe the permissible intermediate behaviour of a process. This theory in particular provides assertional reasoning foundations for hybrid systems.
5. In Section 6 we combine the results from Sections 4 and 5 to construct hybrid reactive design contracts (HRD), including specialised operators for describing differential equations and preemption, as in the hybrid relational calculus.
6. In Section 7 we bring together all the results described above to construct a denotational semantics for the Modelica language, with a particular focus on Modelica blocks.

The focus of this deliverable is principally on theory that underlies hybrid and cyber-physical systems. However, the results from this deliverable are not only theoretical, but also practical. Deliverables D2.3a [54] and D2.3d [11] present a verification technique for FMI that utilises many results from this deliverable. This technique has been practically applied in the context of WP1 and WP3 to verify both pilot studies and industrial case studies, which demonstrates how these theoretical results add value to the INTO-CPS methodology.

2 Preliminaries

In this section we briefly go over preliminaries of the UTP and core theories. These provide the foundation upon which we will build the theory hierarchy that will be used to give the denotational semantics to Modelica.

2.1 UTP

The UTP favours an approach to formal semantics that grasps at Platonic "universals" of programming language paradigms. It seeks the fundamental abstract ideas that exist as foundations of programming language semantics and formalise them using UTP

theories. For example, a collection of healthiness conditions can describe what it means for a language to be concurrent [29, 9], real-time [47], or object-oriented [45]. The UTP thus promotes reuse of theoretical building blocks that underlie programming language semantics.

UTP is based on an alphabetised relational calculus with operators of higher-order predicate calculus and relation algebra. Relations are simply predicates whose alphabet consists of pairs of variables that denote initial values (x) and final values (x'). The domain of alphabetised relations forms a number of important algebraic structures including (1) a complete lattice [29], where the order is refinement, ***false*** is top, and ***true*** is bottom; (2) a relation algebra [49, 20]; (3) a cylindric algebra [26, 21]; and (4) a quantale [21], which induces (5) a Kleene algebra [1]. Together these provide a rich set of base properties supporting program verification [1, 21].

UTP theories are specified in terms of three parts:

1. an alphabet of typed *observational variables*, which are used to encode observable semantic quantities important for the theory;
2. a *healthiness condition* (***HC***), specified as a function on predicates with the above alphabet;
3. a *signature*: that is, a set of constructors and other functions for healthy elements of the theory.

A theory's alphabet is often open to extension, such that additional observational variables can be added, or the types of variables specialised, assuming an appropriate notion of polymorphism. This also means that UTP theories can readily be combined by merging the alphabets and composing the healthiness conditions.

The image of the healthiness condition gives rise to the UTP theory's domain. For this reason it is necessary that the functions are idempotent ($\mathit{HC} \circ \mathit{HC} = \mathit{HC}$), and usually also monotonic. Monotonicity ensures that that the UTP theory forms a complete lattice, substantiated by the Knaster-Tarski theorem [48]. This gives rise to a theory top ($\top_{\mathit{HC}}$), bottom ($\bot_{\mathit{HC}}$), infimum, supremum, and fixed-point operators (such as the weakest fixed-point operator μ_{HC}). The top and bottom can be obtained by applying the healthiness condition to ***false*** and ***true***, respectively. However, the induced lattice does not in general share the same operators as the alphabetised predicate lattice. Thus, for our purposes, we are interested in the stronger property **continuity**, which give rise to additional properties.

Definition 2.1 (Continuous Healthiness Conditions)**.** A healthiness condition ***HC*** is said to be continuous if it satisfies $\mathit{HC}(\bigsqcap A) = \bigsqcap\{\mathit{HC}(P) \mid P \in A\}$ for $A \neq \emptyset$.

This notion of continuity is stronger than the related notion of Scott-continuity [46], which requires that A also be directed. Every continuous healthiness condition is also monotonic and thus induces a complete lattice. Continuity also means that the theory infimum is the same operator as the alphabetised predicate infimum ($\bigsqcap$). This means that a number of additional laws can be transplanted into the theory, some of which are illustrated below.

Theorem 2.1 (Continuous Theory Laws)**.**

$$P \sqcap P = P \tag{2.1.1}$$
$$P \sqcap Q = Q \sqcap P \tag{2.1.2}$$
$$P \sqcap (Q \sqcap R) = (P \sqcap Q) \sqcap R \tag{2.1.3}$$
$$\bot_{\mathit{T}} = \mathbf{HC}(\mathbf{true}) \tag{2.1.4}$$
$$\top_{\mathit{T}} = \mathbf{HC}(\mathbf{false}) \tag{2.1.5}$$
$$\bot_{\mathit{T}} \sqcap P = \bot_{\mathit{T}} \quad \textit{provided}\ \mathbf{HC}(P) = P \tag{2.1.6}$$
$$\top_{\mathit{T}} \sqcap P = P \quad \textit{provided}\ \mathbf{HC}(P) = P \tag{2.1.7}$$
$$\mu_{\mathit{T}} X \bullet F(X) = \mu\, X \bullet F(\mathbf{HC}(X)) \tag{2.1.8}$$

Of particular interest here is Theorem 2.1.8 that shows how a theory's weakest fixed-point operator can be rewritten to the alphabetised predicate weakest fixed-point. The requirement is that the continuous healthiness condition ***HC*** can be applied after each unfolding of the fixed-point to ensure that the function F is only ever presented with a healthy predicate. Healthiness conditions in the UTP are often built from several component functions. That being the case, continuity and idempotence properties of the overall healthiness condition can be obtained by composition.

2.2 Designs

The UTP theory of designs has two observational variables, $ok, ok' : \mathbb{B}$, flags that denote whether a program was started and whether it terminated, respectively. The signature, $P \vdash Q$, states that if a program is started and the state satisfies precondition P, then it will terminate and satisfy postcondition Q. This is denoted as follows.

Definition 2.2 (Designs)**.**

$$P \vdash Q \triangleq (ok \wedge P) \Rightarrow (ok' \wedge Q)$$

Here, P and Q are relations on variables excluding ok and ok'. Effectively this encoding allows a pair of predicates to be encoded as a single predicate.

Designs have a natural notion of refinement such that $P_1 \vdash P_2 \sqsubseteq Q_1 \vdash Q_2$ if the precondition is weakened ($P_1 \Rightarrow Q_1$) and the postcondition is strengthened within the window of precondition P_1 ($Q_2 \wedge P_1 \Rightarrow P_2$). Designs form a complete lattice [29], where the bottom $\bot_D$ corresponds to an abort, arising for instance due to a violated precondition such as $\mathbf{false} \vdash P$, and top $\top_D$ corresponds to a miraculous design. The infimum $P \sqcap Q$ corresponds to a nondeterministic choice between two designs, and refinement reduces non-determinism: $P \sqcap Q \sqsubseteq P$.

Designs are closed under sequential composition, disjunction, and conjunction [29, 8], the latter of which give rise to similar properties present in contract theory [3]. The main healthiness conditions of designs are ***H*** and ***N*** which are given below.

Definition 2.3 (Design Healthiness Conditions)**.**

$$\textbf{\textit{H1}}(P) \triangleq (ok \Rightarrow P) \qquad \textbf{\textit{J}} \triangleq (ok \Rightarrow ok') \wedge \mathit{II}_{A \setminus \{ok\}}$$
$$\textbf{\textit{H2}}(P) \triangleq P \,;\, \textbf{\textit{J}} \qquad \mathit{II}_{\textbf{\textit{D}}} \triangleq \textbf{\textit{true}} \vdash \mathit{II}$$
$$\textbf{\textit{H3}}(P) \triangleq P \,;\, \mathit{II}_{\textbf{\textit{D}}} \qquad \textbf{\textit{H}} \triangleq \textbf{\textit{H1}} \circ \textbf{\textit{H2}}$$
$$\textbf{\textit{N}} \triangleq \textbf{\textit{H1}} \circ \textbf{\textit{H3}}$$

H1 states that until a design has been given permission to execute via ok, no observations are possible. ***H2*** states that no design can require non-termination. A more intuitive form that characterises fixed-points is $P[\mathit{false}/ok'] \Rightarrow P[\mathit{true}/ok']$: every non-terminating behaviour for which $ok' = \mathit{false}$ has an equivalent terminating behaviour for which $ok' = \mathit{true}$. ***H3*** designs additionally require that P is a condition: it does not refer to dashed variables. The latter subclass is useful for "normal" specifications where the precondition does not refer to the final state. ***H3*** designs, with a few notable exceptions, are the most common form of design, and are thus sometimes known as *normal designs* [22], as indicated by healthiness condition ***N***. Since every ***H3*** predicate is also ***H2*** healthy, in defining ***N***, we do not need to include ***H2*** in the composition. ***H*** and ***N*** are both idempotent and continuous, and thus both theories are complete lattices.

2.3 Reactive Processes

The theory of reactive processes [29, 9] unifies the semantics of different reactive languages. The two main goals of reactive processes are to (1) embed traces into the relational calculus, which is achieved through ***R1*** and ***R2***, and (2) introduce intermediate observations which is acheived through ***R3***. In addition to ok and ok', the theory has three pairs of observational variables $wait, wait' : \mathbb{B}$ that determine whether a process (or its predecessor) is waiting for interaction with its environment or else has terminated; tr, tr' : seq *Event* that describes the trace before and after the process' execution; and $ref, ref' : \mathbb{P}\, \mathit{Event}$ that describe the events being refused during an intermediate state, as required by the failures model of CSP [28].

Our version of reactive processes removes the ref and ref' variables to allow extension to behavioural semantic models other than failures. Moreover, we add $st, st' : \Sigma$ to explicitly model state as suggested by [5], where Σ is a suitable record type. In our previous work [18] we have shown how the UTP theory of reactive processes can be generalised by characterising the trace model with an abstract algebra, called a "trace algebra". A trace algebra $(\mathcal{T}, \frown, \epsilon)$ is a form of cancellative monoid that axiomatises operators for trace concatenation $(x \frown y)$, trace prefix $(x \leq y)$, and trace difference $(x - y)$. The trace algebra will be discussed in more detail in Section 3.

From these algebraic foundations we have reconstructed the complete theory of reactive processes, including its healthiness conditions and associated laws, in particular those for sequential and parallel composition. We thus generalise the type of tr and tr' to be an instance of a suitable trace algebra $\mathcal{T}$, and recreate the three reactive healthiness conditions, with some modifications.

Definition 2.4 (Stateful Reactive Healthiness Conditions)**.**

$$\begin{aligned}
\textbf{\textit{R1}}(P) &\triangleq P \wedge tr \leq tr' \\
\textbf{\textit{R2}}_c(P) &\triangleq P[\langle\rangle, \textbf{\textit{tt}}/tr, tr'] \lhd tr \leq tr' \rhd P \\
\textbf{\textit{R3}}_h(P) &\triangleq \mathit{II}_{\textbf{\textit{R}}} \lhd \mathit{wait} \rhd P \\
\mathit{II}_{\textbf{\textit{R}}} &\triangleq ((\exists\, st \bullet \mathit{II}) \lhd \mathit{wait} \rhd \mathit{II}) \lhd \mathit{ok} \rhd \textbf{\textit{R1}}(\textbf{\textit{true}}) \\
\textbf{\textit{tt}} &\triangleq (tr' - tr) \\
\textbf{\textit{R}}_s &\triangleq \textbf{\textit{R1}} \circ \textbf{\textit{R2}}_c \circ \textbf{\textit{R3}}_h
\end{aligned}$$

R1 states that tr is monotonically increasing; processes are not permitted to undo past events. $\textbf{\textit{R2}}_c$ is a version of ***R2***, created to overcome an issue with definedness of sequence difference [18], but semantically equivalent in the window of ***R1***. It states that a process must be history independent: the only part of the trace it may constrain is $tr' - tr$, that is, the portion since the previous observation tr. Specifically, if the history is deleted, by substituting ϵ for tr and $tr' - tr$ for tr', then the behaviour of the process is unchanged. Our formulation of $\textbf{\textit{R2}}_c$ deletes the history only when $tr \leq tr'$, which ensures that $\textbf{\textit{R2}}_c$ does not depend on ***R1***, and thus commutes with it. Intuitively, an ***R1***-$\textbf{\textit{R2}}_c$ healthy predicate syntactically does not constrain the trace history (tr), but only the trace contribution (***tt***), as the following theorem illustrates.

Theorem 2.2 (***R1***-$\textbf{\textit{R2}}_c$ trace contribution)**.**

$$\textbf{\textit{R1}}(\textbf{\textit{R2}}_c(P)) = (\exists\, tt \bullet P[\epsilon, tt/tr, tr'] \wedge tr' = tr \frown tt)$$

Finally, we have $\textbf{\textit{R3}}_h$, a version of ***R3*** taken from [5] that introduces the concept of intermediate observations, whilst ensuring that state variables are not included. $\textbf{\textit{R3}}_h$ states that if a process observes *wait* to be true, then its predecessor has not yet terminated and thus the current process should behave like the reactive identity, $\mathit{II}_{\textbf{\textit{R}}}$. For example, in a composition like $P\ ;\ Q$, if P is intermediate then Q, if $\textbf{\textit{R3}}_h$ healthy, will behave like $\mathit{II}_{\textbf{\textit{R}}}$.

The reactive identity simply maintains the present value of all variables, other than the state st when the predecessor is in an intermediate state, or behaves like ***R1***(***true***) if ok is false. The latter scenario means that our predecessor has diverged and thus we can guarantee nothing other than that the trace increases monotonically. Intuitively, a $\textbf{\textit{R3}}_h$ process conceals the state of any predecessor in an intermediate state. This allows that several independent state valuations are concurrently possible, yet concealed from one another, until an observation is made through an event interaction.

For comparison, we recall the definition of healthiness condition ***R3*** which was previously used in both CSP [29, 9] and *Circus* [40].

Definition 2.5 (***R3*** Healthiness Condition)**.**

$$\begin{aligned}
\textbf{\textit{R3}}(P) &\triangleq \mathit{II}_{\mathrm{rea}} \lhd \mathit{wait} \rhd P \\
\mathit{II}_{\mathrm{rea}} &\triangleq \mathit{II} \lhd \mathit{ok} \rhd \textbf{\textit{R1}}(\textbf{\textit{true}})
\end{aligned}$$

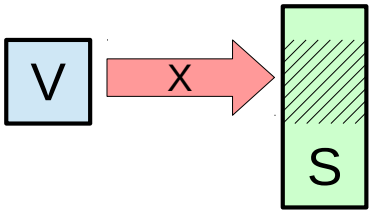


Figure 2: A typical lens

$$\mathit{get}\,(\mathit{put}\,s\,v) = v \qquad \text{(PutGet)}$$
$$\mathit{put}\,(\mathit{put}\,s\,v')\,v = \mathit{put}\,s\,v \qquad \text{(PutPut)}$$
$$\mathit{put}\,s\,(\mathit{get}\,s) = s \qquad \text{(GetPut)}$$

Figure 3: Lens laws

The only difference from $\mathbf{R3}_h$ is that the identity II_{rea} is used in intermediate states. This operator does not give special treatment to state variables: they are simply identified in intermediate states like other observational variables.

We compose the three constituents to yield $\mathbf{R}_s$, the overall healthiness condition of (stateful) reactive processes, which is idempotent and continuous.

Theorem 2.3 (Reactive Process Theory Properties)**.** $\mathbf{R}_s$ *is idempotent* $\mathbf{R}_s(\mathbf{R}_s(P)) = \mathbf{R}_s(P)$ *and continuous* $\mathbf{R}_s(\bigsqcap A) = \bigsqcap P \in A \bullet \mathbf{R}_s(P)$.

As for designs, a corollary of this theorem is that we obtain a complete lattice and the continuous theory properties of Theorem 2.1. Thus we now have a UTP theory of stateful reactive processes which will act a principle foundation for reactive design contracts.

2.4 Isabelle/UTP

Isabelle/UTP [21, 52, 20] is a mechanisation of the UTP semantic framework in the proof assistant Isabelle/HOL [39]. It allows us to define UTP theories within the alphabetised relational calculus, whilst taking advantages of Isabelle's type checker, and then mechanically prove associated theorems, such as algebraic laws. Such laws can then be applied to program verification tasks in Isabelle.

An alphabetised relation is essentially a set of possible observations that can be made of the model, such as the set of possible input and output mappings. Our model of alphabetised predicates, therefore, is $\alpha\ \mathit{upred} \triangleq (\alpha \Rightarrow \mathsf{bool})$, where α is a suitable type for modelling the alphabet, that corresponds to the state space. This means that we can easily implement the usual operators of boolean algebra and complete lattices by lifting the corresponding HOL notions on sets. Similarly, relational operators like composition $P\ ;\ Q$ can also be obtained by lifting the corresponding HOL functions. A relation with input alphabet α and output alphabet β has the type $(\alpha, \beta)\ \mathit{rel}$ in Isabelle/UTP, which is syntactic sugar for a predicate of type $(\alpha \times \beta)\ \mathit{upred}$.

Variables in the state space α are modelled abstractly using *lenses* [17, 16], which are perhaps best known in the functional programming world. A lens $V \Longrightarrow S$, for view type V and source type S, identifies V with a subregion of S. This is illustrated in Figure 2, where the hatched region denotes the portion of S that V corresponds to. Lenses can be used to abstract many types of data structure. For example, if S is a record type, then V might be a particular field, or if S is a function type, then V might be an element of the domain.

A lens consists of two functions: ***get*** that extracts a view from a larger source, and ***put*** that puts back an updated view. Moreover the behaviour of lenses is constrained by a number of algebraic laws which are summarised in Figure 3. Since lenses are semantic rather than syntactic entities, we cannot compare them just using (in)equality, and thus we introduce further operators. Lens equivalence, $X \approx Y$, states that lenses X and Y view precisely the same region of the source, though these views may have different types. Lens independence, $X \bowtie Y$, states that the two lens views are independent: manipulating the source type using X has no effect on the region identified by Y and vice-versa. Such operators can be used as the basis for comparison of variables.

We have mechanised a theory of lenses in Isabelle during this project, including an algebra that allows us to variously compose lenses in the style of separation algebra [6]. For example, the sum lens $X \oplus Y$ represents the lens that simultaneously views the regions characterised by both lenses X and Y. For more details please see our recent paper [21]. In this particular deliverable, lenses are of vital importance to modelling continuous variables.

We model variables as abstract views on program state spaces with a uniform semantic interface. A variable $x : \tau \Longrightarrow \alpha$ is a lens that views a particular subregion of type τ in α, which affords a very general state model. The main advantage lenses thus provide us with is an abstract notion of variables and state space in UTP predicates, such that a wide variety of different representations are possible. A commonly employed model for state spaces is that of Isabelle/HOL records, with fields to model variables, as these afford a large amount of built-in proof automation, which thus aids the program verification effort. Our use of lenses enables a more abstract characterisation, and ensures that any lens-based model for variables can be applied to proven laws of Isabelle/UTP, including for example partial function maps to encode a deeper predicate model with variable names as first-class citizens.

In order to support construction of program state space, each of which contains a number of variables, we have created the **alphabet** command in Isabelle/UTP. It creates a new record type, where each field corresponds to a variable in the state, and assigns a lens to each of them. It automatically generates well-formedness theorems for each of the lenses, and independence theorems for each pair. A template alphabet command is shown below:

$$\begin{aligned}&\textbf{alphabet}\ \textit{st-name} =\\ &\quad \textit{var}_1 :: \textit{type}_1\\ &\quad \textit{var}_2 :: \textit{type}_2\\ &\quad \ldots\\ &\quad \textit{var}_n :: \textit{type}_n\end{aligned}$$

This creates a state space type called *st-name* with n variables of a given type. Each of the variables is assigned a lens which can be referred to by name in a UTP predicate or relation. A program over this state space would have type $(\textit{st-name}, \textit{st-name})\ \textsf{rel}$, and would support relational constructions like $\textit{var}_2 := v$. The command automatically generates all possible independence theorems of the form $\textit{var}_i \bowtie \textit{var}_j$, where $i \neq j$.

Mechanisation of the predicate calculus requires that we can specify meta-logical provisos, such as $x \notin \mathsf{fv}(P)$, that is, that variable x is not free in P, and also variable substitution.

$$
\begin{aligned}
(\exists\, x \bullet P \wedge x = e) &= P[e/x] && \text{if } x \in \mathsf{mwb\text{-}lens}, x \sharp e \\
x := e \,;\, P &= P[e/x] \\
x := e \,;\, x := f &= x := f && \text{if } x \sharp f \\
x := e \,;\, y := f &= y := f \,;\, x := e && \text{if } x \bowtie y, x \sharp f, y \sharp e
\end{aligned}
$$

Table 1: Isabelle/UTP laws

Alphabetised predicates are principally semantic rather than syntactic entities, and so these notions cannot be specified using, for example, recursive functions that depend on an abstract syntax tree. Instead, we leverage our theory of lenses to define weaker semantic notions. For free variables, we introduce the concept of *unrestriction*, written $x \sharp P$ for some variable (lens) x. A predicate P is unrestricted by variable x if the valuation of P does not depend on x. For example, the predicate $y > 10$ is unrestricted by x, assuming $x \bowtie y$, since the value of x clearly has no bearing on the truth value of the predicate.

Substitution is also introduced semantically using the notation $\sigma \dagger P$, where $\sigma : \alpha \to \alpha$ is a homogeneous substitution function on the state space. Application of a substitution to a predicate updates all possible observations using the function. The most basic substitution is the identity id, which maps all variables to their present value. We can also write $\sigma(x \mapsto_s e)$, which updates a substitution such that x takes the value of expression e. We also introduce the short-hand $[x_1 \mapsto_s e_1, \cdots, x_n \mapsto_s e_n] = id(x_1 \mapsto_s e_1, \cdots, x_n \mapsto_s e_n)$. A substitution $P[e_1, \cdots, e_n/x_1, \cdots, x_n]$ of n expressions to corresponding variables is then expressed as $[x_1 \mapsto_s e_1, \cdots, x_n \mapsto_s e_n] \dagger P$. This model allows us to obtain the usual laws of substitution, such as $(P \wedge Q)[v/x] = (P[v/x] \wedge Q[v/x])$.

With a complete relational calculus and associated meta-logical operators defined we are able to mechanise all the usual laws of predicate calculus, relation algebra, Kleene algebra, and other typical laws of programming, such as those in Table 1. These show how we specify meta-logical provisos in Isabelle/UTP. For example the last law, commutativity of assignments, requires that x and y be different variables, specified using lens independence, that x is not free in f and that y is not free in e.

So far we have mechanised several hundred of such algebraic laws, which provide the foundation for automated reasoning about programs and models. These can be seen by viewing our Isabelle/UTP git repository[1]. Moreover, we have also created a number of proof tactics for predicate calculus (pred-tac) and relational calculus (rel-tac), which also greatly aid the proof effort. When these are combined with Isabelle's built-in automated proof facilities [4] like the auto deduction tactic, the sledgehammer automated theorem prover integration, and the nitpick counterexample generator, Isabelle/UTP greatly aids the effort of mechanising UTP theories and eventually applying them to verification.

[1] Please see https://github.com/isabelle-utp/utp-main/

3 Generalised Reactive Processes

In the section we construct a UTP theory which generalises the standard UTP theory of reactive processes [29, 9] with an abstract model of trace. In the UTP model, the trace can only be a discrete sequence of events, as used by process algebras like CSP. However, for hybrid computation we of course require continuously evolving variables, rather than simply discrete snapshots at given intervals. Therefore, we generalise the trace model such that constructions like piecewise continuous functions can also be supported. This thus provides an important first towards the hybrid relational calculus. Parts of this section have been submitted as a paper to *Information Processing Letters* [18].

3.1 Trace Algebra

In this section, we describe the trace algebra that underpins our generalised theory of reactive processes, the hybrid relational calculus, and ultimately the Modelica semantics. We define traces as an abstract set $\mathcal{T}$ equipped with two operators: trace concatenation $\frown : \mathcal{T} \to \mathcal{T} \to \mathcal{T}$, and the empty trace $\epsilon : \mathcal{T}$, which obey the following axioms.

Definition 3.1 (Trace algebra)**.** A trace algebra $(\mathcal{T}, \frown, \epsilon)$ is a cancellative monoid satisfying the following axioms:

$$x \frown (y \frown z) = (x \frown y) \frown z \tag{TA1}$$
$$\epsilon \frown x = x \frown \epsilon = x \tag{TA2}$$
$$x \frown y = x \frown z \;\Rightarrow\; y = z \tag{TA3}$$
$$x \frown z = y \frown z \;\Rightarrow\; x = y \tag{TA4}$$
$$x \frown y = \epsilon \;\Rightarrow\; x = \epsilon \tag{TA5}$$

As expected, $\frown$ is associative and has left and right units. Axioms TA3 and TA4 show that $\frown$ is injective in both arguments. As an aside, TA3 holds only in models without infinitely long traces, as such a trace x would usually annihilate y in $x \frown y$. Axiom TA5 states that there are no "negative traces", and so if x and y concatenate to ϵ then x is ϵ. We can also prove the reciprocal law: $x \frown y = \epsilon \;\Rightarrow\; y = \epsilon$. From this algebraic basis, we derive a prefix relation and subtraction operator.

Definition 3.2 (Trace prefix and subtraction)**.**

$$x \leq y \;\Leftrightarrow\; \exists\, z \bullet y = x \frown z$$

$$y - x \;\triangleq\; \begin{cases} \iota z \bullet y = x \frown z & \text{if } y \leq x \\ \epsilon & \text{otherwise} \end{cases}$$

Trace prefix, $x \leq y$, requires that there exists z that extends x to yield y. Trace subtraction $y - x$ obtains that trace z when $x \leq y$, using the definite description operator (Russell's ι), and otherwise yields the empty trace. This is slightly different from the standard UTP operator, which is defined only when $x \leq y$. We can prove the following laws about trace prefix.

Theorem 3.1 (Trace prefix laws)**.** *For* $x, y, z : \mathcal{T}$*:*

$$(\mathcal{T}, \leq) \text{ is a partial order} \tag{TP1}$$
$$\epsilon \leq x \tag{TP2}$$
$$x \leq x \frown y \tag{TP3}$$
$$x \frown y \leq x \frown z \Leftrightarrow y \leq z \tag{TP4}$$

TP2 tells us that ϵ is the smallest trace, TP3 that concatenation builds larger traces, and TP4 that concatenation is monotonic in its right argument. We also have the following trace subtraction laws.

Theorem 3.2 (Trace subtraction laws)**.**

$$x - \epsilon = x \tag{TS1}$$
$$\epsilon - x = \epsilon \tag{TS2}$$
$$x - x = \epsilon \tag{TS3}$$
$$(x \frown y) - x = y \tag{TS4}$$
$$(x - y) - z = x - (y \frown z) \tag{TS5}$$
$$(x \frown y) - (x \frown z) = y - z \tag{TS6}$$
$$y \leq x \wedge x - y = \epsilon \Leftrightarrow x = y \tag{TS7}$$
$$x \leq y \Rightarrow x \frown (y - x) = y \tag{TS8}$$

Laws TS1-TS3 relate trace subtraction and the empty trace. TS4 shows that subtraction inverts concatenation. TS5 shows that subtracting two traces is equivalent to subtracting their concatenation. TS6 shows that subtraction can be used to remove a common prefix. TS7 shows that two traces are equal if, and only if, the first is a prefix of the second and they subtract to ϵ. TS8 shows that a trace can be split into its prefix and suffix.

In the next section, we show that standard notions of traces are models. Afterwards, in Section 3 we use the algebra to create the generalised theory of reactive processes.

3.2 Trace Models

In this section we describe three trace models: positive reals, sequences, and timed traces. Other models are possible; for example, we can further extend timed traces to "super-dense time" [35] to encompass multiple distinguished discrete state updates at a time instant. We leave study of other models as future work.

Positive real numbers $\mathbb{R}_{\geq 0}$ form one of the simplest models of the trace algebra.

Theorem 3.3. $(\mathbb{R}_{\geq 0}, +, 0)$ *is a trace algebra.*

Proof. $+$ is clearly associative and has 0 as its left and right unit. Moreover, since $+$ is commutative and $\mathbb{R}_{\geq 0}$ contains no negative numbers then $x + y = x + z$ implies $y = z$. Finally, for the same reasons $+$ has no additive inverse. □

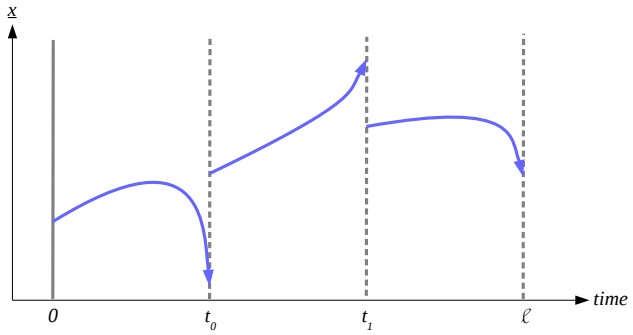


Figure 4: Piecewise continuous timed traces

Positive reals can be used to express timed programs with a clock variable $time : \mathbb{R}_{\geq 0}$ [25]. Sequences, unsuprisingly, also form a trace algebra, when we set $\frown$ to the usual sequence composition operator and ϵ to the empty sequence.

Theorem 3.4. $(\text{seq } Event, \frown, \langle\rangle)$ *is a trace algebra.*

Though simple, we note that the sequence-based trace model has been shown to be sufficient to characterise both untimed [40] and discrete time modelling languages [51].

A more complex model is that of piecewise continuous functions, for which we adopt and refine a model called *timed traces* ($\mathbb{TT}$) [24]. A timed trace is a partial function of type $\mathbb{R}_{\geq 0} \to \Sigma$, for continuous state type Σ, which represents the system's continuous evolution with respect to time.

In our model we also require that timed traces be piecewise continuous, to allow both continuous and discrete information. A timed trace is split into a finite sequence of continuous segments, as shown in Figure 4. Each segment accounts for a particular evolution of the state interspersed with discontinuous discrete events. This necessitates that we can describe limits and continuity, and consequently we require that Σ be a topological space, such as $\mathbb{R}^n$, though it can also contain discrete topological information, like events. Continuous variables are projections such as $x : \Sigma \to \mathbb{R}$. We give the formal model below.

Definition 3.3 (Timed Traces)**.**

$$\mathbb{TT} \triangleq \left\{ \begin{array}{l} f : \mathbb{R}_{\geq 0} \nrightarrow \Sigma \\ \mid \exists\, t \bullet \operatorname{dom}(f) = [0, t) \\ \qquad \land\ t > 0 \Rightarrow \exists\, I : \mathbb{R}_{\text{oseq}} \\ \qquad \bullet \left(\begin{array}{l} \operatorname{ran}(I) \subseteq [0, t] \\ \land\ \{0, t\} \subseteq \operatorname{ran}(I) \\ \land \left(\begin{array}{l} \forall\, n < \#I - 1 \bullet \\ \quad f \text{ cont-on } [I_n, I_{n+1}) \land \\ \quad \lim\limits_{t \to I_{n+1}^-} f(t) \text{ exists} \end{array} \right) \end{array} \right) \end{array} \right\}$$

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

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

A timed trace is a partial function f with domain $[0, t)$, for end point $t \geq 0$. When the

trace is non-empty ($t > 0$), there exists an ordered sequence of instants I giving the bounds of each segment. $\mathbb{R}_{\text{oseq}}$ is the subset of finite real sequences such that for every index n in the sequence less than its length $\#x$, $x_n < x_{n+1}$. I must naturally contain at least 0 and t, and only values between these two extremes. The timed trace f is required to be continuous on each interval $[I_n, I_{n+1})$. The operator f cont-on A denotes that f is continuous on the range given by A. We also require that each segment be convergent, so that the limit as f approaches I_{n+1} from the left exists.

We now introduce the core timed trace operators, which take inspiration from Höfner's algebraic trajectories [30].

Definition 3.4 (Timed Trace Operators)**.**

$$\begin{aligned} f \gg n &\triangleq \lambda\, x \bullet f(x - n) \\ \text{end}(f) &\triangleq \min(\mathbb{R}_{\geq 0} \setminus \text{dom}(f)) \\ \epsilon &\triangleq \emptyset \\ f \frown g &\triangleq f \cup (g \gg \text{end}(f)) \end{aligned}$$

Operator $f \gg n$ shifts the indices of a partial function $f : \mathbb{R}_{\geq 0} \nrightarrow A$ to the right by $n : \mathbb{R}$. The operator $\text{end}(f)$ gives the end time of a trace $f : \mathbb{TT}$ 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 when it goes beyond the end of f, and takes the union. We establish laws governing these trace operators.

Theorem 3.5. *Timed-trace laws*

$$(f \gg m) \gg n = f \gg (m + n) \tag{T1}$$
$$(f \cup g) \gg n = (f \gg n) \cup (g \gg n) \tag{T2}$$
$$\text{end}(\epsilon) = 0 \tag{T3}$$
$$\text{end}(x \frown y) = \text{end}(x) + \text{end}(y) \tag{T4}$$

T1 shows that shifting a function twice equates to a single shift on their summation. T2 shows that shift distributes through function union. T3 shows that the length of the empty trace is 0, and T4 shows that the length of a trace is the sum of its parts. $\mathbb{TT}$ is closed under trace concatenation.

Theorem 3.6 (Trace concatenation closure)**.** *$tt_1, tt_2 \in \mathbb{TT}$ if, and only if, $tt_1 \frown tt_2 \in \mathbb{TT}$*

This theorem tells us that decomposition of a timed trace always yields timed traces. Finally, trace concatenation satisfies our trace algebra.

Theorem 3.7. *$(\mathbb{TT}, \frown, \epsilon)$ forms a trace algebra*

Proof. For illustration, we show the derivation for associativity. The other proofs are

simpler.

$$
\begin{aligned}
& x \frown (y \frown z) \\
& = x \cup ((y \cup (z \gg \mathrm{end}(y)) \gg \mathrm{end}(x)) \\
& = x \cup ((y \gg \mathrm{end}(x)) \cup (z \gg \mathrm{end}(y) \gg \mathrm{end}(x))) \\
& = (x \cup (y \gg \mathrm{end}(x))) \cup (z \gg (\mathrm{end}(x) + \mathrm{end}(y))) \\
& = (x \frown y) \cup (z \gg (\mathrm{end}(x) + \mathrm{end}(y))) \\
& = (x \frown y) \cup (z \gg (\mathrm{end}(x \frown y))) \\
& = (x \frown y) \frown z
\end{aligned}
$$

□

This model provides the basis for hybrid computation. We introduce the theory in the next section.

3.3 Theory of Reactive Processes

Here, we use our trace algebra to provide a generalised theory of reactive processes. We prove the key laws of reactive processes, thus demonstrating the conservative nature of our theory. Many of the properties here have been previously proved [9], but we restate and prove many of them due to our weakening of the trace model and some small differences. Another novelty is that all these theorems have been mechanised in our Isabelle/UTP repository. Following [29, 9] we define the theory in terms of two pairs of observational variables:

- $wait, wait' : \mathbb{B}$ – describe when the previous or current process, respectively, is in an intermediate state;
- $tr, tr' : \mathcal{T}$ – the trace that occurred prior to and after execution of the current process in terms of a trace algebra $(\mathcal{T}, \frown, \epsilon)$.

Our theory does not contain refusal variables ref, ref', as these are not always necessary to describe reactive processes [51]. We describe three healthiness conditions namely ***R1***, $\boldsymbol{R2}_c$, and ***R3***. ***R1*** and ***R3*** are already presented in [29]; for their ***R2*** we have a different formulation, which we call $\boldsymbol{R2}_c$.

Definition 3.5. Reactive healthiness conditions

$$
\begin{aligned}
\boldsymbol{R1}(P) &\triangleq P \wedge tr \leq tr' \\
\boldsymbol{R2}_c(P) &\triangleq P[\epsilon, tr' - tr / tr, tr'] \lhd tr \leq tr' \rhd P \\
\boldsymbol{R3}(P) &\triangleq \varPi \lhd wait \rhd P \\
\boldsymbol{R} &\triangleq \boldsymbol{R3} \circ \boldsymbol{R2}_c \circ \boldsymbol{R1}
\end{aligned}
$$

R1 states that tr is monotonically increasing; processes are not permitted to undo past events. $\boldsymbol{R2}_c$ states that a process must be history independent: the only part of the trace it may constrain is $tr' - tr$, that it, the portion since the previous observation tr. Specifically, if the history is deleted, by substituting ϵ for tr and $tr' - tr$ for tr', then the behaviour of the process is unchanged. Our formulation of $\boldsymbol{R2}_c$ deletes the history

only when $tr \leq tr'$, which ensures that $\mathbf{R2}_c$ does not depend on **R1**, and thus commutes with it. Finally, **R3** states that if a prior process is intermediate ($wait'$) then the current process must identify all variables.

We compose the three to yield $\mathbf{R}$, the overall healthiness condition of reactive processes. An example $\mathbf{R}$ healthy predicate is $\mathbf{R3}(tr' = tr \frown \langle a \rangle \wedge v' = v)$, which extends the trace with a single event a and leaves program variable v unchanged. We show that $\mathbf{R}$ is idempotent and monotonic.

Theorem 3.8 ($\mathbf{R}$ idempotence and monotonicity)**.**

$$\mathbf{R} = \mathbf{R} \circ \mathbf{R} \quad \textit{and} \quad P \sqsubseteq Q \Rightarrow \mathbf{R}(P) \sqsubseteq \mathbf{R}(Q)$$

A corollary of Theorem 3.8 is that reactive processes form a complete lattice.

Theorem 3.9. *Reactive processes form a complete lattice with infimum* $\bigsqcap_R A$ *and supremum* $\bigsqcup_R A$*.*

This, in particular, provides us with specification and reasoning facilities about recursive reactive processes using the fixed-point operators.

Having stated the lattice theoretic properties of reactive processes, we move onto the relational operators. Intuitively, **R1** and $\mathbf{R2}_c$ together ensure that the reactive behaviour of a process contributes an extension $\mathbf{t}$ to the trace.

Theorem 3.10 ($\mathbf{R1}$-$\mathbf{R2}_c$ trace contribution)**.**

$$\mathbf{R1}(\mathbf{R2}_c(P)) = (\exists\, \mathbf{t} \bullet P[\epsilon, \mathbf{t}/tr, tr'] \wedge tr' = tr \frown \mathbf{t})$$

This shows that for any $\mathbf{R1}$-$\mathbf{R2}_c$ process there exists a trace extension $\mathbf{t}$ recording its behaviour, and that tr' is the prior history appended with this extension. Aside from illustrating **R1** and $\mathbf{R2}_c$, this allows us to restate a process containing tr and tr' to one with only the extension logical variable $\mathbf{t}$, which provides a more natural entry point for reasoning about the trace contribution of a process. In particular, we can prove a related law about sequential composition of reactive processes.

Theorem 3.11 ($\mathbf{R1}$-$\mathbf{R2}_c$ sequential)**.** *If P and Q are $\mathbf{R1}$-$\mathbf{R2}_c$ healthy, then*

$$\begin{aligned} P \,;\, Q = \exists\, \mathbf{t}_1, \mathbf{t}_2 \bullet ((&P[\epsilon, \mathbf{t}_1/tr, tr'] \,; \\ &Q[\epsilon, \mathbf{t}_2/tr, tr']) \wedge \\ &tr' = tr \frown \mathbf{t}_1 \frown \mathbf{t}_2) \end{aligned}$$

Proof. By Theorem 3.10 and relational calculus. □

This theorem shows that two sequentially composed processes have their own unique contribution to the trace without sharing or interference. When applied in the context of a timed trace, for example, it allows us to subdivide the trajectory into segments, which we can reason about separately. This theorem allows us to demonstrate closure of $\mathbf{R1}$-$\mathbf{R2}_c$ predicates under sequential composition.

Theorem 3.12 ($\mathbf{R1}$-$\mathbf{R2}_c$ sequential closure)**.** *If P and Q are both **R1** and $\mathbf{R2}_c$ healthy then*

$$\mathbf{R1}(\mathbf{R2}_c(P \,;\, Q)) = P \,;\, Q$$

Closure of ***R3*** has previously been shown [9] and we have mechanised this proof. This allows us to prove the following theorem.

Theorem 3.13 (***R*** sequential closure)**.** *If P and Q are both **R** healthy then $P \mathbin{;} Q$ is **R** healthy.*

We have now shown that reactive processes are closed under the lattice and relational operators, and can use these results to demonstrate the algebraic nature of the theory, by showing that reactive processes form a weak unital quantale.

Theorem 3.14. ***R*** *predicates form a weak unital quantale; provided $A \neq \emptyset$ the following laws hold:*

$$P \mathbin{;} (\bigsqcap_R A) = (\bigsqcap_R Q \in A \bullet P \mathbin{;} Q)$$
$$(\bigsqcap_R A) \mathbin{;} Q = (\bigsqcap_R P \in A \bullet P \mathbin{;} Q)$$
$$P \mathbin{;} \mathit{II} = \mathit{II} \mathbin{;} P = P$$

Proof. Since $\bigsqcap_R A = \mathbf{R}(\bigsqcap A)$ and sequential composition left and right distributes through $\bigsqcap$ it suffices, to show that ***R*** is continuous. □

Unital quantales are an important algebraic structure that give rise to Kleene algberas [1]. Significantly, they have a close connection with the point-free laws of sequential programming.

Our final result is closure under parallel composition. The UTP provides an operator called *parallel-by-merge* [29], $P \parallel_M Q$, whereby the composition of processes P and Q seperates their states, calculates their independent concurrent behaviours, and then merges the results. The operator is parametric over merge predicate M that specifies how synchronisation is performed. Different programming language semantics require formation of a bespoke merge predicate depending on their concurrency scheme. We give a slightly simplified version of the UTP definition, which is nevertheless equivalent.

Definition 3.6 (Parallel-by-merge)**.**

$$P \parallel_M Q \triangleq (\lceil P \rceil_0 \wedge \lceil Q \rceil_1 \wedge v' = v) \mathbin{;} M$$

Operator $\lceil P \rceil_n$ augments the after variables of P with an index; for example:

$$\lceil x' = 7 \cdot y \rceil_0 = (0.x' = 7 \cdot y)$$

The three conjuncts rename the after variables of P and Q to ensure no clashes, and identify all before variables v. Thus M has access to the state of each variable before execution (v), and from the respective composed processes ($0.v$ and $1.v$). M can thus invoke $tr' = f(0.tr, 1.tr)$ with a suitable trace merge function f, such as interleaving.

The healthiness conditions ***R1*** and ***R3*** can be directly applied to M, modulo some differences in alphabet. $\mathbf{R2}_c$ requires adaptation as it is possible to access the trace history through the two indexed traces, $0.tr$ and $1.tr$, in addition to tr. It is, therefore, necessary to delete the history from the two in the revised healthiness condition $\mathbf{R2}_m$ below.

Definition 3.7 (***R2***$_c$ for merge predicates)**.**

$$\textbf{\textit{R2}}_m(M) \triangleq (P[\epsilon, tr' - tr, 0.tr - tr, 1.tr - tr / tr, tr', 0.tr, 1.tr]) \lhd tr \leq tr' \rhd P$$

R2$_m$ has the same form as ***R2***$_c$ except that it deletes the history of three extant traces, tr', $0.tr$, and $1.tr$. From M's perspective, $0.tr$ and $1.tr$ contain the trace the parallel processes have executed. Thus we need to delete the history, through substitution, from these as well so that they contain only the contributions of their respective processes. This allows us to show that the overall composition is ***R2***$_c$. We define a condition for merge predicates – ***R***$_m \triangleq$ ***R1*** $\circ$ ***R2***$_m$ $\circ$ ***R3*** – and prove the following final theorem.

Theorem 3.15. *$P \parallel_M Q$ is **R** healthy provided that P, Q are **R** healthy, and M is **R**$_m$ healthy.*

Thus our generalised theory of reactive processes is conservative and unifies the denotational semantics of concurrent programming.

3.4 Reactive Relations and Conditions

In this section we introduce a form of reactive relation which we will use to describe assumptions and guarantees in our reactive contracts. A reactive relation is an ***R1***-***R2***$_c$ healthy predicate that does not have ok, ok', $wait$, and $wait'$ in its alphabet. Such a relation is effectively an alphabetised relation with the non-relational trace variable ***tt*** present.

We define the following healthiness condition for reactive relations:

Definition 3.8 (Reactive Relations)**.**

$$\textbf{\textit{RR}}(P) = (\exists\, ok, ok', wait, wait' \bullet \textbf{\textit{R1}}(\textbf{\textit{R2}}_c(P)))$$

Reactive relations are used in our contracts to represent the preconditions, periconditions, and postconditions, that is the constituent parts of the contract's assumptions and guarantees. Clearly, reactive relations form a complete lattice where the top remains ***false*** and the bottom is ***R1***$(true)$, that is the most non-determinstic relation where the trace is monotonically increasing.

Since reactive relations are a kind of condition, it is useful to have an associated Boolean algebra to support contract and specification construction. However, logical negation is not closed under ***R1*** and thus it is necessary to redefine negation, and also implication, for similar reasons, for reactive relations.

Definition 3.9 (Reactive Relation Logical Operators)**.**

$$\textbf{\textit{true}}_r \triangleq \textbf{\textit{R1}}(\textbf{\textit{true}}) \qquad \neg_r P \triangleq \textbf{\textit{R1}}(\neg P) \qquad P \Rightarrow_r Q \triangleq (\neg_r P \vee Q)$$

Reactive negation $\neg_r P$ negates and then applies ***R1***. Effectively this yields a predicate whose corresponding set of trace extensions do not satisfy P. Since ***RR*** is closed under the other Boolean operator we can prove the following theorem.

Theorem 3.16. $(\textbf{\textit{RR}}, \wedge, \vee, \neg_r, \textbf{\textit{true}}_r, \textbf{\textit{false}})$ *forms a Boolean algebra*

Sequential composition is closed under ***RR*** and $\varPi$, with an appropriate alphabet, is also a reactive relation. We define an assignment operator, in the style of Back's update action [2], that applies a substitution function $\sigma : \Sigma \to \Sigma$ to the state-space variable st, and leaves all other variables unchanged.

Definition 3.10 (Reactive Relational Assignment)**.**

$$\langle \sigma \rangle_r \triangleq (tr' = tr \wedge st' = \sigma(st) \wedge r' = r)$$

Since the alphabet is open, we use the shorthand r to refer to all other variables excluding ok, $wait$, tr, and st. We can write a singleton assignment using the derived syntax $x :=_r v \triangleq \langle \{x \mapsto v\} \rangle_r$. Assignment is ***RR***, since we conjoin with $tr' = tr$. As usual, we also introduce the degenerate form $\varPi_r \triangleq \langle id \rangle_r$ which simply retains the values of all variables. We also define a state condition operator $[s]_r \triangleq \textbf{\textit{R1}}(s)$, where s is a predicate over undashed state variables only. The state condition is ***RR*** healthy: it is clearly ***R1***, and also it is $\textbf{\textit{R2}}_c$ since s contains no reference to trace variables.

A useful subset of the reactive relations, is the reactive conditions, which we use to encode contractual preconditions. A relational condition is a relation b that does not refer to dashed variables; they can be characterised by the idempotent $\textbf{\textit{C}}(P) = P \ ; \textbf{\textit{true}}$. For reactive relations, we cannot exclude all dashed variables as we do wish to express trace constraints using ***tt***, which includes tr and tr'. Consequently, reactive conditions are characterised by the following healthiness condition, ***RC***.

Definition 3.11 (Reactive Conditions)**.**

$$\textbf{\textit{RC1}}(P) \triangleq \neg_r ((\neg_r P) \ ; \textbf{\textit{true}}_r) \qquad \textbf{\textit{RC}} \triangleq \textbf{\textit{RC1}} \circ \textbf{\textit{RR}}$$

We require that $\textbf{\textit{true}}_r$ is a right unit of the predicate's negated form, which means, firstly, that a healthy P can refer only to undashed state and observational variables other than tr. For example, any state condition $[s]_r$ is ***RC1***, as the following derivation shows.

$$\begin{aligned}
\textbf{\textit{RC1}}([s]_r) &= \neg_r ((\neg_r [s]_r) \ ; \textbf{\textit{true}}_r) && [\textbf{\textit{RC1}} \text{ definition}] \\
&= \neg_r ([\neg s]_r \ ; \textbf{\textit{true}}_r) && [\text{predicate calculus}] \\
&= \neg_r ((tr \leq tr' \wedge \neg s) \ ; tr \leq tr') && [\textbf{\textit{R1}}, \textbf{\textit{true}}_r, [-]_r \text{ definitions}] \\
&= \neg_r (tr \leq tr' \ ; tr \leq tr' \wedge (\neg s) \ ; tr \leq tr') && [\text{relational calculus}] \\
&= \neg_r (tr \leq tr' \wedge (\neg s) \ ; tr \leq tr') && [\text{transitivity of} \leq] \\
&= \neg_r (tr \leq tr' \wedge (\exists t_0 \bullet (\neg s) \wedge t_0 \leq tr')) && [; \text{ definition, substitution}] \\
&= \neg_r (tr \leq tr' \wedge \neg s) && [\text{predicate calculus}] \\
&= \neg_r (\neg_r [s]_r) && [\neg_r, \textbf{\textit{R1}} \text{ definitions}] \\
&= [s]_r && [\text{double negation}]
\end{aligned}$$

Reactive conditions can also refer to tr', but only provided that the corresponding trace extension ***tt*** refers only to a prefix of the trace. For example, $\neg_r (\langle a \rangle \leq \textbf{\textit{tt}})$ is ***RC*** healthy, because it only refers to a prefix of ***tt*** in its negated form, as the following derivation

confirms.

$$\begin{aligned}
\textbf{\textit{RC1}}(\neg_r(\langle a\rangle \leq \textbf{\textit{tt}})) &= \neg_r((\neg_r \neg_r(\langle a\rangle \leq \textbf{\textit{tt}}))\ ;\ \textbf{\textit{true}}_r) && [\textbf{\textit{RC1}}\text{ definition}]\\
&= \neg_r((\langle a\rangle \leq \textbf{\textit{tt}})\ ;\ \textbf{\textit{true}}_r) && [\text{double negation}]\\
&= \neg_r(tr \frown \langle a\rangle \leq tr'\ ;\ tr \leq tr') && [\textbf{\textit{tt}},\ \textbf{\textit{true}}_r\text{ definition}]\\
&= \neg_r(tr \frown \langle a\rangle \leq tr') && [\text{composition of } \leq]\\
&= \neg_r(\langle a\rangle \leq \textbf{\textit{tt}}) && [\textbf{\textit{tt}}\text{ definition}]
\end{aligned}$$

Effectively, reactive conditions serve to restrict permissible initial behaviours in the trace; the previous example states that the event a must not be performed initially.

RC1 is monotonic, and thus ***RC*** predicates form a complete lattice. In particular, we retain the lattice top and bottom elements of ***false*** and $\textbf{\textit{true}}_r$, and also the connectives $\wedge$ and $\vee$. However, ***RC*** predicates are not closed under reactive negation. This, however, is not necessary for the purposes of this paper.

We also define a reactive weakest precondition operator [15].

Definition 3.12 (Reactive Weakest Precondition)**.**

$$P\ \textbf{\textit{wp}}_r\ Q\ \triangleq \neg_r(P\ ;\ \neg_r\ Q)$$

This operator is similar to that given for relations in the UTP book and tutorials [29, 8]. We have simply replaced relational negation with reactive negation. Thus, we have proven a number of standard ***wp*** laws [15] in this context, which we enumerate below.

Theorem 3.17 (Reactive Weakest Precondition Laws)**.**

$$P\ \textbf{\textit{wp}}_r\ \textbf{\textit{true}}_r = \textbf{\textit{true}}_r \tag{3.17.1}$$

$$P\ \textbf{\textit{wp}}_r\ (Q \wedge R) = (P\ \textbf{\textit{wp}}_r\ Q \wedge P\ \textbf{\textit{wp}}_r\ R) \tag{3.17.2}$$

$$(P \lhd b \rhd_r Q)\ \textbf{\textit{wp}}_r\ R = (P\ \textbf{\textit{wp}}_r\ R \lhd b \rhd_r Q\ \textbf{\textit{wp}}_r\ R) \tag{3.17.3}$$

$$(P\ ;\ Q)\ \textbf{\textit{wp}}_r\ R = P\ \textbf{\textit{wp}}_r\ (Q\ \textbf{\textit{wp}}_r\ R) \tag{3.17.4}$$

$$\textit{II}_r\ \textbf{\textit{wp}}_r\ R = R \tag{3.17.5}$$

$$\langle\sigma\rangle_r\ \textbf{\textit{wp}}_r\ R = \sigma \dagger R \tag{3.17.6}$$

$$\textbf{\textit{false}}\ \textbf{\textit{wp}}_r\ P = \textbf{\textit{true}}_r \tag{3.17.7}$$

$$(P \sqcap Q)\ \textbf{\textit{wp}}_r\ R = (P\ \textbf{\textit{wp}}_r\ R) \wedge (Q\ \textbf{\textit{wp}}_r\ R) \tag{3.17.8}$$

$$\left(\prod i \in A \bullet P(i)\right)\ \textbf{\textit{wp}}_r\ R = (\forall\, i \in A \bullet P(i)\ \textbf{\textit{wp}}_r\ R) \tag{3.17.9}$$

The majority of these laws are identical to those given by Dijkstra [15]. Our assignment law uses a substitution operator $\sigma \dagger R$ to apply substitution function σ to predicate R.

Like UTP relations, reactive relations do not have the expressivity to account for non-termination. These are accounted for by our contracts.

4 Hybrid Relational Calculus

In this section we describe a substantial upgrade to our hybrid relational calculus, which was previously described in Deliverable D2.2c [10, 19]. The underlying semantic model, rather than a bespoke UTP theory, is now our theory of generalised reactive processes. Moreover, the signature has substantially changed, with additional operators provided, and supporting theorems.

4.1 Core Calculus

Hybrid relations are used to describe the assumptions and guarantees associated with hybrid reactive designs by constraining the possible evolutions of continuous variables. A hybrid relation is a form of reactive relation where the underlying trace model is $(\mathbb{TT}, \frown, \epsilon)$. Thus, the trace contribution ($\boldsymbol{tt}$) refers to a particular evolution of the continuous state space Σ_c, which is a topological (Hausdorff) space. We introduce the syntax $\ell \triangleq \mathrm{end}(\boldsymbol{tt})$ which refers to the length of the present evolution.

Variables in the timed trace model are projections of the continuous state space Σ, which is a topological space. Technically, we uses lenses [16, 21] to model these projects, such that each continuous variable x identifies a region of Σ_c, such that $x : \mathbb{R} \Longrightarrow \Sigma_c$. Actually, the source type of each lens is not limited to $\mathbb{R}$ but can also be any topological space. We introduce the syntax $s{:}x$ to project the part of state space s described by lens x. A continuous variable expression $\underline{x}(t)$ can then be defined as follows.

Definition 4.1 (Continuous Variable Expression)**.**

$$\underline{x}(t) \triangleq \boldsymbol{tt}(t){:}x$$

A continuous variable $\underline{x}$ is a function that obtains the continuous state space at time t and then projects the corresponding region.

For the sake of generality, we split the overall state space of a hybrid relation Σ, described by observational variable st, into both a discrete state space (Σ_d) and a continuous state space (Σ_c). We therefore introduce lenses $\boldsymbol{d} : \Sigma_d \Longrightarrow \Sigma$ and $\boldsymbol{c} : \Sigma_c \Longrightarrow \Sigma$ that refer to these sub-regions of the state space, respectively. As in our previous work [19], we unify continuous variable assignment and evolution such that $\boldsymbol{c}$ is tied to the evolution in the trace ($\boldsymbol{tt}$). Nevertheless to avoid confusion, it is important to distinguish continuous **state variables**, that is the valuation of the continuous variables at the beginning or end of a computation, from continuous **trajectory variables**, which are functions on the timed trace. These quantities are linked, but are not identical.

We also note that the discrete variables within $\boldsymbol{d}$ are not precisely the same concept as discrete variables in the Modelica sense. They are variables that are not represented in the trajectory and exist only as imperative assignable variables. For the most part such variables are useful to store temporary local variables used in imperative program fragments. In contrast, for Modelica, discrete variables are really a subclass of continuous variable that remain constant over a trajectory evolution, and we shall model them as such.

Next, we describe instant relations and the interval operator, which we have previously given a semantics in a bespoke theory of hybrid relations [19].

Definition 4.2 (Instant Relations and Intervals)**.**

$$P \mathbin{@} t \;\triangleq\; \{\boldsymbol{c}' \mapsto \boldsymbol{tt}(t)\} \dagger P$$

$$\lceil P(ti) \rceil \;\triangleq\; \boldsymbol{R1}\,(\forall\, t \in [0, \ell) \bullet P(t) \mathbin{@} t)$$

An instant predicate expression $P \mathbin{@} t$ lifts primed continuous state variables, referred to in P, to continuous trajectory variables. For example, the expression $(x' > 7.5) \mathbin{@} t$ is equivalent to $\underline{x}(t) > 7.5$, assuming that x is a continuous state variable. This is a slight departure from previous work, in that instant relations can now refer to the initial values of continuous variables, which was not previously possible. This is why we use the primed version of each state variable. 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 $\mathtt{tt}(t)$: the trajectory state at t.

An interval specification $\lceil P(ti) \rceil$ states that such a relation P holds over the entire evolution of the trajectory. Here, P is also parametrised by the current time ti, which allows continuous variables to also depend on time. Technically, $ti : \mathbb{R}_{\geq 0}$ is distinguished variable that is often used in continuous time predicates. The definition of $\lceil P(ti) \rceil$ states that P holds at every instant ti between 0 and ℓ, and additionally enforces $\boldsymbol{R1}$ to ensure only healthy timed traces are permitted. The construction is automatically $\boldsymbol{R2}_c$ since it only refers to $\boldsymbol{tt}$ and not tr or tr' explicitly. Thus, since neither ok nor $wait$ are mentioned, $\lceil P(ti) \rceil$ is an $\boldsymbol{RR}$ healthy reactive relation.

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 4.3 (Continuous Coupling Invariants)**.**

$$\boldsymbol{II}(x) \;\triangleq\; \boldsymbol{R1}\,(\boldsymbol{c}{:}x = \underline{x}(0))$$

$$\boldsymbol{rI}(x) \;\triangleq\; \boldsymbol{R1}\left(\boldsymbol{c}{:}x' = \lim_{t \to \ell^-} \underline{x}(t)\right)$$

The first coupling invariant, $\boldsymbol{II}(x)$, links together the initial value of continuous state variable x with the corresponding trajectory variable at time 0. The second, $\boldsymbol{rI}(x)$, 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 of timed traces, we know that the latter limit must exist, since our trajectories are always piecewise convergent.

The asymmetry of the two invariants is important. Whilst the trajectory explicitly defines a value at time 0, as invoked by $\boldsymbol{II}$, 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 sequential 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 $\boldsymbol{II}(x)$ and $\boldsymbol{rI}x$ are healthy reactive relations.

Using these operators, we can now define operators for continuous function evolution.

Definition 4.4 (Function Evolution)**.**

$$x \leftarrow f(ti) \triangleq (\lceil x' = f(ti) \rceil \wedge \ell > 0)$$

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

$$x \underset{[s,t]}{\leftarrow} f(ti) \triangleq (x \leftarrow f(ti) \wedge \ell \in [s,t] \wedge \boldsymbol{d}' = \boldsymbol{d} \wedge \boldsymbol{rl}(\boldsymbol{v}))$$

The first operator, $x \leftarrow f(ti)$, simply states that the trajectory variable x evolves according to continuous function f. 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 in Modelica. It is also worth noting that other continuous variables not mentioned such a statement are unconstrained and thus behaved nondeterministically. This is an important feature of the model as it will allow the use of nondeterminism to model concurrency of parallel hybrid processes.

The second operator, $x \leftarrow_{\leq t} f(ti)$, is the same but adds the requirement that the duration be at most t. The third and final operator, $x \leftarrow_{[s,t]} f(ti)$, states that the evolution terminates non-deterministically in the interval $s \leq ti \leq t$. 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 $\boldsymbol{rl}(\boldsymbol{v})$ to set the final state of all continuous variables. All the function evolution operators form healthy reactive relations.

In addition to these evolution operators, we also define a specialised version of evolution that is analogous to an assignment in the imperative world.

Definition 4.5 (Evolution by Assignment)**.**

$$\{\sigma(ti)\}_h \triangleq (\lceil \langle \sigma(ti) \rangle \rceil \wedge \ell > 0)$$

This operator illustrates the usefulness of having relational predicates in intervals. It takes a parametric variable assignment σ and requires that this assignment hold at every instant. The assignment can be used to encode how variables evolve with respect to their initial value and the current time. For example, we can have $\{[x \mapsto x + 2 * ti]\}_h$ which means that x takes the value $x_0 + 2 * ti$ at each instant ti. We next define the preemption operator.

Definition 4.6 (Preemption)**.**

$$P \triangle \langle b \,|\, c \rangle \triangleq (P \wedge \lceil b \rceil \wedge \ell > 0 \wedge \boldsymbol{rl}(\boldsymbol{v}) \wedge c \wedge \boldsymbol{d}' = \boldsymbol{d})$$

The preemption operator ($P \triangle \langle b \,|\, c \rangle$) is a little different to the one described in our previous hybrid relational calculus [19], and now takes three parameters rather than two. It states that P evolves for some non-zero duration, while ever condition b holds. At some undetermined point, c should become true finally, and at this point the operator can terminate, yielding final values for all variables using the right limit, and identifying all discrete variables.

Intuitively, the first condition, b, is similar to the state invariants present in hybrid automata [27]. Evolution of P can continue while b remains true, which is ensured by conjunction with the interval specification $\lceil b \rceil$. 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 non-determinism as to when P terminates, which is necessary when handling imprecision of measured continuous values. If $b = (\neg c)$ then 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***, the miraculous reactive relation. Clearly then, as is common with the standard UTP relational theory, we cannot account for non-terminating hybrid relations in this domain. This will be allowed in the theory of hybrid reactive designs in Section 6. Reactive relations are closed under preemption. We now demonstrate an important theorem regarding the termination of a function evolution.

Theorem 4.1 (Evolution Termination)**.** *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:

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

This theorem demonstrates the condition under which a function evolution under 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 non-deterministically 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.

We next describe how we give a semantics to ODEs.

4.2 Derivatives and Ordinary Differential Equations

The ability to express properties of derivatives of continuous variables is of course central to hybrid system modelling. In the hybrid relational calculus we introduce the notation x ***has-der*** $f(ti)$ which states that the derivative of continuous variable x is determined by expression f, which is parametrised over the relative time ti. 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 $x : \mathbb{R}_{\geq 0} \Rightarrow \mathbb{R}^n$, that give the value for each of the n continuous variables at a given instant. The ODE is defined by function $f : \mathbb{R}_{\geq 0} \times \mathbb{R}^n \Rightarrow \mathbb{R}^n$ that gives the derivative of each variable at time t, and depends on the present value of the variables, that is $x(t)$. A solution to an ODE is any function x that changes at the rate specified by f.

Naturally, when animating or theorem proving a system, a single solution is normally desired, and for this it is necessary to construct an initial value problem (IVP) that supplements the ODE with initial values for all continuous variables. Then the Picard-Lindelöf theorem [12] can be applied to show that, provided f is Lipschitz continuous, then a unique solution exists to the initial value problem [33]. Lipschitz continuity essentially limits the rate at which a continuous function can change.

We can now describe our operator for ODEs in the hybrid relational calculus.

Definition 4.7 (Ordinary Differential Equation)**.**

$$\langle\, x \bullet f \,\rangle \triangleq (\exists\, g, l \bullet \ell > 0 \wedge \ell = l \wedge \textbf{\textit{II}}(x) \wedge \lceil g \textbf{ \textit{has-ode-deriv} } f \textbf{ \textit{at} } (t < l) \wedge x' = g(t) \rceil)$$

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 semantics of this operator states that there exists a solution function g and duration l such that the duration is both non-zero and equal to l, the initial values of trajectory variables are taken from the state variables, and the ODE holds over the duration. The latter is specified in terms of the interval operator, and a quaternary operator - ***has-ode-deriv*** - ***at*** $(- < -)$, that formally states that the derivative of g is f at every instant $t \in [0, l)$. Finally, the value of lens x is linked at every instant to the solution function g. Every operator of which the ODE operator is composed is ***R1*** and $\textbf{\textit{R2}}_c$, and thus it is a healthy reactive relation.

We provide a simple example below to illustrate the operator's use.

Example 4.1 (Gravity ODE)**.**

$$\langle\, h, v \bullet (\lambda(h, v) \bullet (v, -9.81)) \,\rangle$$

Here, we are describing two continuous variables: height h and velocity v. The derivatives are described by the function on the right, so that the derivative of h is v and the derivative of v is -9.81, Earth's gravitational acceleration. It should be noted we have been verbose here, but h and v have different meanings on the left and right of the $\bullet$. On the left they are global continuous variables, whilst on the right they are local to the function definition.

An ODE is equivalent to a derivative constraint, as the following theorem demonstrates.

Theorem 4.2 (ODE Derivative Constraint)**.**

$$\langle\, x \bullet f(ti) \,\rangle = (\textbf{\textit{II}}(x) \wedge x \textbf{ \textit{has-der} } (f(ti)(x)))$$

This theorem gives rise to alternative slightly simpler definition of the ODE operator. The definition applies the initial value coupling invariant, and asserts that lens x has the

derivative given by the characteristic ODE function f. Note that x is not necessarily a single variable, but as above it can be a vector of variables.

Sometimes it is useful to state that variables not defined by an ODE should be held constant during evolution. Ordinarily such variables will be permitted to behave arbitrarily which may be undesirable. We therefore have the following derived operator for framed ODEs.

Definition 4.8 (Framed ODE)**.**

$$\langle\, x : f \,\rangle \triangleq (\langle\, x \bullet f \,\rangle \wedge \lceil x{:}[\textbf{\textit{true}}] \rceil)$$

We utilise the framing operator from the UTP relational calculus, $x : [P]$ which states that relation P makes changes to only variables within x; all others remain unchanged. The behaviour of $\lceil x{:}[\textbf{\textit{true}}] \rceil$ is then the set of evolutions in which x can change according to the ODE, but every other variable is held constant over the evolution. This is useful for modelling Modelica's discrete variables which are held constant in this way.

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

Theorem 4.3 (ODE Solution)**.** *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 $\langle\, x \bullet f \,\rangle = x \leftarrow g(x, t)$.*

This theorem allows us to transform a differential equation into a solution function evolution. It has some subtleties that require further explanation. Function $g : \mathbb{R}^n \times \mathbb{R} \Rightarrow \mathbb{R}^n$ 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 $x \leftarrow g(x, t)$. 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 decided when an initial value is supplied by a preceding assignment.

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.

4.3 Perturbation

Dynamical models in languages like Modelica, and also their realisation in the physical world seldom follow the precise continuous mathematics that we have hitherto considered. Rather, the observable state of the real system at a given point in time ought be be within some error bound of the the idealised system, in order to account for necessary imprecision in detecting events. For this reason, we introduce perturbation operators into the hybrid relational calculus that allows us to weaken a predicate or hybrid relation according some

error bound $\epsilon : \mathbb{R}_{\geq 0}$. We first define an operator that weakens a predicate in continuous variables.

Definition 4.9 (Nearly Operator)**.**

$$\textbf{\textit{nrly}}(p, x, \epsilon) \triangleq \begin{cases} (\exists\, v \bullet p[v/x] \wedge |x - v| < \epsilon) & \text{if } \epsilon > 0 \\ p & \text{otherwise} \end{cases}$$

The ***nrly*** operator take a predicate on undashed variables p, a lens projecting a real vector from the continuous state space $x : \mathbb{R}^n \to \Sigma_c$, and the pertubation $\epsilon : \mathbb{R}_{\geq 0}$. It evaluates to true when x comes within ϵ of the truth boundary defined by p. More precisely, it expands the scope of p by an open ball of radius ϵ. For example, $\textbf{\textit{nrly}}(x \geq 0, x, 1)$ is equal to $x > -1$. It is a weakening operator, as the following theorem shows.

Theorem 4.4 (Nearly Weakens)**.** $p \Rightarrow \textbf{\textit{nrly}}(p, x, \epsilon)$

We also have a second operator, defined below, which similarly acts on the trajectory of a hybrid relation. It states that at every instant a given portion of the continuous state lies within ϵ of the ideal continuous state.

Definition 4.10 (Perturbation Operator)**.**

$$\textbf{\textit{ptrb}}(P, x, \epsilon) \triangleq \begin{cases} \exists f : \mathbb{TT} \bullet \left(\begin{array}{c} P[f/\textbf{\textit{tt}}] \wedge tr \leq tr' \wedge \ell = \text{end}(f) \wedge \\ \left(\begin{array}{c} \forall\, t \in [0, \text{end}(f)) \bullet \exists\, v : \mathbb{R}^n \bullet \\ |(f(t){:}x) - v| \leq \epsilon \wedge \\ \textbf{\textit{tt}}(t) = f(t)(x \mapsto v) \end{array} \right) \end{array} \right) & \text{if } \epsilon > 0 \\ P & \text{otherwise} \end{cases}$$

Function $\textbf{\textit{ptrb}}(P, x, \epsilon)$ takes an existing hybrid relation P, a lens projecting a real vector from the continuous state space $x : \mathbb{R}^n \to \Sigma_c$, and the pertubation $\epsilon : \mathbb{R}_{\geq 0}$. The definition states that there exists a timed trace f which is equated (by substitution) with the ideal timed trace of P. From f, a new perurbed trace is then constructed. This new trace must be well formed ($tr \leq tr'$) and it must have the duration of the original trace ($\text{end}(f)$). Moreover, for every instant t within this duration, there must exist a real vector v which is the perturbed state observation. The distance between v and the original observation $f(t){:}x$ must be no more than ϵ. Then, the new perturbed trace is set to have the original trace state at t with x updated to be v. This means that portions of state outside of x will remain unchanged by the perturbation.

In order to illustrate the perturbation operator, consider the simple evolution $P \triangleq \lceil x = 0 \rceil$ for $x : \mathbb{R} \Longrightarrow \Sigma_c$, which states that continuous variables x is 0 at every instant. If we perturb this hybrid relation with $\textbf{\textit{ptrb}}(P, x, 1)$, for instance, then the result is $\lceil -1 \leq x \wedge x \leq 1 \rceil$. In other words, the perturbation adds non-determinism to the previously deterministic evolution, and simply requires that the timed trace has x between -1 and 1 at every instant.

We can prove a number of characteristic theorems of $\textbf{\textit{ptrb}}(P, x, \epsilon)$. First of all, if P is a hybrid relation then so is $\textbf{\textit{ptrb}}(P, x, \epsilon)$. Secondly, application of a 0 perturbation yields the same hybrid relation, as shown below.

Theorem 4.5 (Zero Perturbation)**.** $\textbf{\textit{ptrb}}(P, x, 0) = P$

Secondly, we can show that pertubation is inherently a weaking operator, as shown below.

Theorem 4.6 (Perturbation Weakening)**.** *If P is a hybrid relation, and x is a very well-behaved lens, then* $\textbf{\textit{ptrb}}(P, x, \epsilon) \sqsubseteq P$.

This shows that the greater the perturbation applied, the more non-deterministic the hybrid relation will become. Thus, we can model a dynamical system using an idealised hybrid relation, and then apply such perturbations to weaken the specification, taking care of course to chose an appropriately small ϵ. Then a given implementation must satisfy this weakened hybrid relation.

4.4 Example

In order to exemplify the use of the hybrid relational calculus, we describe below part of the Modelica train model from the WP1 Railways case study as described in Deliverable D1.2b [41]. This example has also previously been reported in a conference paper on this case study [53].

We focus on the situation when the train is stopping due to an approaching red signal. We formalise this situation using continuous variables for train acceleration *acc*, velocity *vel* and position on the track *pos*. We note that *normal-deceleration* below is negative and determines the rate at which the train reduces its speed as a result of braking forces being applied.

Definition 4.11 (Braking Train in Hybrid Relational Calculus)**.**

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

We first assign initial values to the continuous variables, and this effectively creates initial conditions for the ODE. We then evolve the continuous variables, according to the ODE, 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 $\mathit{vel} > 0$, and the preemption condition is $\mathit{vel} \leq 0$. After this, we set the acceleration to 0, so that the train halts and does not start moving backwards.

We have also encoded this example in Isabelle/UTP and mechanised a proof (see Figure 5) that the train stops before the track ends, that is,

$$(\mathit{accl}' = 0 \wedge \lceil \mathit{pos} < 44 \rceil) \sqsubseteq \mathit{BrakingTrain}$$

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

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

Figure 5: The braking train scenario encoded in Isabelle/UTP.

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 *symbolically* to obtain a function evolution statement. This requires us to show Lipschitz continuity of the ODE so that, via the Picard Lindelöf theorem, there is precisely one such solution;
2. Use the assigned values to obtain the set of initial conditions;
3. Calculate the precise time at which the velocity reaches zero; here, that is approximately after 2.97 seconds;
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:

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

which includes the solution for the position derivative. In Isabelle, this can be done using the lesser-known *approximate* tactic [31], which safely employs a floating-point approximation to prove the conjecture with respect to the reals.

5 Reactive Design Contracts

In this section we give an overview of the signature of our theory of reactive design contracts and proven algebraic laws, leaving the definition of the UTP theory's healthiness condition (***NSRD***) for the interested reader in Appendix A. All the laws we present are mechanically proven theorems of our UTP theory, though we provide some intuition for why the laws holds. We illustrate the use of our contract notation with a number of *Circus*-based examples which give intuition, though stateful failure-divergences not the only applicable semantic model. Reactive programs proceed through three phases:

1. **pre-execution** – the program waits for its predecessor to terminate and does not contribute any observable behaviour.
2. **intermediate execution** – the program begins the main body of its execution, which includes communication with other concurrent processes and updates to its state. During this time the state is hidden from its successor.
3. **termination** – the program ceases interaction with the environment, reveals its final state to the successor, and signals permission for it to begin.

In our model, we largely assume that parallel programs do not directly share state but, as is typical in process algebras, they must explicitly communicate using a suitable mechanism such as channels. All other activity, such as state updates, is internalised to the sequential behaviour of the process, though it is possible to merge the state of several terminated parallel processes [40]. Shared variables can, nevertheless, be modelled by encoding them within the trace.

Reactive programs can also diverge, meaning they exhibit erroneous behaviour such as engaging in an infinite sequence of internal activity without any communication. Divergence in particular corresponds to violation of a contract's assumptions.

A reactive design contract is a triple [7] of the form

$$[\, P(st, \mathbf{tt}, r) \vdash Q(st, \mathbf{tt}, r, r') \mid R(st, st', \mathbf{tt}, r, r') \,]$$

The three parts of such a contract are:

1. The **precondition** P, with assumptions the contract makes before it executes, violation of which corresponds to a programmer error such as divergence. It is a reactive condition, and can therefore refer to the initial state st, the trace contribution $\mathbf{tt}$, and potentially other (unprimed) observational variables in the alphabet (r), but crucially not observational variables ok or $wait$, or primed variables other then tr'. Access to tr' is usually indirect through $\mathbf{tt}$.
2. The **pericondition** Q, with commitments the contract guarantees to fulfil during its intermediate states. It is a reactive relation on the initial state only, $\mathbf{tt}$, and any other variables (r, r').
3. The **postcondition** R, with commitments that will be fulfilled should the program terminate. It is a reactive relation that can additionally refer to the final state st', unlike the pre and pericondition.

Such contracts can be used both as a specification mechanism, for encoding assumptions and guarantees for a subsystem, or alternatively as a means to encode the semantics of a reactive programming language. A reactive design contract has the following definition.

Definition 5.1 (Reactive Design Contract)**.**

$$[\, P \vdash Q_1 \mid Q_2 \,] \triangleq \mathbf{R}_s(P \vdash Q_1 \diamond Q_2)$$

This definition assumes that P, Q_1, and Q_2 are formed as specified above. This is a form of UTP design which has been made reactive using $\mathbf{R}_s$. In previous work [40], reactive

designs would often be written in just two parts ($\mathbf{R}_s(P \vdash Q)$), the assumption and guarantee, with the intermediate and final behaviours intertwined. Here, we adopt the triple notation first developed in [7] as it allows us to consider these separately and simplifies many laws. The diamond $Q_1 \diamond Q_2$ is simply an abbreviation for $Q_1 \lhd wait' \rhd Q_2$ that distinguishes intermediate and final non-divergent observations.

We illustrate the use of contracts with the example of *Circus* event prefix as a reactive design triple, for which we need to specialise the semantic model to failure-divergences by adding the observational $ref' : \mathbb{P}\ Event$, as usual [9].

Example 5.1 (Event Prefix Reactive Design)**.**

$$a \rightarrow \mathbf{Skip} \triangleq [\, \mathbf{true}_r \vdash a \notin ref' \land \mathbf{tt} = \langle\rangle \mid st' = st \land \mathbf{tt} = \langle a \rangle \,]$$

A terminated prefix has a true precondition since it can never diverge; every context is a valid context. Its pericondition states that event a is not refused and the trace does not change in intermediate states. The postcondition states that the state is unchanged by the event, and the trace is extended with a. In *Circus* one can use this definition to represent the more general prefix construct using sequential composition: $a \rightarrow P \triangleq (a \rightarrow \mathbf{Skip})\ ;\ P$.

Another example is the **Skip** action, which represents a termination, and the **Stop** action, which represents a deadlock.

Example 5.2 (Terminated and Deadlocked Actions)**.**

$$\mathbf{Skip} \triangleq [\, \mathbf{true}_r \vdash \mathbf{false} \mid \mathbf{tt} = \langle\rangle \land st' = st \,]$$
$$\mathbf{Stop} \triangleq [\, \mathbf{true}_r \vdash \mathbf{tt} = \langle\rangle \mid \mathbf{false} \,]$$

The terminated process **Skip** has a true precondition. It has no intermediate observations, as it is essentially instantaneous. In the postcondition it makes no contribution to the trace, and leaves the state variables unchanged. The deadlocked process likewise has a true precondition. No state is a final state, indicated by the false postcondition, since the process does not terminate. In the intermediate states it is simply required that the trace is unchanged.

Example 5.3 (External Choice)**.**

$$\Box\, i \in A \bullet [\, P_1(i) \vdash P_2(i) \mid P_3(i) \,] =$$
$$\left[\bigwedge_{i \in A} P_1(i) \;\middle|\; \left(\bigwedge_{i \in A} P_2(i) \right) \lhd \mathbf{tt} = \langle\rangle \rhd \left(\bigvee_{i \in A} P_2(i) \right) \;\middle|\; \bigvee_{i \in A} P_3(i) \right]$$

Our final example is external choice over a contract indexed by set A. This permits internal activity in the choice branches, but the choice itself is not resolved until an external event occurs. The precondition requires that all branches of the external choice hold in the initial state. In the pericondition, while the trace has not changed and thus no event has occurred – $\mathbf{tt} = \langle\rangle$ – all periconditions of the choice hold simultaneously. Once an event has occurred only one of the periconditions need hold. This is the reason why the pericondition does not refer to final states, as these are concealed until termination

or observation. Finally, in the postcondition, one of the choice branch postcondition holds.

Though these previous three definitions look different to the standard presentation in ***Circus***, they are largely equivalent. The exception is event prefix, which is slightly different to the definition in [40] in that we conceal the state whilst waiting for the event.

Though language-specific operators like those presented above for ***Circus*** can be expressed, it turns out that many core contract operators can be introduced generically, as shown below.

Theorem 5.1 (Reactive Design Core Operators)**.**

$$
\begin{aligned}
\mathit{II}_{\mathbf{R}} &= [\,\mathbf{true}_r \vdash \mathbf{false} \mid \mathit{II}_r\,] \\
\langle\sigma\rangle_{\mathbf{R}} &= [\,\mathbf{true}_r \vdash \mathbf{false} \mid \langle\sigma\rangle_r\,] \\
\mathbf{Miracle} &= [\,\mathbf{true}_r \vdash \mathbf{false} \mid \mathbf{false}\,] \\
\mathbf{Chaos} &= [\,\mathbf{false} \vdash \mathbf{false} \mid \mathbf{false}\,]
\end{aligned}
$$

Operator $\mathit{II}_{\mathbf{R}}$ is reactive skip. It has a true precondition, and a false pericondition indicating that it has no intermediate states and so is essentially instantaneous. The postcondition defines that it contributes nothing to the trace, and simply identifies the before and after states. Since the alphabet at this point is open, we also add the conjunct $r' = r$ which is shorthand for saying all additional variables are identified. This distinguishes $\mathit{II}_{\mathbf{R}}$ from the ***Circus*** specific ***Skip*** operator from Example 5.2 which leaves ref' unconstrained.

$\langle\sigma\rangle_{\mathbf{R}}$ is a generalised assignment operator, again similar to Back's update action [2], where $\sigma : \Sigma \to \Sigma$ is a function on the state space. Its postcondition defines an update of the state by applying σ to it using the reactive relational assignment. The more specific assignment $x :=_{\mathbf{R}} v$ can be expressed as $\langle\{x \mapsto v\}\rangle_{\mathbf{R}}$.

Miracle is the miraculous reactive design. It has a true precondition, but has no intermediate or final states, and thus is effectively impossible to execute. As we shall see it corresponds to the top element of the refinement lattice, as it is the most deterministic reactive contract in that no behaviour implements it.

Chaos, in contrast, is the contract with an unsatisfiable precondition and thus always yields a program error. It corresponds to the bottom of the refinement lattice, and is the least deterministic contract. It can be used to identify states that are erroneous, and thus the context should avoid as illustrated by the following example.

Example 5.4 (Divergent Process)**.**

$$
\begin{aligned}
&a \to \mathbf{Chaos} \,\Box\, b \to \mathbf{Skip} = \\
&\quad [\,\neg_r\,(\langle a\rangle \leq \mathbf{tt}) \vdash \mathbf{tt} = \langle\rangle \wedge a \notin ref' \wedge b \notin ref' \mid \mathbf{tt} = \langle b\rangle \wedge st' = st\,]
\end{aligned}
$$

This ***Circus*** action allows either an a or b event, but if the environment chooses a then it diverges. The precondition therefore defines the assumption that the environment does not extend the trace by a, using a reactive condition. If it does, then the behaviour is unpredictable. The pericondition states that the trace has not yet been extended, and

the action does not refuse a or a b. However, though it is not refused, a can never lead to a terminating state as defined in the postcondition, which specifies that the trace is extended by b and leaves the state unchanged.

Contracts can also be constructed using the programming and specification operators of the UTP's relational calculus. This effectively means that relational laws of programming can be directly imported for use in proofs about contracts. We have proved a number of theorems that show the results of composing contracts.

Theorem 5.2 (Reactive Design Compositions)**.**

$$[\, P_1 \vdash P_2 \mid P_3 \,] \sqcap [\, Q_1 \vdash Q_2 \mid Q_3 \,] = [\, P_1 \wedge Q_1 \vdash P_2 \vee Q_2 \mid P_3 \vee Q_3 \,] \tag{5.2.1}$$

$$\prod_{i \in I} [\, P_1(i) \vdash P_2(i) \mid P_3(i) \,] = \left[\bigwedge_{i \in I} P_1(i) \;\middle|\; \bigvee_{i \in I} P_2(i) \;\middle|\; \bigvee_{i \in I} P_3(i) \right] \tag{5.2.2}$$

$$[\, P_1 \vdash P_2 \mid P_3 \,] \sqcup [\, Q_1 \vdash Q_2 \mid Q_3 \,] = \left[\begin{array}{c|c|c} P_1 \\ \vee\ Q_1 \end{array} \right. \left| \begin{array}{c} P_1 \Rightarrow_r P_2 \wedge \\ Q_1 \Rightarrow_r Q_2 \end{array} \right| \left. \begin{array}{c} P_1 \Rightarrow_r P_3 \wedge \\ Q_1 \Rightarrow_r Q_3 \end{array} \right] \tag{5.2.3}$$

$$[\, P_1 \vdash P_2 \mid P_3 \,] \lhd b \rhd [\, Q_1 \vdash Q_2 \mid Q_3 \,] = \left[\begin{array}{c|c|c} P_1 \\ \lhd b \rhd \\ Q_1 \end{array} \right. \left| \begin{array}{c} P_2 \\ \lhd b \rhd \\ Q_2 \end{array} \right| \left. \begin{array}{c} P_3 \\ \lhd b \rhd \\ Q_3 \end{array} \right] \tag{5.2.4}$$

$$[\, P_1 \vdash P_2 \mid P_3 \,] \,;\, [\, Q_1 \vdash Q_2 \mid Q_3 \,] = \left[\begin{array}{c|c|c} P_1 \wedge \\ P_3 \ \mathbf{wp}_r\ Q_1 \end{array} \right. \left| \begin{array}{c} P_2 \vee \\ P_3 \,;\, Q_2 \end{array} \right| \left. P_3 \,;\, Q_3 \right] \tag{5.2.5}$$

$$[\, P \vdash Q \mid R \,]^{n+1} = \left[\bigwedge_{i \leq n} (R^i \ \mathbf{wp}_r\ P) \;\middle|\; \bigvee_{i \leq n} R^i \,;\, Q \;\middle|\; R^{n+1} \right] \tag{5.2.6}$$

The internal choice of two contracts, $(P \sqcap Q)$, yields a contract that assumes both preconditions hold, and yields the combined intermediate and final states by disjunction. The preconditions are conjoined since the choice is nondetereministic, and thus there must be no possibility of divergence in all possible branches. Internal choice can, alternatively, be viewed as a disjunction operator for contracts similar to that in [3]. Similarly, an internal choice over a set of basic actions indexed by a set I conjoins all the preconditions, and disjoins the peri and postconditions. Dual to disjunction, the conjunction of two contracts $(P \sqcup Q)$ requires that one of the preconditions holds, and takes the conjunction of the corresponding intermediate and final states. The conditional $P \lhd b \rhd Q$, where b is a predicate on st alone, can be distributed through the pre, peri, and postconditions of the respective reactive designs.

Sequential composition $P \,;\, Q$, where $P = [\, P_1 \vdash P_2 \mid P_3 \,]$ and $Q = \sqcap [\, Q_1 \vdash Q_2 \mid Q_3 \,]$, is a little more involved. The combined precondition conjoins the precondition of P with a predicate requiring that the postcondition of P does not violate the precondition of Q. The latter is specified using the reactive weakest precondition operator, $P\ \mathbf{wp}_r\ Q$. The pericondition states that either P is in an intermediate state, and thus P_2 holds, or else Q is in intermediate state, P having terminated, and thus $P_3 \,;\, Q_2$ holds. Finally the postcondition states that both P and Q have terminated, that is, $P_3 \,;\, Q_3$.

We additionally show the law for finite iteration of a reactive design, P^{n+1}, assuming at least one execution, that is for $[\,P \vdash Q \mid R\,]\ ;\ [\,P \vdash Q \mid R\,]\ ;\ \cdots\ ;\ [\,P \vdash Q \mid R\,]$. This law can be later applied to calculate the contract for a recursive reactive program. The precondition requires that after $i \leq n$ iterations of the postcondition R, the precondition P is not violated. The pericondition states that postcondition R has been established a number of times $i \leq n$, following by the pericondition Q holding. In other words, one of the iterations is still in an intermediate state. Finally, the overall postcondition states that R has been established $n+1$ times.

The final law can be used to compute the contract of a tail recursive program of the form $\mu_{\mathbf{R}}\, X \bullet P\ ;\ X$, where $\mu_{\mathbf{R}}$ is the weakest fixed-point operator, which allows us to tackle iterative contract. This is subject to P being a productive [13] contract, that is, one that extends the trace when it terminates. For example, $a \rightarrow \textbf{\textit{Skip}}$ is productive because it always produces an a event upon termination. On the other hand, $\langle\sigma\rangle_{\mathbf{R}}$ is not productive because it contributes no events to the trace. Productivity is related, but not the same as the common notion of "guardedness" [29], as we shall see in Section A.4. If a contract's postcondition is productive, then we have the following theorem.

Theorem 5.3 (Recursive Reactive Design)**.** *If R is productive, that is, $R \wedge \epsilon < \textbf{\textit{tt}} = R$, then*

$$\mu_{\mathbf{R}}\, X \bullet [\,P \vdash Q \mid R\,]\ ;\ X = \left[\, \bigwedge_{i \in \mathbb{N}} \left(R^i \ \textbf{\textit{wp}}_r\ P\right) \;\middle|\; \vdash \bigvee_{i \in \mathbb{N}} R^i\ ;\ Q \;\middle|\; \textbf{\textit{false}} \,\right]$$

Such a recursive contract has a false postcondition, since it does not terminate. The precondition requires that, no matter how many times postcondition R is established, it does not violate the contract's precondition P. The pericondition is where the main behaviour of the contract is specified. It states that the postcondition is executed some number of times, and then the pericondition holds. In other words, the contract has executed its body and terminated into a final state of the body several times, but then finally the contract always lands in an intermediate state, since it does not terminate itself.

We now prove some of the algebraic laws of reactive design contracts.

Theorem 5.4 (Reactive Design Laws)**.**

$$\begin{aligned}
\textbf{\textit{Miracle}} \sqcap [\,P_1 \vdash P_2 \mid P_3\,] &= [\,P_1 \vdash P_2 \mid P_3\,] && \text{(RD1)}\\
\textbf{\textit{Chaos}} \sqcap [\,P_1 \vdash P_2 \mid P_3\,] &= \textbf{\textit{Chaos}} && \text{(RD2)}\\
\mathit{II}_{\mathbf{R}}\ ;\ [\,P_1 \vdash P_2 \mid P_3\,] &= [\,P_1 \vdash P_2 \mid P_3\,] && \text{(RD3)}\\
[\,P_1 \vdash P_2 \mid P_3\,]\ ;\ \mathit{II}_{\mathbf{R}} &= [\,P_1 \vdash P_2 \mid P_3\,] && \text{(RD4)}\\
[\,P_1 \vdash P_2 \mid \textbf{\textit{false}}\,]\ ;\ [\,Q_1 \vdash Q_2 \mid Q_3\,] &= [\,P_1 \vdash P_2 \mid \textbf{\textit{false}}\,] && \text{(RD5)}\\
\textbf{\textit{Miracle}}\ ;\ [\,P_1 \vdash P_2 \mid P_3\,] &= \textbf{\textit{Miracle}} && \text{(RD6)}\\
\textbf{\textit{Chaos}}\ ;\ [\,P_1 \vdash P_2 \mid P_3\,] &= \textbf{\textit{Chaos}} && \text{(RD7)}\\
[\,P_1 \vdash P_2 \mid P_3\,]\ ;\ \textbf{\textit{Miracle}} &= [\,P_1 \vdash P_2 \mid \textbf{\textit{false}}\,] && \text{(RD8)}\\
[\,P_1 \vdash P_2 \mid P_3\,]\ ;\ \textbf{\textit{Chaos}} &= [\,P_1 \wedge (P_3\ \textbf{\textit{wp}}_r\ \textbf{\textit{false}}) \vdash P_2 \mid \textbf{\textit{false}}\,] && \text{(RD9)}
\end{aligned}$$

All of these laws can be proved by calculation using the definitions in Theorem 5.1 and laws in Theorem 5.2. RD1 establishes that a choice between a ***Miracle*** and P yields P,

since the least deterministic behaviour is chosen. For the same reason, RD2 establishes that a choice between ***Chaos*** and P yields ***Chaos***. The reactive skip is a left and right identity for any contract P, as stated by RD3 and RD4. Law RD5 states that any non-terminating contract — that is where the postcondition is ***false*** — is a left zero for sequential composition, as clearly then Q is unreachable. Thus, in particular ***Miracle*** and ***Chaos*** are both left zeros for sequential composition, as shown by RD6 and RD7.

Law RD8 is a property first observed in [50]: placing a ***Miracle*** after a reactive design eliminates final states, and thus yields a non-terminating process. This is because it is impossible to reach a miraculous state, and thus inserting one effectively prunes transitions that lead to it.

Finally, RD9 is a similar law for ***Chaos***, which likewise removes final states. Crucially, however, the behaviour of ***Chaos*** is not impossible, but simply undesirable or unpredictable. Thus the composition additionally inserts an assumption P_3 $\mathbf{wp}_r$ ***false***, which effectively states that postcondition P_3 should not be established, because otherwise chaos will ensue. This explains Example 5.4: the left branch of the choice, $a \rightarrow$ ***Chaos*** is equivalent to $(a \rightarrow$ ***Skip***$)$; ***Chaos***. The postcondition of $a \rightarrow$ ***Skip*** is $st' = st \wedge$ ***tt*** $= \langle a \rangle$. The occurrence of ***Chaos*** mandates that this postcondition should not be established, which means that trace extension is negated and added to the assumption, yielding the reactive condition $\neg_r (\langle a \rangle \leq$ ***tt***$)$. This important distinction illustrates the difference between ***Miracle*** and ***Chaos*** – usually the latter is used to encode behaviour that should be prevented by the environment.

The next theorem gives the laws of reactive assignment.

Theorem 5.5 (Reactive Assignment Laws)**.**

$$\langle id \rangle_R = \mathit{II}_R \tag{RA1}$$
$$\langle \sigma \rangle_R \,;\, [\, P_1 \vdash P_2 \mid P_3 \,] = [\, \sigma \dagger P_1 \vdash \sigma \dagger P_2 \mid \sigma \dagger P_3 \,] \tag{RA2}$$
$$\langle \sigma \rangle_R \,;\, \langle \rho \rangle_R = \langle \rho \circ \sigma \rangle_R \tag{RA3}$$
$$\langle \sigma \rangle_R \,;\, \textbf{\textit{Miracle}} = \textbf{\textit{Miracle}} \tag{RA4}$$
$$\langle \sigma \rangle_R \,;\, \textbf{\textit{Chaos}} = \textbf{\textit{Chaos}} \tag{RA5}$$

Law RA1 establishes that an assignment using the identity function id yields the reactive skip. RA2 captures the effect of precomposing a reactive contract with an assignment; the assignment function is applied as a substitution in the pre, peri, and postconditions. RA3 states that composition of two assignments yields a single assignment built by composition of the individual assignment functions. Laws RA4 and RA5 establish that ***Miracle*** and ***Chaos*** are both right zeros for assignment. This is because they both remove final states, but, since assignments have no intermediate states, this eliminates all observable behaviours.

The final law of this section shows how we can demonstrate a refinement between two reactive designs. It requires that we weaken the precondition, and strengthen both the pericondition and postcondition.

Theorem 5.6 (Reactive Design Refinement)**.** $[\, P_1 \vdash P_2 \mid P_3 \,] \sqsubseteq [\, Q_1 \vdash Q_2 \mid Q_3 \,]$ *provided* $P_1 \Rightarrow Q_1$, $Q_2 \wedge P_1 \Rightarrow P_2$, *and* $Q_3 \wedge P_1 \Rightarrow P_3$.

Along with the laws for calculating reactive contracts previously covered, this law provides a way of proving properties of contracts using refinement and predicate and relational calculus.

6 Hybrid Reactive Designs

In this section we bring together all the results we have established as foundations in Sections 3–5 to construct our theory of hybrid reactive designs, which is the basis of the Modelica semantics. A hybrid reactive design is a reactive design contract whose pre, peri, and postconditions are specified using hybrid relations. Thus, a hybrid reactive contract specifies the assumptions and guarantees of a system in terms of constraints on the continuous variables. Moreover, unlike hybrid relations, the theory of hybrid reactive designs distinguishes intermediate and final observations, meaning that non-terminating hybrid processes can be described.

We will now proceed to define the operators for hybrid reactive designs. The first operator lifts a hybrid relation to a non-terminating reactive design.

Definition 6.1 (Hybrid Predicate)**.**

$$[P]_H \triangleq [\,\textbf{\textit{true}}_r \vdash P \mid \textbf{\textit{false}}\,]$$

The hybrid predicate simply states that the continuous variables evolve according to P in all intermediate states and there is no termination. It is useful to lift constraints on continuous variables to non-terminating hybrid processes that satisfy the constraints. For example, the reactive design $[x \textbf{\textit{ has-der }} y]_H$ is a partially specified hybrid process that constrains x to have derivative y, where the continuous variable y is otherwise unspecified. We can use this operator to encode differential constraints the sum of which forms a system of DAEs. Since this has a ***false*** postcondition and does not terminate, it follows, by Theorem RD5, that $[P]_H \mathbin{;} Q = [P]_H$, for any hybrid relation P and reactive design Q.

We can use this operator to lift function evolution from hybrid relational calculus to reactive designs.

Definition 6.2 (Function Evolution)**.**

$$x \Leftarrow f(ti) \triangleq [x \leftarrow f(ti)]_H$$

$$x \underset{[s,t]}{\Leftarrow} f(ti) \triangleq \left[\, \textbf{\textit{true}}_r \,\middle|\, x \underset{\leq t}{\leftarrow} f(ti) \,\middle|\, x \underset{[s,t]}{\leftarrow} f(t) \,\right]$$

The first operator, $x \Leftarrow f(t)$ describes a function evolution that doesn't terminate. In intermediate observations, any duration of evolution is possible, and there are no final observations. The second operator, $x \underset{[s,t]}{\Leftarrow} f(t)$, describes an evolution that terminates at some point in the interval $[s, t]$. In intermediate observations an evolution of any time up to t is possible, and in the final observation an evolution in the interval $[s, t]$ is observed. Moreover, the final state also populates the final values for state variables as per Definition 4.4.

ODE evolution is obtained by lifting of the corresponding hybrid relation operator.

Definition 6.3 (Ordinary Differential Equations)**.**

$$\langle\, x \bullet f \,\rangle_H \triangleq [\, \mathit{true}_r \vdash \langle\, x \bullet f \,\rangle \mid \mathit{false} \,]$$

Like $[P]_H$, this is a non-terminating operator and therefore is a left zero for any hybrid reactive design. The preemption operator is obtained similarly by lifting, but is slightly more complex.

Definition 6.4 (Preemption)**.**

$$[\, P_1 \vdash P_2 \mid P_3 \,] \blacktriangle \langle b \mid c \rangle \triangleq [\, P_1 \vdash P_2 \wedge \lceil b \rceil \mid P_3 \vee P_2 \triangle \langle b \mid c \rangle \,]$$

The precondition is simply the precondition of the preempted hybrid process. In the pericondition we require that P_2 holds and that the preemption invariant b must always hold as well. Usually, P_2 is an ODE or or a set of continuous constraints and thus the effect of this is to allow any permissible continuous evolution satisfying the invariant. The operator will terminate if P_3 is established — the hybrid process terminates (unusual) — or else P_2 exhibits a behaviour in which b holds and finally c holds, as provided by the hybrid relational preemption operator.

With these operator definitions, it is possible to lift Theorem 4.1 for hybrid relations to hybrid reactive designs.

Theorem 6.1 (Evolution Termination)**.** *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:

$$(x \Leftarrow f(ti) \blacktriangle \langle b \mid c \rangle) = \left(x \underset{[k..l]}{\Leftarrow} f(ti) \right)$$

Similarly, we have the following theorem for hybrid reactive design ODEs.

Theorem 6.2 (ODE Solution)**.** *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 $\langle\, x \bullet f \,\rangle_H = x \Leftarrow g(x, t)$.*

7 Modelica Semantics

In this section we will use our theory of hybrid reactive designs to give a denotational semantics to Modelica. For this deliverable our focus is on the block-based control law notation of the language, though we also revise our semantics for the core language,

adding more detail. In particular, our semantics of event handling is substantially more precise than our previous work [19]. In creating our formal semantics, we have drawn on the following principle sources:

- The Modelica Language Specification, version 3.4 [38], that provides a number of key details of the semantics in a mixture of natural language and formal mathematics.
- A paper from 2008 on the Modelica event handling mechanism [36], which gives a reasonably detailed semantics. The main contributions of this paper are also contained in the standard [38], but with less detail.
- The online Modelica reference guide[2].
- The online e-book "Modelica by Example" by Michael M. Tiller[3].

Additionally, we have used the OpenModelica tool[4] to conduct a number of experiments in order to gain an adequate intuition of language features. Thus, OpenModelica, rather than one of the other tools like Dymola[5] or Wolfram SystemModeler[6], serves as the benchmark for our semantics.

7.1 Semantics Overview

Our approach to the semantics of Modelica is to denote the core constructs of the language in the theory of hybrid reactive designs. Since we are interested in proof facilities for Modelica, our semantics is also mechanised in Isabelle/UTP. In terms of presentation, we give a mainly mathematical semantics and leave details of the mechanisation to later sections.

Fundamentally, a Modelica program is denoted by the set of possible evolutions of its variables; that is, a set of trajectories. This set of trajectories is characterised by the operators of hybrid reactive designs. This includes both a model's differential and algebraic equations, and its instantaneous events. The latter are catered for by the fact that timed traces are piecewise continuous. Thus, the trace of a Modelica model consists of an alternating sequence of continuous evolutions of the variables, followed by discrete jumps when events occur.

Structurally, Modelica allows the description of collections of models that collectively specify the possible behaviours of discrete and continuous variables. A "model", to use Modelica terminology, consists of the following principle concepts.

Parameters. These are static inputs to a model that are supplied at the point of instantiation, and can be used to encode constants.

Variables. Continuous and discrete variables for which the model can specify the behaviour. They can optionally have initial values associated. In the Modelica block language, these are used to encode inputs to and outputs from a given block. Every Modelica

[2]http://doc.modelica.org/
[3]http://book.xogeny.com/
[4]https://www.openmodelica.org/
[5]https://www.3ds.com/products-services/catia/products/dymola/
[6]http://www.wolfram.com/system-modeler/

model has a special built-in continuous variable called `time` whose derivative is always 1.

Equations. The core behavioural construct of a model. They describe how different quantities relate to one another. In a valid Modelica model, every variable must have its behaviour described by an equation to ensure determinism. Every equation can be conditional in nature, meaning it is only activated when its Boolean guard evaluates to true. Variables come in three varieties: differential, algebraic, and discrete equations, which are described below.

- Differential equations describe the behaviour of a continuous variable using a derivative, for example `der(x) = 5`.
- Algebraic equations link continuous variables using equality and various arithmetic operators, for example `x = y / z`.
- Discrete equations describe the valuation of discrete variables that are held constant during evolution, and can change only when events are triggered.

There are also initial equations that hold at $time = 0$, and can be any of the above kinds.

Events. Modelica is a hybrid systems modelling language, and thus also allows discontinuous updates to variables when "events" are triggered. There are two kinds of events in Modelica: state events and time events [36]. State events are triggered at the instant when a conditional predicate becomes satisfied by a model's evolving continuous variables, by moving its valuation from false to true, or true to false. State event conditions are modelled using zero crossing functions. Specifically, when particular monitored functions of the model cross zero, an event is raised. Time events occur at specific time instants, and are thus more predictable than state events, but can otherwise be handled similarly to state events. When an event is triggered two actions are performed.

Firstly, all discrete equations are reevaluated in context of the current value of the continuous variables. Since they are conditional, discrete equations can cascade, meaning evaluation of one leads to the condition of another becoming true, and thus needing to be evaluated. Thus discrete equations are applied iteratively until no more changes occur: in Modelica this process is called "event iteration" [38].

Secondly, the system of differential equations is reinitialised, using conditional "reset equations" (our term) that assign new values to continuous variables and thus form a new initial value problem. These are specified using the special `reinit()` operator. Reset equations are not mentioned in the standard, but the idea of special assignments following normal event iteration is implied by the Modelica reference manual. When describing the `reinit()` operator, the following comment is made:

"The reinit operator does not break the single assignment rule, because reinit(x,expr) in equations evaluates expr to a value (value), then at the end of the current event iteration step it assigns this value to x (this copying from values to reinitialized state(s) is done after all other evaluations of the model and before copying x to pre(x))."[7]

[7] https://build.openmodelica.org/Documentation/ModelicaReference.Operators.%27reinit()%27.html

Thus, reset equations are applied following event iteration, once application of the discrete equations has reached a fixed point.

Discrete and reset equations are specified in the body of a **when** clause. A typical when clause has the form **when** `P` **then** `Q;`. Here, `P` is a condition on the continuous variables. When the condition becomes true at some instant (having been false previously), the discrete equations in `Q` are instantaneously activated.

Consider, for example, the following bouncing ball example.

Example 7.1 (Bouncing Ball)**.**

```
model BouncingBall
  parameter g = 9.81;
  Real h; Real v;
initial equation
  v = 0;
  h = 2.0;
equation
  der(h) = v;
  der(v) = -g;
  when h <= 0 and v < 0 then
    reinit(v, -0.8*pre(v));
  end when;
end BouncingBall;
```

There is a single parameter g which is fixed gravitational constant. Actually, if we desired a more general model we could allow this to be given as a parameter at instantiation which would allow the simulation of bouncing balls on bodies other than Earth. There are two continuous variables: height h, and velocity v, both of type real. These are given initial values of 0 and 2.0, respectively. There are two differential equations which set the derivative of h to v, and the derivative of v to $-g$. There are no algebraic equations in this example.

The single when clause states that when the height becomes zero, and the velocity is negative, the following reset equation is instantaneously executed. This discrete equation, `reinit(v, -0.8*pre(v))`, sets up a new initial condition for v which is the previous value's inverse with a damping factor applied. This only holds at that very instant; it can alternatively be seen as an imperative assignment.

7.2 Core Language Semantics

Modelica models can be described using a 10-tuple $(\mathcal{D}, \mathcal{A}, \mathcal{Q}, I, D, A, Q, R, Z, T)$, whose components are as follows:

- $\mathcal{D}$ is a lens denoting the set of dynamic variables, that is variables which are described by differential equations.
- $\mathcal{A}$ is a lens denoting the set of algebraic variables, that is variables that do not appear differentiated.

- $\mathcal{Q}$ is a lens denoting the set of discrete variables.
- I is the initialisation predicate, that is the set of initial equations defined in the block.
- D is a hybrid relation denoting the set of (conditional) differential equations.
- A is a predicate denoting the set of (conditional) algebraic equations.
- Q is a predicate denoting the set of (conditional) discrete equations.
- R is a set of relations, each denoting a (conditional) reset equation.
- Z is a set of expressions, each denoting a zero-crossing function that can trigger a state event.
- T is a predicate denoting when time events can occur.

Lenses $\mathcal{D}$, $\mathcal{A}$, and $\mathcal{Q}$ partition the continuous state $\boldsymbol{c}$ and thus specify all variables that are characterised by the model's trajectory.

For the bouncing ball example, we have $\mathcal{D} = \{h, v\}$, $\mathcal{A} = \emptyset$, and $\mathcal{Q} = \emptyset$, since there are only dynamic variables in that model. Then we have $I = (v = 0 \wedge h = 2.0)$ for the initialisation, and $D = (h \textbf{ has-der } v \wedge v \textbf{ has-der } (-9.81))$ for the differential equations. There are no algebraic or discrete equations, and so $A = Q = \textbf{true}$ – that is, there are no constraints imposed. There is a single conditional reset equation which updates the ball's velocity upon impact, and so we have:

$$R = (v := -0.8 * pre(v)) \lhd (h \leq 0 \wedge v < 0) \rhd (v := pre(v))$$

This states that the reset is performed only provided the condition is satisfied, as stated in the **when** equation of Example 7.1. Otherwise, the variable v is simply left unchanged. Of course, in this model there is only one reset equation and one event, so v will always be changed in reality.

There are two potential zero crossings to be detected, for h and v, and thus we have $Z = \{h, v\}$. An event is thus triggered whenever a crossing is detected on one of these variables. Finally, there are no time events, only state events, and thus $T = \textbf{false}$.

Modelica models are simulator dependent: the precise semantics depends on the precision of a given numerical solver. For this reason, Modelica has a special fixed global real number constant called `eps`, a strictly positive number that denotes the smallest possible value a simulator can distinguish. This is necessary, for instance, because event preemption does not occur precisely at the point that a function crosses zero but slightly after it, as measured by `eps`. Thus, in our semantics we also encode `eps` as a fixed mathematical constant called ϵ.

At the level of hybrid reactive designs, the semantics of a Modelica model can then be denoted as follows.

Definition 7.1. Modelica Hybrid Reactive Design Semantics

$$
\begin{array}{l}
\mathit{solve}(I' \wedge Q' \wedge A') \;;\; \boldsymbol{c}_{pre} :=_{\boldsymbol{R}} \boldsymbol{c} \;; \\
\mu\, X \bullet ([D \wedge \lceil A' \rceil \wedge q \leftarrow q]_H \blacktriangle \langle (\neg T) \wedge \mathit{inv}(Z) \mid T \vee \mathit{exit}(Z) \rangle \;; \\
\quad \textbf{\textit{until}}\ (\boldsymbol{c} = \boldsymbol{c}_{pre})\ \textbf{\textit{do}} \\
\quad\quad \boldsymbol{c}_{pre} :=_{\boldsymbol{R}} \boldsymbol{c} \;; \\
\quad\quad \mathit{solve}(d' = d \wedge A' \wedge Q') \\
\quad \textbf{\textit{od}} \;; \\
\quad \mathit{interleave}(R) \;;\; \boldsymbol{c}_{pre} :=_{\boldsymbol{R}} \boldsymbol{c} \;;\; X)
\end{array}
$$

where

$$
\begin{array}{rcl}
\mathit{inv}(Z) & \triangleq & \bigwedge_{z \in Z} ((z' > -\epsilon) \lhd z \geq 0 \rhd (z' < \epsilon)) \\
\mathit{exit}(Z) & \triangleq & \bigvee_{z \in Z} ((z' < 0) \lhd z \geq 0 \rhd (z' > 0)) \\
\mathit{interleave}(A) & \triangleq & \left[\boldsymbol{true}_r \vdash \boldsymbol{false} \;\middle|\; \prod \mathrm{dom}(S) = A \bullet (\fatsemi\, P : S \bullet P) \right] \\
\mathit{solve}(P) & \triangleq & [\, \boldsymbol{true}_r \vdash \boldsymbol{false} \mid P \,]
\end{array}
$$

The first step is to find an initial condition for the differential equations. This is obtained by the hybrid contract with postcondition $I' \wedge Q' \wedge A'$, which we abbreviate using the *solve* function. This states that the initial value for variables must satisfy the initial equations, I, the discrete equations Q', and the algebraic equations A'. Usually, this collection of constraints such yield a deterministic assignment for the initial values.

Following initialisation, we populate the variable $\boldsymbol{c}_{pre}$ with a copy of the whole continuous state $\boldsymbol{c}$. The former is used to populate expressions that contain the Modelica function **pre** and also at event iteration. Then, the main behavioural loop begins, denoted by the fixed-point operator μ. The first evolution of the continuous variables is first performed. This evolution is characterised by three conjuncts in the hybrid predicate: $[D \wedge \lceil A' \rceil \wedge q \leftarrow q]_H$. The differential equations and algebraic equations are required to hold over the whole interval. Discrete variables are held constant, as denoted by third conjunct using the evolution operator.

The evolution of the system of equations is interrupted when either a time event or a state event is triggered. The bounds of the evolution is determined by both the invariant of the preemption operator ($\blacktriangle$) and also the preemption condition. The precise moment of state event triggering is non-deterministic within the interval $(-\epsilon, \epsilon)$. It is worth emphasising that both the evolution invariant and preemption condition are relations. The undashed variables refer to the instant when the present evolution began, and the dashed variables to every following instant. This allows us to compare the present continuous state to its initial value at every instant.

The evolution invariant states that evolution can continue so long as (1) no time event has occurred ($\neg T$), and (2) no zero crossing has occurred in the present evolution, described by $\mathit{inv}(Z)$. The $\mathit{inv}(Z)$ predicate states that, for all zero crossing functions, if the function

was positive initially then evolution can continue until it reaches $-\epsilon$, or if negative initially, then ϵ. This then gives the maximal bounds on when evolution can continue.

The evolution preemption condition states evolution can terminate either when (1) a time event has occurred (T), or (2) a zero crossing occurred, as described by $exit(Z)$. The $exit(Z)$ predicate states that, for any zero crossing function, if z was positive initially and it is now negative, or negative initially and now positive, then evolution can terminate. Thus evolution terminates between 0 and $-\epsilon$ for a decreasing function, and 0 and ϵ for an increasing function. Thus in our semantics an event is triggered whenever a zero crossing is detected for an increasing or decreasing function. This is important to ensure that an event is triggered both when a condition becomes true, having been false, and also vice-versa, so that events are enabled and disabled appropriately. The use of strict equality for 0 is also important to ensure that the function has gone beyond this and not stopped at 0.

If an event triggered, then event iteration commences. As in the initialisation phase, this involves assigning values to the different model variables. The set of constraints to be satisfied is, however, slightly different to those at initialisation. The set of constraints is calculated in two steps. Firstly, the ***until*** loop is used to iterate the system of algebraic and discrete equations to find a fixed-point. The iterated equation system holds all dynamic variables constant, indicated by $d' = d$, and conjoins all discrete and algebraic equations using *solve*. Iteration will continue until no more changes are being made to the continuous state by the discrete equations and thus $\boldsymbol{c} = \boldsymbol{c}_{pre}$. This algorithm closely mirrors the one given in the Modelica standard [38, page 273].

Following a terminated event iteration, it is necessary to apply reset equations which will assign new values to the dynamic variables for reinitialisation. The order of execution of the reset equations should not matter as all reset equations should have $\boldsymbol{c}_{pre}$ on the right-hand side instead of $\boldsymbol{c}$. Thus, we simply pick an arbitrary execution order for them using the function *interleave*. Finally, following all the resets, we copy $\boldsymbol{c}$ to $\boldsymbol{c}_{pre}$ for the final time and then iterate to commence continuous evolution again.

We will illustrate the semantics with the bouncing ball example. The initialisation for the bouncing ball is $solve(v' = 0 \wedge h' = 2.0)$, since there are not algebraic or discrete equations in this model. Since this fully describes the continuous state, this can be rewritten to an assignment $h, v :=_{\boldsymbol{R}} 2.0, 0$.

The evolution behaviour has the form:

$$[h \textbf{ has-der } v \wedge v \textbf{ has-der } (-9.81)]_H \blacktriangle \langle inv(Z) \mid exit(Z) \rangle$$

There are only differential equations in this model and so the expression $q \Leftarrow q$, to hold discrete variables, is vacuous. The evolution invariant, $inv(Z)$, is

$$((h' > -\epsilon) \lhd h > 0 \rhd (h' < \epsilon)) \wedge ((v' > -\epsilon) \lhd v > 0 \rhd (v' < \epsilon))$$

and the preemption condition $exit(Z)$ is

$$((h' < 0) \lhd h \geq 0 \rhd (h' > 0)) \wedge ((v' < 0) \lhd v \geq 0 \rhd (v' > 0))$$

Thus an event is triggered when either of h or v crosses zero in either direction. Again, this is non-deterministic and in reality the only relevant zero crossings for the model are

those for h. Nevertheless, both directions of zero crossing for h are necessary so that the bounce event can be both detected and re-enabled following execution. Thus we have

$$inv(Z) = ((h' > -\epsilon) \lhd h > 0 \rhd (h' < \epsilon)) \text{ and } exit(Z) = ((h' < 0) \lhd h \geq 0 \rhd (h' > 0))$$

With respect to event iteration, there are no discrete equations. Thus the body of the event iteration loop has the equation $solve(d' = d)$, which is equivalent to $solve(h' = h \wedge v' = v)$. Event iteration will thus terminate immediately with no state updates and can thus be simplified away. All that remains is handling of the single reset equation. Interleaving of the singleton set

$$interleave(\{(v := -0.8 * pre(v)) \lhd (h \leq 0 \wedge v < 0) \rhd (v := pre(v))\})$$

is equivalent to

$$(\boldsymbol{c}{:}v :=_{\boldsymbol{R}} -0.8 * \boldsymbol{c}_{pre}{:}v) \lhd (\boldsymbol{c}{:}h \leq 0 \wedge \boldsymbol{c}{:}v < 0) \rhd (\boldsymbol{c}{:}v :=_{\boldsymbol{R}} \boldsymbol{c}_{pre}{:}v)$$

Note that $pre(v)$ is rewritten to $\boldsymbol{c}_{pre}{:}v$ – the value of v as it was in the previous valuation of the continuous state. Thus the overall semantics of the bouncing ball is as follows.

Example 7.2 (Bouncing Ball Semantics)**.**

$$\boldsymbol{c}{:}h, \boldsymbol{c}{:}v :=_{\boldsymbol{R}} 2.0, 0 \; ; \; \boldsymbol{c}_{pre} :=_{\boldsymbol{R}} \boldsymbol{c} \; ;$$

$$\mu\, X \bullet \left(\begin{array}{l} [h \textbf{ has-der } v \wedge v \textbf{ has-der } (-9.81)]_H \blacktriangle \left\langle \dfrac{(h' > -\epsilon) \lhd h > 0 \rhd (h' < \epsilon)}{(h' < 0) \lhd h \geq 0 \rhd (h' > 0)} \right\rangle ; \\ \boldsymbol{c}_{pre} :=_{\boldsymbol{R}} \boldsymbol{c} \; ; \\ (\boldsymbol{c}{:}v :=_{\boldsymbol{R}} -0.8 * \boldsymbol{c}_{pre}{:}v) \lhd (\boldsymbol{c}{:}h \leq 0 \wedge \boldsymbol{c}{:}v < 0) \rhd (\boldsymbol{c}{:}v :=_{\boldsymbol{R}} \boldsymbol{c}_{pre}{:}v) \; ; \; X \end{array} \right)$$

To aid readability, we separate the invariant and condition of the preemption with a horizontal rather than vertical bar. In the first iteration of this model, v and h are assigned initial values, and then the continuous evolution begins. At some point when $h \in (0, \epsilon)$, the first event is triggered. Since at this point h is below 0, and v is negative, the assignment of $-0.8 * \boldsymbol{c}_{pre}{:}v$ to $\boldsymbol{c}{:}v$ will occur, and then the second evolution will commence from that point.

The second event occurs, not on the second bounce, but when h crosses 0 again in the opposite direction. The reason for this is that the event condition switches from true to false. In order to register it moving from false to true again, for the second bounce, it is necessary to account for the condition changing every time. At this point the condition is not true, and thus v is simply equated with its previous value. The third event will then be the second bounce, and the model continues in this manner.

Having given a semantics for the core language, we will next turn our attention to Modelica blocks.

7.3 Modelica Blocks

Though Modelica is, at its core, a textual language for describing dynamical systems, it also has a sophisticated diagrammatic frontend for traditional block-based control law diagrams. Modelica control blocks are a specialisation of models that divide the continuous variables into inputs, outputs, and internal variables. A Modelica block diagram describes the behaviour of all continuous variables by a collection of blocks with connections from inputs to outputs. These connections effectively become additional algebraic equations of the model that tie together pairs of variables in different blocks.

Modelica has a large library of blocks in the `Modelica.Blocks` namespace including:

- `Continuous` blocks, like Integrator and Derivative
- `Discrete` blocks, that sample at a fixed time period
- `Logical` blocks, that implement logic gate style blocks
- `Math` blocks, that include arithmetic, trigonometry, and other related functions
- `Sources` blocks, that include various signal generators

An example block diagram is shown in Figure 6, which represents a similar bouncing ball model as that described in Example 7.1. We use a number of blocks from Modelica's standard library including an `Integrator` from the `Modelica.Blocks.Continuous` namespace (Height), and a `Gain` block from `Modelica.Blocks.Math` (Restitution). Modelica also allows straightforward definition of customised blocks, and we therefore also define two of our own blocks. For convenience, we add textual labels to each of the connections in the diagram in dark red.

The core dynamical behaviour of this diagram is provided by two integrator blocks, called respectively Velocity and Height. The Velocity block corresponds to the differential equation $\dot{v} = -9.81$, and the Height block to the differential equation $\dot{h} = v$. The gravitional constant is supplied to the former integrator by a constant block called *Gravity* which provides a constant signal output, and the input of Height is connected to the output of Velocity. The Height block is an instance of the Modelica `Integrator` block, which takes a single input signal, and provides the integrated signal output. The Velocity block is an adapted block called `ResetIntegrator`. The latter takes two inputs, in addition to the middle one that takes the signal to integrate. The top input is an initial value, which is used to construct the initial value problem. Initially, this value is supplied by the constant block InitVel that supplies a constant 0. The bottom input is a Boolean signal that can be used to reset the integration; that is to halt integration and form a new initial value problem. This is an example of event iteration.

The remainder of the diagram handles the associated hybrid aspects of the system. The logic of when a bounce occurs is encoded by three blocks at the bottom right. We use the `And` logic block to conjoin two conditions. If the present velocity is less than 0, as characterised by top inequality block, and the height is less than or equal to zero, as characterised by the bottom inequality, then the bounce is signalled by the output from the block and1. This is fed into both the Velocity integrator to trigger a reset, and also into a block called IvpSwitch. IvpSwitch is used to decide the input initial value to the

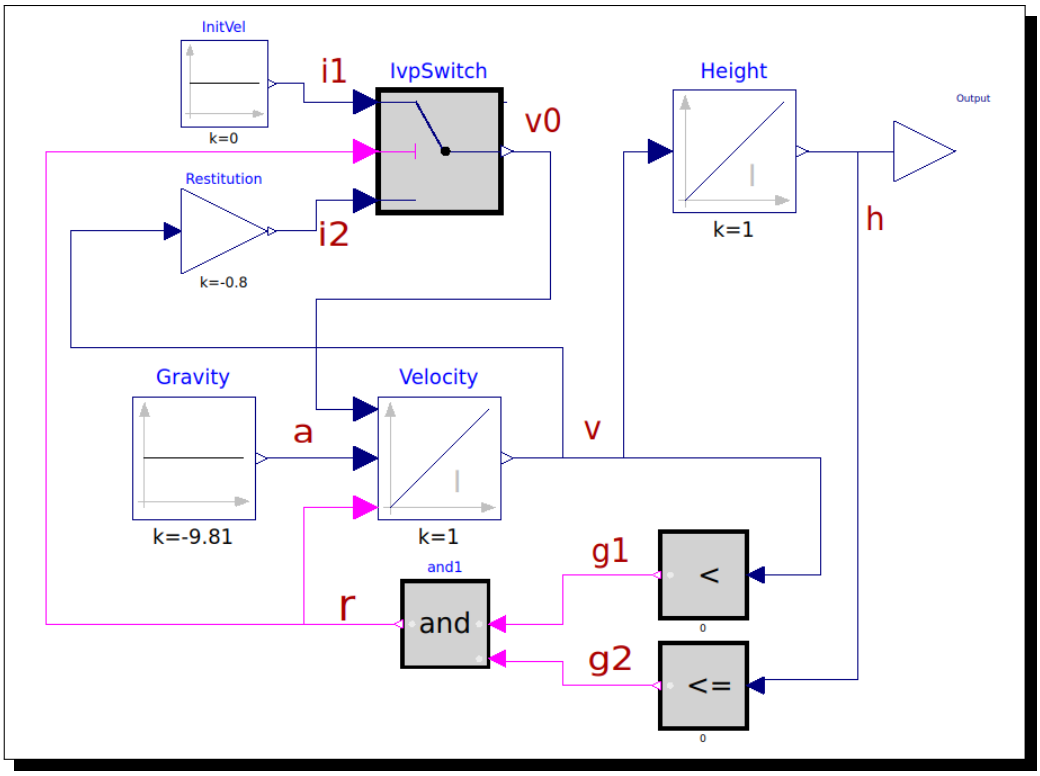


Figure 6: Bouncing Ball in Modelica blocks

Velocity integrator. Initially this value is taken from the InitVel block. However, once the Boolean input is received on the central connector then the output is calculated using a gain block called Restitution as a multiple of the present output from Velocity and -0.8: the coefficient of restitution. Thus a new initial condition is supplied to the Velocity block, and integration continues.

The simulation of this example in the OpenModelica tool is shown in Figure 7 over a 5 second interval. We capture data from two continuous variables. The red line is the height of the ball, which follows the expected decreasing bounces. The blue line is the output of the <= block: it shows the instants at which the height hits zero and thus the resets are triggered.

Having introduced the control law notation, we now illustrate the internals of these block definitions with the two custom block types: `ResetIntegrator` and `Toggle`.

Example 7.3 (`ResetIntegrator` Block).

```
block ResetIntegrator
  import Modelica.Blocks.Types.Init;
  import Modelica.Blocks.Interfaces.*;
  parameter Real k(unit = "1") = 1;
  extends Modelica.Blocks.Interfaces.SISO(y);
  RealInput y_start;
  BooleanInput reset;
initial equation
  y = y_start;
equation
```

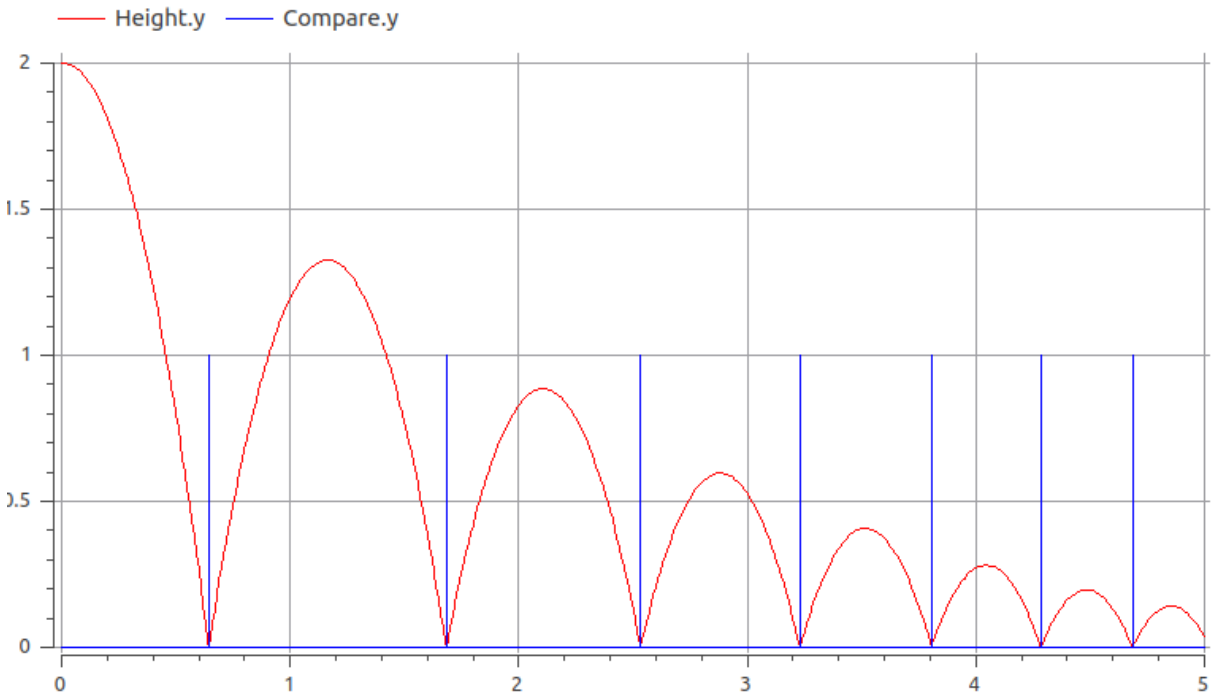


Figure 7: Simulation of block-based bouncing ball

```
  der(y) = k * u;
  when reset then
    reinit(y, y_start);
  end when;
end ResetIntegrator;
```

Such a definition would normally include annotation information for documentation and illustration of the block in the tool. We elide these details here. The `ResetIntegrator` definition imports several type for interfaces and initialisation from the Modelica standard library. It has a single parameter, k, which is a coefficient that can be used to scale the input signal and by default takes the value 1. The block extends the interface `SISO`, which is the class of single input / single output blocks. This superclass provides a template where there is a single input called `u` and a single output called `y`, both of which are real number inputs. In addition we add two further inputs: `y_start`, a real number input which gives the initial value for the output, and `reset` a Boolean input that determines when the block should reset.

The remainder of the block definition follows a typical Modelica model structure. The single initial equation states that the initial output has the same value as `y_start`. The first regular equation states that the derivative of the output `y` is the product of the input `u` and the coefficient `k`. The second equation is a **when**-clause: it states that when the `reset` variable becomes true, `y` is instantaneously identified with `y_start` again. This sets up the new initial value problem, following event iteration.

Example 7.4 (`Toggle` Block).

```
block Toggle
  imports Modelica.Blocks.Interfaces.*;
  extends Modelica.Blocks.Icons.PartialBooleanBlock;
  BooleanInput u2;
  RealInput u3;
  RealOutput y;
  RealInput u1;
initial equation
  y = u1;
equation
  when u2 then
    y = u3;
  elsewhen not u2 then
    y = u1;
  end when;
end Toggle;
```

The `Toggle` block is similar to Modelica block `Switch`, but is explicitly event-driven. The block switches between two real inputs `u1` and `u3` based on the valuation of Boolean input `u2`. Initially, the output `y` is identified with `u1`, and is held at this value. When `u2` changes value, the behaviour is changed so that `y` is held at `u3`'s value at that instant over the subsequent evolution. If `u2` changes in the other direction, `y` is switched back to `u1` at that instant.

Every Modelica block has a definition similar to the two given above. The entire diagram illustrate in Figure 6 is described by the following model.

Example 7.5 (Bouncing Ball Block Composition).

```
model BallBlocks

  imports Modelica.Blocks.Continuous.*;
  imports Modelica.Blocks.Interfaces.*;
  imports Modelica.Blocks.Logical.*;
  imports Modelica.Blocks.Math.*;
  imports Modelica.Blocks.Sources.*;

  ResetIntegrator Velocity;
  Constant Gravity(k = -9.81);
  Toggle IvpSwitch;
  Constant InitVel(k = 0);
  Gain Restitution(k = -0.8);
  Integrator Height(y_start = 2);
  RealOutput Output;
  LessEqualThreshold Compare;
  LessThreshold lessthreshold1;
  And and1;
```

```
equation
  connect(Restitution.y, IvpSwitch.u3);
  connect(Height.y, Output);
  connect(Gravity.y, Velocity.u);
  connect(Height.y, Compare.u);
  connect(Velocity.y, Height.u);
  connect(and1.y, IvpSwitch.u2);
  connect(lessthreshold1.y, and1.u1);
  connect(and1.y, Velocity.reset);
  connect(Compare.y, and1.u2);
  connect(Velocity.y, lessthreshold1.u);
  connect(Velocity.y, Restitution.u);
  connect(InitVel.y, IvpSwitch.u1);
  connect(IvpSwitch.y, Velocity.y_start);

end BallBlocks;
```

The diagram model first imports the parts of the Modelica block library it depends on. Then it instantiates each of the blocks using the appropriate classes, with static parameters supplied where necessary. Finally, the connections between each of the classes is given, each of which corresponds to an algebraic equality.

This completes our overview of the Modelica block language. In the next section we will show how such behaviours can be given a denotational semantics in our UTP theory of hybrid reactive designs.

7.4 Block Semantics

In this section we show how Modelica blocks are formally modelled as hybrid reactive designs. In Section 7.2 we described the denotational semantics for Modelica models. Behaviourally, blocks simply specialise this semantics and so the semantics of block diagrams will show how to construct such models by composition. Effectively, we give a formal semantics to the Modelica flattening procedure, in that the input is a collection of blocks and connections, and the output is a flat Modelica model.

Every block diagram is denoted by two parts: (1) the alphabet of continuous variables that will be used to represent connections between blocks and internal variables, and (2) the collection of composed instantiated blocks. Each block is generated by a function whose arguments consist of the static parameters associated with the block class, and any input and output connections, and the output is a model 7-tuple (I, D, A, Q, R, Z, T). Our semantics is variable-centric in that blocks do not export named continuous variables which must be then wired up, but these connections are defined independently as part of the continuous state and then passed to the block constructors. This has the advantage that the set of equations is minimised, and thus reduces the number of sparse equations.

We begin with semantics for some of the arithmetic blocks from Modelica.Blocks.Math[8].

Definition 7.2 (Mathematical Blocks)**.**

$$\begin{aligned}
\mathit{MathBlk}(e, y) &\triangleq (\textbf{\textit{true}}, \textbf{\textit{true}}_r, y = e, \textbf{\textit{true}}, \emptyset, \emptyset, \textbf{\textit{false}}) \\
\mathit{Gain}(k, u, y) &\triangleq \mathit{MathBlk}(k * u, y) \\
\mathit{Add}(k1, k2, u1, u2, y) &\triangleq \mathit{MathBlk}(k1 * u1 + k2 * u2, y) \\
\mathit{Product}(u1, u2, y) &\triangleq \mathit{MathBlk}(u1 * u2, y) \\
\mathit{Division}(u1, u2, y) &\triangleq \mathit{MathBlk}(u1/u2, y)
\end{aligned}$$

Arithmetic blocks usually take a collection of inputs, and produce a single output y that is the result of applying the operator. All such blocks are described in terms of meta-block definition *MathBlk* that takes an expression e defining the blocks behaviour, and a variable y which is the output. The majority of the behaviour of such a block is trivial: is has no initialisation equations, differential equations, discrete equations, reset equations, state events, or time events. The only affirmative behaviour is the algebraic equation that associates the output with the inputs.

The *Gain* block takes a single static parameter k, the gain coefficient, the input variable u and the output variable y. Its behaviour is given by *MathBlk*, where the defining expression equates y to $k * u$. The other math blocks can be defined similarly. Compare our definition of the gain block with the following Modelica fragment.

Example 7.6.

```
block Gain "Output the product of a gain value with the input
    signal"
  parameter Real k(start=1) "Gain value multiplied with input
      signal";
  RealInput u "Input signal connector";
  RealOutput y "Output signal connector";
equation
  y = k*u;
end Gain;
```

As we can see, for this kind of block, the semantic mapping is almost direct. By convention, in our mathematical definitions we will usually abbreviate variable names longer than two or three characters to aid readability. We can similarly give a semantics to source blocks.

[8] A description of all the blocks we here consider can be found at https://build.openmodelica.org/Documentation/Modelica.Blocks.html

Definition 7.3 (Source Blocks)**.**

$$\begin{aligned}
\mathit{SrcBlk}(e, y, T) &\triangleq (\textbf{\textit{true}}, \textbf{\textit{true}}_r, y = e, \textbf{\textit{true}}, \emptyset, \emptyset, T) \\
\mathit{Clock}(o, sT, y) &\triangleq \mathit{SrcBlk}(o + (0 \lhd time < sT \rhd time - sT), y, time = sT) \\
\mathit{Constant}(k, y) &\triangleq \mathit{SrcBlk}(k, y, \textbf{\textit{false}}) \\
\mathit{Step}(h, o, sT, y) &\triangleq \mathit{SrcBlk}(o + (0 \lhd time < sT \rhd h), y, time = sT) \\
\mathit{Ramp}(h, d, o, sT, y) &\triangleq \mathit{SrcBlk}\left(\begin{array}{l} o + \left(\begin{array}{l} 0 \lhd time < sT \rhd (time - sT) * h/d \\ \quad \lhd time < sT + d \rhd h \end{array}\right) \\ , y, time \in \{sT, sT + d\} \end{array}\right) \\
\mathit{Sine}(a, f, p, o, sT, y) &\triangleq \mathit{SrcBlk}(o + (0 \lhd time < sT \rhd a \cdot sin(2 \cdot \pi \cdot f \cdot (time - sT) + p)), y)
\end{aligned}$$

Our meta-block definition *SrcBlk* is similar to *MathBlk*, but source blocks often have time events associated, and we add the parameter T to characterise these. The *Clock* block produces a signal that mimics the $time$ variable, but with an offset and start time. It thus has two parameters, offset o and start time sT. The behaviour sets y to o when $time < sT$, and otherwise to $time - sT$. Since the derivative of y thus changes at time sT, we raise a time event at this point indicated by the expression $time = sT$. The *Constant* and *Step* blocks have similar definitions. The *Ramp* block is a little more complicated as it describes three phases of the signal: it is 0 initially, when $time = sT$ the signal begins to increase toward h, and finally when $time = sT + d$ it settles to h. Thus, there are two time events associated with this block to intersperse these two phases. The *Sine* block generates a sine wave signal on y with given amplitude, frequency, phase, offset and start time.

We can similarly give semantics to some of the logic blocks.

Definition 7.4 (Logic Blocks)**.**

$$\begin{aligned}
\mathit{LogBlk}(e, y, Z) &\triangleq (\textbf{\textit{true}}, \textbf{\textit{true}}_r, \textbf{\textit{true}}, y = e, \emptyset, Z, \textbf{\textit{false}}) \\
\mathit{And}(u1, u2, y) &\triangleq \mathit{LogBlk}(u1 \wedge u2, y, \emptyset) \\
\mathit{Or}(u1, u2, y) &\triangleq \mathit{LogBlk}(u1 \wedge u2, y, \emptyset) \\
\mathit{Not}(u, y) &\triangleq \mathit{LogBlk}(\neg u, y, \emptyset) \\
\mathit{Xor}(u1, u2, y) &\triangleq \mathit{LogBlk}((u1 \wedge u2) \vee (\neg u1 \wedge \neg u2), y, \emptyset) \\
\mathit{Nor}(u1, u2, y) &\triangleq \mathit{LogBlk}(\neg(u1 \vee u2), y, \emptyset) \\
\mathit{GreaterThreshold}(thr, u, y) &\triangleq \mathit{LogBlk}(u > thr, y, \{|u - thr|\}) \\
\mathit{GreaterEqualThreshold}(thr, u, y) &\triangleq \mathit{LogBlk}(u \geq thr, y, \{|u - thr|\}) \\
\mathit{LessThreshold}(thr, u, y) &\triangleq \mathit{LogBlk}(u < thr, y, \{|u - thr|\}) \\
\mathit{LessEqualThreshold}(thr, u, y) &\triangleq \mathit{LogBlk}(u \leq thr, y, \{|u - thr|\})
\end{aligned}$$

LogBlk is similar to *MathBlk*, but it exports a discrete equation rather than an algebraic equation. Moreover it has a parameter for zero crossings events, which are necessary for

blocks that can change state depending on a real-valued input. The operators of Boolean logic are denoted using ***LogBlk*** by lifting the corresponding logic operators. They blocks therefore calculate an initial value for their Boolean output y, and then hold this constant until an event is triggered.

The threshold blocks are similar, but they export zero crossing events as well. Specifically, an event is triggered when one of these blocks detects that the absolute difference between its input and the given threshold parameter, thr, crosses zero. At this point all discrete equations are re-evaluated, and thus the threshold block, plus any other logic blocks in the diagram, can change output at this point.

We give a semantics to the Integrator from Modelica.Blocks.Continuous.

Definition 7.5 (Integrator)**.**

$$\mathit{Integrator}(iT, k, y_0, u, y) \triangleq \left(\begin{array}{l} (iT \in \{\mathit{InitialState}, \mathit{InitialOutput}\} \Rightarrow y = y_0) \wedge \\ (iT = \mathit{SteadyState} \Rightarrow k * u = 0), \\ y \ \textbf{\textit{has-der}}\ k * u, \textbf{\textit{true}}, \textbf{\textit{true}}, \emptyset, \emptyset, \textbf{\textit{false}} \end{array} \right)$$

The Integrator block takes three parameters: initialisation type iT, an integration coefficient k, and initial output value y_0. It has a single input u, and output y. The dynamic behaviour is defined by a simple continuous equation, which states that the derivative of y is $k * u$, and there are no algebraic equations, discrete equations, or events. The initialisation predicate is more complex though as there are four possibilities for iT. If $iT = \mathit{NoInit}$, then no particular initialisation is given, and thus the initialisation predicate will be ***true***. If it is *InitialState* or *InitialOutput* then y is initially identified with the parameter y_0. Finally, if it is *SteadyState* then $k * u$ must take the value 0 initially.

Our more complex ResetIntegrator block also adds events.

Definition 7.6 (Integrator with Reset)**.**

$$\mathit{ResetIntegrator}(k, u, y, y_0, r) \triangleq \left(\begin{array}{l} y = y_0, y \ \textbf{\textit{has-der}}\ u, \textbf{\textit{true}}, \textbf{\textit{true}}, \\ (y := \mathit{pre}(y_0)) \lhd r \rhd \mathit{II}, \emptyset, \textbf{\textit{false}} \end{array} \right)$$

It has a single parameter k, the integration gain. The four connections are input u, output y, initial value input y_0, and reset flag input r. The equations are similar to those for the *Integrator*, but we have simplified the initial equation. The major difference is the addition of the reset equation that, upon an event, will reassign y depending on the value of r. If true, y will be initialised to the value of y_0 at that instant. Otherwise, no changes will be made. Interestingly, this block has no actual events associated with it. This is because the origin of the event is not this block itself, but another block, like *GreaterThreshold*, that does detect zero crossings.

Next, we exemplify discrete blocks with the *Sampler* block.

Definition 7.7 (Sampler Block)**.**

$$\mathit{DisBlk}(e, y, T) \triangleq (\textbf{\textit{true}}, \textbf{\textit{true}}_r, \textbf{\textit{true}}, y = e, \emptyset, \emptyset, T)$$

$$\mathit{sample}(t, p) \triangleq (\exists\, k \bullet \mathit{time} = t + k \cdot p)$$

$$\mathit{Sampler}(sP, sT, u, y) \triangleq \mathit{DisBlk} \left(\begin{array}{l} y = u \lhd \mathit{time} = 0 \vee \mathit{sample}(sT, sP) \rhd u = \mathit{pre}(u), \\ y, \mathit{sample}(sT, sP) \end{array} \right)$$

The ***Sampler*** block sets its output y to have the same value as input y at regular sample intervals, defined by a sample period (sP) and start time (sT). It is defined in terms of the function *sample* that determines whether the current value of *time* is one of the sample instants. The sample block raises a time event whenever this is the case, and at these points the discrete equation $y = u$ is applied to sample the input.

The final block we describe is our custom toggle block from Definition 7.4.

Definition 7.8 (Toggle Block)**.**

$$\textit{Toggle}(u1, u2, u3, y) \triangleq (y = u1, \textbf{\textit{true}}_r, \textbf{\textit{true}}, y = u3 \lhd u2 \rhd u1, \emptyset, \emptyset, \textbf{\textit{false}})$$

The toggle block simply switches the output y between $u3$ and $u1$, depending on the value of Boolean input $u2$.

7.5 Model Composition

Having give block semantics for a portion of the Modelica library, we now show how to give semantics to a Modelica block diagram. Naturally, block constructors are only partial and the semantics of several blocks must be composed to produce an overall control law diagram. We therefore define the following operator for composing model fragments.

Definition 7.9 (Model Composition)**.**

$$\begin{aligned}(I_1, D_1, A_1, Q_1, R_1, Z_1, T_1) \oplus_m (I_2, D_2, A_2, Q_2, R_2, Z_2, T_2) \triangleq \\ (I_1 \wedge I_2, D_1 \wedge D_2, A_1 \wedge A_2, Q_1 \wedge Q_2, R_1 \cup R_2, Z_1 \cup Z_2, T_1 \vee T_2)\end{aligned}$$

Composition of a model from parts involves conjunction of the equations, including initial, differential, algebraic, and discrete equations. We also take the union of the sets of reset equations and zero crossing functions, and the disjunction of all time event conditions. Using this operator, and the block constructors defined above, we can construct composite Modelica models. However, it is also necessary to define the behaviour of the built-in continuous variable *time*, which is described by the model fragment below.

Definition 7.10 (Core Time)**.**

$$\textit{CoreTime} \triangleq (time = 0, time \ \textbf{\textit{has-der}}\ 1, \textbf{\textit{true}}, \textbf{\textit{true}}, \emptyset, \emptyset, \textbf{\textit{false}})$$

CoreTime ensures that the *time* variable is 0 initially, and the differential equation causes it to advance in line with the global clock.

With these definitions we can now give a semantics to the Bouncing Ball example from Figure 6. We first define the alphabet of this diagram, which is

$$\{a : \mathbb{R}, g1 : \mathbb{B}, g2 : \mathbb{B}, h : \mathbb{R}, i1 : \mathbb{R}, i2 : \mathbb{R}, r : \mathbb{B}, v0 : \mathbb{R}, v : \mathbb{R}\}$$

each variable of which corresponds to a connection in the diagram. Moreover, these variables can be split into the three sub-classes:

1. dynamic variable $\mathcal{D} \triangleq \{v, h\}$;
2. algebraic variables $\mathcal{A} \triangleq \{a, i1, i2\}$;
3. discrete variables $\mathcal{Q} \triangleq \{g1, g2, r, v0\}$.

With the continuous state-space defined, the model itself is defined as follows.

Example 7.7 (Bouncing Ball Block Semantics)**.**

$$\begin{aligned} \mathit{BouncingBall} \triangleq\ & \mathit{Constant}(0, i1) \oplus_m \\ & \mathit{Gain}(-0.8, v, i2) \oplus_m \\ & \mathit{Toggle}(i1, r, i2, v0) \oplus_m \\ & \mathit{Constant}(-9.81, a) \oplus_m \\ & \mathit{ResetIntegrator}(1, a, v, v0, r) \oplus_m \\ & \mathit{Integrator}(v, h) \oplus_m \\ & \mathit{LessThreshold}(0, v, g1) \\ & \mathit{LessEqualThreshold}(0, h, g2) \\ & \mathit{And}(g1, g2, r) \end{aligned}$$

This then gives the overall behaviour of the model as a hybrid reactive design.

8 Mechanisation in Isabelle/UTP

We have mechanised the majority of the work described in this deliverable in our proof assistant Isabelle/UTP[9], including our theories of generalised reactive processes, hybrid relations, reactive contracts, hybrid reactive designs, and finally the semantics of Modelica blocks. This includes all the definitions we've given and mechanical proof of all of the theorems. On the whole the syntax of Isabelle/UTP closely mirrors the mathematical syntax presented in our deliverables, though there are some notable changes. Details of these can be found online in our syntax document[10]. Our UTP development can be found in our online git repository[11].

A major advantage that the use of Isabelle/HOL has provided is its substantial mechanisation of real number theory, topological spaces, and calculus, all in the scope of the Multivariate Analysis package [23]. This has been invaluable for allowing us to reason about continuous and hybrid systems. Moreover, we have also utilised the HOL-ODE package [33, 32] which provides support of ODEs, initial value problems, and their solutions.

The foundation of all our mechanisation work is the theory of timed traces that acts as the underlying trace model for the hybrid relational calculus. This work can be found under `dynamics/Timed_Traces.thy`. It includes the definition of types for contiguous functions and then piecewise continuous and convergent functions along with all the

[9] Isabelle/UTP website: https://www-users.cs.york.ac.uk/~simonf/utp-isabelle/
[10] https://github.com/isabelle-utp/utp-main/raw/master/doc/syntax/utp-syntax.pdf
[11] https://github.com/isabelle-utp/utp-main

```
lemma hoare_ex_1:
  "{|true|}(z := &x) ◁ (&x ≥u &y) ▷r (z := &y){|&z =u maxu(&x, &y)|}u"
  by (hoare_auto)

lemma hoare_ex_2:
  assumes "X > 0" "Y > 0"
  shows
  "{|&x =u «X» ∧ &y =u «Y»|}
    while ¬(&x =u &y)
    invr &x >u 0 ∧ &y >u 0 ∧ (gcdu(&x,&y) =u gcdu(«X»,«Y»))
    do
       (x := (&x - &y)) ◁ (&x >u &y) ▷r (y := (&y - &x))
    od
    {|&x =u gcdu(«X», «Y»)|}u"
  using assms by (hoare_auto, (metis gcd.commute gcd_diff1)+)
```

Figure 8: Isabelle/UTP Hoare Logic tactic

operators defined in Section 3 of the deliverable for shifting timed traces and concatenating traces. We have also proved all the theorems in that Isabelle file. Altogether, this amount to approximately 1500 lines of Isabelle definitions and proof.

The Isabelle/UTP core itself, located under the `core/` directory, has grown substantially to support the development of the theories defined in the deliverable. In particular, our theory of lenses, that allows us to account for both discrete and continuous state, has now spun off to its own development which can be found on the Isabelle *Archive of Formal Proofs* [12]. The Isabelle/UTP core now amounts to approximately 10,000 lines of Isabelle definitions and proof. This includes the laws of predicate and relational calculus, meta-logical operators like substitution, Hoare logic, weakest precondition calculus, operational semantics, and an array of proof tactics for various kinds of program verification.

The Hoare logic tactic, called `hoare_auto`, in particular are being employed in WP2 sister deliverables D2.3a [54] and D2.3d [11] to provide a verification technique for FMI multi-models. An example of the use of this tactic can be seen in Figure 8. The tactic applies Hoare logic introduction laws to break a goal into a series of verification conditions which can then be discharged using Isabelle's array of automated proof tactics. These tactics, in particular, are also being employed in WP1 deliverables D1.3b [42] and D1.3d [14] to verify the railway and buildings case studies, respectively. All of this shows the added value of having mechanised semantics: it brings formal verification facilities to the real world problems. It is worth stressing that all of this development ensures that, ultimately, all the semantics given in this deliverable boils down to the axioms of set theory, and thus ensures that are foundations are sound mathematics.

Founded on the core, we have also mechanised the theories of generalised reactive processes, designs, and reactive designs, including a large number of generally applicable laws, as exhibited in Section 5, and also more specialised laws for *Circus* which are also being applied in the context of the FMI semantics deliverable D2.3c [55]. Again, all of these laws have been proved utilising the tactics from Isabelle/UTP core. This develop-

[12] https://www.isa-afp.org/entries/Optics.html

```
theorem ode_solution':
  assumes
    "vwb_lens x"
    "⋀ x l. l > 0 ⟹ (𝓕(x) usolves_ode 𝓕' from 0) {0..l} UNIV"
    "⋀ x. 𝓕(x)(0) = x"
  shows "⟨x • 𝓕'(ti)⟩ₕ = x ←ₕ «𝓕»($x)ₐ(«ti»)ₐ"
  by (simp add: assms(1) assms(2) assms(3) ode_solution)
```

Figure 9: Solutions to ODEs in Isabelle/UTP

```
definition ResetIntegrator ::
  "real ⇒ (real, 'l, 'c) mcon ⇒ (real, 'l, 'c) mcon ⇒
   (real, 'l, 'c) mcon ⇒ (bool, 'l, 'c) mcon ⇒ ('l, 'c) mblock" where
[upred_defs, mo_defs]:
  "ResetIntegrator k u y y_start r =
    ⦇ mieqs = (&y =ᵤ &y_start)
    , mdeqs = y has-der («k»*&u)
    , maeqs = true
    , mqeqs = true
    , mreqs = {($y´ =ᵤ $y_start) ◁ &r ▷ᵣ ($y´ =ᵤ $y)}
    , mzcfs = {}
    , mtevs = (λ t. False)
    ⦈"
```

Figure 10: Reset Integrator in Isabelle/UTP

ment of pre-hybrid UTP theories amount to approximately 10,000 lines of Isabelle. These key UTP theories can be found in the repository under `theories/`.

On top of the theory development outlined, we have also constructed the UTP theory of hybrid relations, including all the operators described in Section 4 and their laws. As an example, we demonstrate the proven theorem for ODE solution in Figure 9 which corresponds to Theorem 4.3. This theorem in particular builds on the HOL-ODE package [33, 32]. The theorem has as parameters a lens x that captures the n continuous variables of the ODE, $\mathcal{F}' : \mathbb{R} \to \mathbb{R}^n \to real^n$ which is the characteristic function of the ODE, $\mathcal{F} : \mathbb{R}^n \to \mathbb{R} \to \mathbb{R}^n$ which is the solution function. The three assumption of the theorem are that (1) x must be a very well-behaved lens; (2) $\mathcal{F}$ must be the unique solution to ODE $\mathcal{F}$ at every time instant $l > 0$; and (3) $\mathcal{F}$ must return the initial value at time 0. The usolves-ode primitive, from HOL-ODE describes a unique solution to an ODE. Provided those three assumptions are met, this theorem can invoked to rewrite an ODE to an evolution using the solution.

The hybrid relations development is then combined with reactive designs to provide the theory of hybrid reactive designs. This then allows to merge all the theorems that have come from both of these theories and thus illustrates the UTP's approach to semantics by theory integration. Hybrid relations and reactive designs together represent approximately 2500 lines of Isabelle.

Finally, the theory of hybrid reactive designs is applied to give mechanised semantics to Modelica blocks. In Isabelle a block is a record with seven fields that correspond to the latter seven elements of 10-tuple described in Section 7. An example is shown in Figure 10, which gives the mechanised definition of the *ResetIntegrator* from Definition 7.6. All the

blocks in the library we have encoded follow this typical form.

The definition is a function that constructs an instance of the block, from the given arguments that correspond to block parameters and connectors. The first parameter of type real is the single parameter k which is the gain constant. The next three arguments have type $(\mathsf{real}, '\,l, '\,c)\,\mathsf{mcon}$, and the correspond to connectors of type $\mathbb{R}$ for the input, output, and initial value, respectively. The two type parameters $'l$ and $'c$ represent the local and global state-space for the diagram. The final argument is a boolean connector that is used to reset the block. We then use these arguments to construct a record which describes the initial equation, differential equation, and reset equations for this particular block.

In addition to block definitions, we have also encode the block composition function $\oplus_m$, and the semantic function that produces a hybrid reactive design. These developments can be found in the repository under `modelica/noncomp/`.

9 Conclusion

In this final deliverable for Task 2.3, we have made two main contributions.

Firstly, we have constructed a novel semantic domain in UTP for the representation of hybrid computation, using a collection of UTP theories. These theories include generalised reactive processes, hybrid relations, and reactive designs, we form a layered semantic meta-model. These theories have all been mechanised in Isabelle/UTP, and can be used to perform assisted verifications of hybrid programs.

Secondly, we have used this domain to give a semantics to Modelica. We showed how the core Modelica language can be mapped to a hybrid reactive design, and gave a precise semantics to the event iteration cycle. We also showed how to describe Modelica blocks, and gave a semantics to the flattening process.

Together these contributions provide the necessary foundations for a future theorem prover for Modelica. This will require an automated translation from Modelica to our representation in Isabelle/UTP, and also completion of definitions for each class in the Modelica standard library, both of which we leave as future work. The translation will need to map Modelica arithmetic expressions to Isabelle/HOL expressions, which should be straightforward. It will also need to map Modelica block classes to the 10-tuple representation we have given which requires an underlying knowledge of the equations contributed by each Modelica construct. However, if only block classes from the standard library are needed, then the translation merely needs to instantiate their analogue definitions in Isabelle/UTP.

The generality of our UTP theories also means they can also be applied to other related languages, such as 20-sim and Simulink, which opens the door to integration with these different languages. Simulink, for example, could be supported by giving similar definitions to its block library using our reactive contracts, and composition operators for composing blocks in control diagrams. This would permit formal semantics for multi-models whose constituents are described in a variety of different continuous time notations, and potentially also verification facilities.

Our contract notation also supports verification by refinement, and future work could also consider the development of a contract based specification language for control law diagrams. This language would allow one to abstractly characterise a subsystem as relation between its input and outputs ports, and then use proof tactics to substantiate that a particular diagram fulfils the specification. This would require the development of a bespoke requirement language in order to characterise the desirable behaviour of the continuous variables over time, perhaps drawing on temporal logic. Moreover, this work should also draw formal links with existing frameworks for contract-based design [3, 44].

Finally, our results have been practically applied in the context of WP1 for the buildings and railways case study, which shows how our theoretical foundations add value to the INTO-CPS methodology. More work is needed to turn our work into a realistic theorem prover, such as the development of additional tactics for solving differential equations in Isabelle. This could possibly be supported through integration with computational algebra systems, like Matlab, that include such facilities.

References

[1] A. Armstrong, V. Gomes, and G. Struth. Building program construction and verification tools from algebraic principles. *Formal Aspects of Computing*, 28(2):265–293, 2015.

[2] R.-J. Back and J. Wright. *Refinement Calculus: A Systematic Introduction*. Springer, 1998.

[3] A. Benveniste, B. Caillaud, A. Ferrari, L. Mangeruca, R. Passerone, and C. Sofronis. Multiple viewpoint contract-based specification and design. In *6th Intl. Symp. on Formal Methods for Components and Objects (FMCO)*, volume 5382 of *LNCS*, pages 200–225. Springer, 2007.

[4] 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.

[5] Andrew Butterfield, Pawel Gancarski, and Jim Woodcock. State visibility and communication in unifying theories of programming. *Theoretical Aspects of Software Engineering*, 0:47–54, 2009.

[6] C. Calcagno, P. O'Hearn, and H. Yang. Local action and abstract separation logic. In *LICS*, pages 366–378. IEEE, July 2007.

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

[8] A. Cavalcanti and J. Woodcock. A tutorial introduction to designs in unifying theories of programming. In *Proc. 4th Intl. Conf. on Integrated Formal Methods (IFM)*, volume 2999 of *LNCS*, pages 40–66. Springer, 2004.

[9] 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.

[10] Ana Cavalcanti, Simon Foster, Bernhard Thiele, and Jim Woodcock. Initial semantics of Modelica. Technical report, INTO-CPS Deliverable, D2.2c, December 2016.

[11] Ana Cavalcanti, Simon Foster, Jim Woodcock, and Frank Zeyda. Multi-Model Linking Semantics. Technical report, INTO-CPS Deliverable, D2.3d, December 2017.

[12] E. A. Coddington and N. Levinson. *Theory of Ordinary Differential Equations*. McGraw-Hill, 1955.

[13] T. Coquand. Infinite objects in type theory. In *Proc. 1st Intl. Workshop on Types for Proofs and Programs*, volume 806 of *LNCS*, pages 62–78. Springer, 1993.

[14] Luis Diogo Couto, Pasquale Antonante, Stylianos Basagiannis, Sara Falleni, Hassan Ridouane, Hajer Saada, Erica Zavaglio, and James Baxter. Building Case Study 3, (Confidential). Technical report, INTO-CPS Confidential Deliverable, D1.3d, December 2017.

[15] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[16] J. Foster. *Bidirectional programming languages*. PhD thesis, University of Pennsylvania, 2009.

[17] J. Foster, M. Greenwald, J. Moore, B. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3), May 2007.

[18] S. Foster, A. Cavalcanti, J. Woodcock, and F. Zeyda. Unifying theories of time with generalised reactive processes. *Submitted to Information Processing Letters*, 2017.

[19] 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*, pages 44–64. Springer, June 2016.

[20] S. Foster, F. Zeyda, and J. Woodcock. Isabelle/UTP: A mechanised theory engineering framework. In *UTP*, volume 8963 of *LNCS*, pages 21–41. Springer, 2014.

[21] 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*, pages 295–314. Springer, 2016.

[22] W. Guttman and B. Möller. Normal design algebra. *Journal of Logic and Algebraic Programming*, 79(2):144–173, February 2010.

[23] 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.

[24] 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.

[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] L. Henkin, J. Monk, and A. Tarski. *Cylindric Algebras, Part I.* North-Holland, 1971.

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

[28] T. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[29] T. Hoare and J. He. *Unifying Theories of Programming.* Prentice-Hall, 1998.

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

[31] J. Hölzl. Proving Inequalities over Reals with Computation in Isabelle/HOL. In *Proceedings of ACM SIGSAM PLMMS 2009*, pages 38–45, August 2009.

[32] F. Immler. Formally verified computation of enclosures of solutions of Ordinary Differential Equations. In *Proc. 6th NASA Formal Methods Symposium (NFM)*, volume 8430 of *LNCS*. Springer, 2014.

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

[34] J.-L. Lassez, V. L. Nguyen, and E. A. Sonenberg. Fixed point theorems and semantics: a folk tale. *Information Processing Letters*, 14(3):112–116, May 1982.

[35] Edward A. Lee. Constructive models of discrete and continuous physical phenomena. *IEEE Access*, 2:797–821, August 2014.

[36] H. Lundvall, P. Fritzson, and B. Bachmann. Event handling in the OpenModelica compiler and runtime system. Technical report, Linköping University, 2008.

[37] A. McEwan. *Concurrent Program Development in Circus.* PhD thesis, Oxford University, 2006.

[38] Modelica Association. Modelica – A Unified Object-Oriented Language for Systems Modeling – Version 3.4. Standard Specification, April 2017.

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

[40] M. Oliveira, A. Cavalcanti, and J. Woodcock. A UTP semantics for *Circus*. *Formal Aspects of Computing*, 21:3–32, 2009.

[41] Julien Ouy and Thierry Lecomte. Railways Case Study 2, (Confidential). Technical report, INTO-CPS Confidential Deliverable, D1.2b, December 2016.

[42] Julien Ouy and Thierry Lecomte. Railways Case Study 3, (Confidential). Technical report, INTO-CPS Confidential Deliverable, D1.3b, December 2017.

[43] A. W. Roscoe and C. A. R. Hoare. The laws of Occam programming. *Theoretical Computer Science*, 60(2):177–229, September 1988.

[44] A. Sangiovanni-Vincentelli, W. Damm, and R. Passerone. Taming dr. frankenstein: Contract-based design for cyber-physical systems. *European Journal of Control*, 3:217–238, 2012.

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

[46] D.S. Scott. Continuous Lattices. In *Proceedings of the 1971 Dalhousie Conference*. Springer-Verlag, 1971.

[47] A. Sherif, A. Cavalcanti, J. He, and A. Sampaio. A process algebraic framework for specification and validation of real-time systems. *Formal Aspects of Computing*, 22(2):153–191, 2010.

[48] A. Tarksi. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

[49] A. Tarski. On the calculus of relations. *J. Symbolic Logic*, 6(3):73–89, 1941.

[50] J. Woodcock. The Miracle of reactive programming. In *Proc. 2nd Intl. Symp. on Unifying Theories of Programming (UTP)*, volume 5713 of *LNCS*, pages 202–217. Springer, 2008.

[51] Jim Woodcock. Engineering UToPiA - Formal Semantics for CML. In *FM 2014: Formal Methods*, volume 8442 of *LNCS*, pages 22–41. Springer, 2014.

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

[53] F. Zeyda, J. Ouy, S. Foster, and A. Cavalcanti. Formalised cosimulation models. In *Proc. 1st Workshop on Co-Simulation of Cyber-Physical Systems (CoSim-CPS)*, LNCS. Springer, 2017.

[54] Frank Zeyda, Ana Cavalcanti, Jim Woodcock, and Julien Ouy. SysML Foundations: Case Study. Technical report, INTO-CPS Deliverable, D2.3a, December 2017.

[55] Frank Zeyda, Simon Foster, Ana Cavalcanti, Jim Woodcock, and Julien Ouy. A Mechanised FMI Semantics. Technical report, INTO-CPS Deliverable, D2.3c, December 2017.

[56] N. Zhan, S. Wang, and H. Zhao. *Formal Verification of Simulink/Stateflow Diagrams*. Springer, 2017.

Appendices

A UTP Theory of Generalised Reactive Designs

In this appendix we examine our UTP theory of generalised reactive design contract in more detail. We describe the healthiness conditions, core signature definitions, important

properties of the theories, and additional algebraic laws. We also show how our theory unifies a variety of existing reactive languages, and also imperative specifications. We consider the formalisation of recursion, and show how Kleene's fixed-point [34] theorem can be applied to calculate tail recursive reactive designs. Finally, we detail preliminary results for the formalisation of parallel composition in our UTP theory.

A.1 Healthiness Conditions

It is often incovenient to rely on a specific syntactic form to reason about contracts, and thus as usual in the UTP methodology we will also define healthiness conditions that characterise well-formed contract predicates. This will allow us to obtain a large number of algebraic laws from lattice theory and related domains, and also allow us to reason about contracts without the need for the syntactic form.

We first define functions that allow us to extract the three parts from a reactive design.

Definition A.1 (Pre-, Peri-, and Postcondition Functions)**.**

$$\begin{aligned} pre_{\mathbf{R}}(P) &\triangleq \neg_r P[true, false, false/ok, ok', wait] \\ peri_{\mathbf{R}}(P) &\triangleq P[true, true, false, true/ok, ok', wait, wait'] \\ post_{\mathbf{R}}(P) &\triangleq P[true, true, false, false/ok, ok', wait, wait'] \end{aligned}$$

These three functions variously substitute the observational variables to obtain the respective predicates. The precondition is false when a reactive design was started ($ok \wedge \neg wait$) but it diverged ($\neg ok'$). Thus, to extract the precondition we set ok, ok', and $wait$ to true, false, and false, respectively, and negate the result using reactive negation. The pericondition is true when the reactive design was started, did not diverge (ok'), but has not yet reached its final state ($\neg wait'$), and thus we substitute the variables appropriately. The postcondition is similarly obtained, but $wait'$ becomes false, of course. With these definitions we can prove theorems that allow us to extract the constituent reactive relations.

Definition A.2 (Extracting Pre-, Peri-, and Postconditions)**.**

$$\begin{aligned} pre_{\mathbf{R}}([\, P \vdash Q_1 \mid Q_2 \,]) &= P \\ peri_{\mathbf{R}}([\, P \vdash Q_1 \mid Q_2 \,]) &= P \Rightarrow_r Q_1 \\ post_{\mathbf{R}}([\, P \vdash Q_1 \mid Q_2 \,]) &= P \Rightarrow_r Q_2 \end{aligned}$$

Provided P, Q_1, and Q_2 are all $\mathbf{RR}$ healthy.

The pericondition and postcondition are viewed through the prism of the precondition being satisfied, which is why the implication is present. This is an important assumption of reactive designs. Although it is possible to define behaviour in the peri and postcondition that violates the precondition, once the contract is constructed these behaviours will be pruned, such that the behaviour of a contract which violates is precondition is always the maximally non-deterministic $\mathbf{true}_r$. Next we define the healthiness conditions for our theory.

Definition A.3 (Reactive Designs Healthiness Conditions)**.**

$$\begin{aligned}
\textbf{\textit{RD1}}(P) &\triangleq ok \Rightarrow_r P \\
\textbf{\textit{RD2}}(P) &\triangleq P \,;\, \textbf{\textit{J}} \\
\textbf{\textit{RD3}}(P) &\triangleq P \,;\, \mathit{II}_{\textbf{\textit{R}}} \\
\textbf{\textit{SRD}}(P) &\triangleq \textbf{\textit{RD1}} \circ \textbf{\textit{RD2}} \circ \textbf{\textit{R}}_s \\
\textbf{\textit{NSRD}}(P) &\triangleq \textbf{\textit{RD1}} \circ \textbf{\textit{RD3}} \circ \textbf{\textit{R}}_s
\end{aligned}$$

RD1 – ***RD3*** are analogous to ***H1*** – ***H3*** from the theory of designs. Moreover, our ***RD1*** and ***RD2*** correspond to ***CSP1*** and ***CSP2*** from the theories of CSP [29, 9] and *Circus* [40]. We rename them, firstly because our theory has a different alphabet founded by the trace algebra, and secondly because we will not here specify the corresponding ***CSP3*** and ***CSP4*** which constrain *ref* and are thus specific to CSP and *Circus*.

RD1 states, like ***H1***, that observations are possible only after initiation indicated by ok. However, unlike ***H1*** if ok is false the resulting predicate is not ***true***, but $\textbf{\textit{true}}_r$, that is the trace must monotonically increase, but the behaviour is otherwise unpredictable. ***RD2*** is identical to ***H2*** and thus also ***CSP2***. ***RD3*** is analogous to ***H3***: it requires that skip is a right unit which ensures (1) that the precondition is a reactive condition; and (2) the pericondition does not depend on st'.

Our overall healthiness condition for stateful reactive designs is then called ***SRD***, which includes ***RD1***, ***RD2***, and $\textbf{\textit{R}}_s$. The class of ***SRD*** predicates admits a number of useful identities that we outline below.

Theorem A.1 (***SRD*** Laws)**.** *If P is* ***SRD*** *healthy then*

$$\mathit{II}_{\textbf{\textit{R}}} \,;\, P = P \tag{A.1.1}$$

$$\textbf{\textit{Chaos}} \,;\, P = \textbf{\textit{Chaos}} \tag{A.1.2}$$

$$\textbf{\textit{Miracle}} \,;\, P = \textbf{\textit{Miracle}} \tag{A.1.3}$$

More important for our theory of contracts, we identify the class of normal stateful reactive designs, ***NSRD***, which is the theory domain of our reactive contracts. ***NSRD*** does not explicitly invoke ***RD2*** as it is subsumed by ***RD3***, as the following theorem demonstrates.

Theorem A.2 (***RD3*** subsumes ***RD2***)**.**

$$\textbf{\textit{RD2}}(\textbf{\textit{RD3}}(P)) = \textbf{\textit{RD3}}(\textbf{\textit{RD2}}(P)) = \textbf{\textit{RD3}}(P)$$

Proof. This follows since $\textbf{\textit{J}} \,;\, \mathit{II}_{\textbf{\textit{R}}} = \mathit{II}_{\textbf{\textit{R}}} \,;\, \textbf{\textit{J}} = \mathit{II}_{\textbf{\textit{R}}}$. See also mechanisation. □ □

Consequently, it is easy to show that every ***NSRD*** healthy predicate is also ***SRD*** healthy. We can next prove that these healthiness conditions admit certain syntactic formulations of reactive design triples as shown in the following theorem.

Theorem A.3 (Reactive Design Formulations)**.**

$$\textbf{\textit{SRD}}(P) = [\, pre_{\textbf{\textit{R}}}(P) \vdash peri_{\textbf{\textit{R}}}(P) \mid post_{\textbf{\textit{R}}}(P) \,] \tag{A.3.1}$$

$$\textbf{\textit{NSRD}}(P) = [\, \textbf{\textit{RC1}}(pre_{\textbf{\textit{R}}}(P)) \vdash (\exists\, st \bullet peri_{\textbf{\textit{R}}}(P)) \mid post_{\textbf{\textit{R}}}(P) \,] \tag{A.3.2}$$

If a predicate is ***SRD*** healthy then it takes the typical form of a stateful reactive design with a pre, peri, and postcondition. Theorem A.3.1 thus allows us to take an ***SRD*** predicate and deconstruct it into its three parts which can then be manipulated as separate entities. Consequently, a well-formed reactive design triple yields a healthy reactive design.

Theorem A.4 (Reactive Contracts are ***SRD*** healthy)**.** *If* P_1*,* P_2*, and* P_3 *are all* ***RR*** *healthy then* $[\, P_1 \vdash P_2 \mid P_3 \,]$ *is* ***SRD*** *healthy.*

NSRD predicates take a similar, albeit more restrictive form to those of ***SRD***. Firstly, st' must not be mentioned in the pericondition, as illustrated by the existential quantifier. This follows as a direct consequence of $\textbf{R3}_h$ [5], which requires that st must not be restricted in an intermediate state. Thus, composition of P with $\Pi_{\mathbf{R}}$ to the right in ***RD3*** has the same effect on st' in P since $\Pi_{\mathbf{R}}$ ignores any intermediate states ($wait' = true$) rather than passing them on to any successor.

Having st' effectively invisible in intermediate observations has a number of advantages. It ensures that operators like external choice which conjoin intermediate observations will not yield miraculous behaviour caused by inconsistent intermediate state updates [5]. It also means that state updates can be deferred into the future as far as possible, with them ultimately only being observable by a direct sequential successor, or when an event reveals them. This, for instance, allows the following algebraic law for *Circus*.

Theorem A.5 (Assignment and Events)**.**

$$(x :=_c e \,;\, c \to P) \;=\; (c[e/x] \to (x :=_c e \,;\, P))$$

Theorem A.5 allows us to push assignment through event prefix, whilst making an appropriate substitution. It can be found in reactive languages like Occam [43]. This law does not hold in previous *Circus* semantics [40] as st' (there called v') is revealed in intermediate states, and thus whilst the left hand side of this equation admits an intermediate observation where x is updated to e, the right hand side does not.

A side effect of ***RD3*** is to prevent encoding of McEwan's interruption operator [37] that retains the state following interruption [5], as no intermediate state variables can be observed. If such an operator is required then the loss of Theorem A.5 must be accepted, and ***R3*** used as the base of reactive designs instead. This is a design choice depending on the kind of reasoning and expressivity needed. The latter healthiness condition is supported in Isabelle/UTP if its use is desired.

The second restriction of ***NSRD*** is that the precondition must take a specific form of a reactive condition (***RC***): $\textbf{true}_r$ must be a right unit of the negated precondition. Indeed, it is this identity which motivated our definition of the reactive condition healthiness condition. Unlike ***H3*** designs, the precondition need not only refer to undashed variables, as it can refer to tr' in certain circumstances. However, all other primed variables such as ok' and $wait'$, for example, cannot be present in the precondition of an ***NSRD*** predicate.

The precondition restriction follows as a direct consequence of ***RD1***. Consider $P \,;\, \Pi_{\mathbf{R}}$: if P diverges and ok' takes the value false, and the behaviour of P is $\neg_r \, pre_{\mathbf{R}}(P)$ since the precondition must have been violated. If ok' is false in P then the behaviour of its

successor $\mathit{II}_{\mathbf{R}}$ is simply $\mathbf{true}_r$, due to ***RD1***. Thus ***RD3*** yields the identity $(\neg pre_{\mathbf{R}}(P))$; $\mathbf{true}_r = (\neg pre_{\mathbf{R}}(P))$ for divergent behaviours, which if the motivation for ***RC1***.

The relaxation of this restriction allows us to express preconditions which constrain the possible behaviour of the environment that the process is willing to accept. Having a reactive condition as the precondition means that the negated precondition cannot constrain the whole future of the trace, only an initial segment. The trace part of the precondition therefore characterises behaviours that the environment must not engage in.

This form of precondition used by Example 5.4 in Section 5, where the environment should not perform an a to ensure correct behaviour. Here the trace requirement is to the left of an implication which means it is effectively negated. All of these forms are ***RC*** healthy. We can finally state the following introduction theorem as a corollary of Theorem A.3.2, which give the conditions under which a predicate is ***NSRD*** healthy.

Theorem A.6 (***NSRD*** Introduction)**.** *Predicate P is **NSRD** healthy provided that the following conditions hold:*

1. *P is **SRD** healthy;*
2. *$pre_{\mathbf{R}}(P)$ is **RC** healthy;*
3. *$peri_{\mathbf{R}}(P)$ does not mention st'.*

SRD and ***NSRD*** are both idempotent and continuous, and therefore both form a complete lattice, as stated in the following theorem.

Theorem A.7 (Reactive Design Lattices)**.** ***SRD*** *and* ***NSRD*** *healthy predicates form complete lattices with* $\top_{\mathbf{R}} \triangleq \mathbf{SRD}(\mathbf{false}) = \mathbf{Miracle}$ *and* $\bot_{\mathbf{R}} \triangleq \mathbf{SRD}(\mathbf{true}) = \mathbf{Chaos}$.

Proof. Standard proofs of idempotency for ***CSP1*** and ***CSP2*** [29, 9] apply also to ***SRD1*** and ***SRD2***, respectively. ***SRD3*** is idempotent since $\mathit{II}_{\mathbf{R}}$; $\mathit{II}_{\mathbf{R}} = \mathit{II}_{\mathbf{R}}$. ***SRD1*** is continuous since it is disjunctive. ***SRD2*** and ***SRD3*** are both continuous since sequential composition distributes through infima to the left and right. □ □

We note that both ***Miracle*** and ***Chaos***, following the form given in Theorem A.6, are both also ***NSRD*** healthy. We thus obtain weakest fixed-point operators $\mu_{\mathbf{R}}$ and $\mu_{\mathbf{N}}$ for the two theories, which we will further explore in Section A.4. Since ***SRD*** and ***NSRD*** are also continuous, by Theorem 2.1 we can rewrite the weakest fixed-points to $\mu\, X \bullet F(\mathbf{SRD}(X))$ and $\mu\, X \bullet F(\mathbf{NSRD}(X))$, respectively. Thus, we can reason about recursive reactive designs using the relational calculus lattice rather than the theory specific ones. We can also show that reactive designs are closed under the standard relational calculus operators, as the following theorems demonstrate.

Theorem A.8 (Reactive design composition)**.** *If P and Q are **NSRD**-healthy then*

$$\begin{aligned} P\ ;\ Q = \mathbf{R}_s(\ & pre_{\mathbf{R}}(P) \wedge (post_{\mathbf{R}}(P)\ \mathbf{wp}_r\ pre_{\mathbf{R}}(Q)) \\ \vdash\ & peri_{\mathbf{R}}(P) \vee (post_{\mathbf{R}}(P)\ ;\ peri_{\mathbf{R}}(Q)) \\ \diamond\ & post_{\mathbf{R}}(P)\ ;\ post_{\mathbf{R}}(Q) \end{aligned}$$

Theorem A.8 is essentially the same as Theorem 5.2.5, but relies on healthiness of P and Q, rather than their syntactic form. As similar law holds for ***SRD*** healthy predicates, though with a more complex precondition. We finally prove closure of the theory under the main programming operators.

Theorem A.9 (Reactive Designs Closure)**.** ***SRD*** *and* ***NSRD*** *healthy predicates are closed under* $\sqcap$, $\sqcup$, $\lhd\, b \,\rhd$, $;$, $\Pi_{\mathbf{R}}$, $\langle\sigma\rangle_{\mathbf{R}}$, ***Miracle***, *and* ***Chaos***

This closure theorem also means that we can import the algebraic laws for the relational calculus operators from core UTP. Finally, we note that our theory admits the following familiar laws from relational calculus for assignment.

Theorem A.10 (Reactive Design Assignment Compositions)**.** *If* P *is* ***NSRD*** *healthy,* x *is a state variable, and* v *is an expression containing only state variables, then:*

$$(x :=_{\mathbf{R}} v) \;;\; P = P[v/x]$$
$$\langle\sigma\rangle_{\mathbf{R}} \;;\; P = \sigma \dagger P$$

The more usual former law is simply an instance of the latter when $\sigma = \{x \mapsto v\}$, crucially provided that x is a state variable and not an arbitrary UTP variable. Theorem A.10 depends directly on ***R3***$_h$ and would not hold if P were constructed from ***R3*** instead. Indeed, if ***R3*** is used then $P[v/x]$ is not even a healthy reactive design.

The reason for this is that in ***R3***$_h$ if $wait = true$ then st is abstracted. Consequently, a substitution for any state variable when P is waiting for its successor is ineffectual: the result will always be $(\exists\, st \bullet \Pi)$. But this is not the case for ***R3*** which instead would yield $\Pi[v/x]$, and thus reveals the state variable in the intermediate state. Since ***R3*** expects that the behaviour when $wait = true$ is simply Π, which is of course different to $\Pi[v/x]$, then state substitution yields an unhealthy predicate in its presence.

Hence, with ***R3***$_h$ we obtain a more abstract way of dealing with state variables in sequential processes. In particular, a state substitution $\sigma \dagger [\, P_1 \vdash P_2 \mid P_3 \,]$ yields the healthy reactive design $[\, \sigma \dagger P_1 \vdash \sigma \dagger P_2 \mid \sigma \dagger P_3 \,]$ as expected, and thus substantiates Law RA2 of Theorem 5.5.

A.2 Unifying Reactive Languages

NSRD healthy predicates encompass a number of existing languages, notably *Circus* and *CML*, and thus acts as a unifying layer for stateful reactive contracts. We will show that these UTP theories are all subsets of ***NSRD***, and thus all the laws we have proved so far are applicable. This section thus serves to demonstrate the breadth of application of our UTP theory.

We first consider CSP and *Circus*. In addition to the reactive design healthiness conditions, ***CSP1*** and ***CSP2***, the UTP CSP semantics adds ***CSP3*** and ***CSP4***.

Definition A.4 (CSP Healthiness Conditions)**.**

$$\mathbf{CSP3}(P) \triangleq SKIP \;;\; P \qquad \mathbf{CSP4}(P) \triangleq P \;;\; SKIP$$

The operator *SKIP* is equivalent to the **Skip** operator, but without state variables as CSP does not have these. CSP can be obtained by simply setting the state-space type Σ to a singleton set, which effectively removes state. In this case, $\mathbf{R3}_h$ degenerates to **R3**, and thus $\mathbf{R}_s$ degenerates to **R**. Cavalcanti and Woodcock [9] proved that *SKIP* is equivalent to the reactive design $\mathbf{R}(\mathbf{true} \vdash tr' = tr \wedge wait')$, which in turn is equivalent to $[\, \mathbf{true}_r \mid \mathbf{false} \mid \mathbf{tt} = \langle\rangle \,]$. We can then prove the following theorem.

Theorem A.11 (CSP processes are **NSRD** healthy)**.** *If P is* **R** *and* **CSP1**-**CSP4** *healthy, then P is* **NSRD** *healthy*

Proof. Since **SRD** is equivalent to $\mathbf{R} - \mathbf{CSP1} - \mathbf{CSP2}$ is suffices to show that P is **RD3** healthy. From **CSP4** we know that $P = P \,;\, SKIP$. Moreover, we know that *SKIP* is **RD3** since it satisfies the form given in Theorem A.3.2: the precondition is clearly **RC1**, and the pericondition does not mention st'. Consequently, $SKIP \,;\, \Pi_{\mathbf{R}} = SKIP$, and thus by associativity of ; it follows that P is **NSRD**. □ □

Circus actions are similar to CSP processes, but have slightly different healthiness conditions that do account for state variables. Effectively, they require that the *Circus* action **Skip** is a left and right unit. Then, following a similar proof to Theorem A.11, we can show that *Circus* actions are indeed **NSRD** healthy. *Circus* additionally has the healthiness condition **C3** [40] that mandates that the precondition of a reactive design can only contain undashed variables. Though true for *Circus* processes, this is over-restrictive and prevents assumptions with trace behaviour in reactive contracts. Thus our model is both unifying and more general, in that we only mandate the the precondition is **RC1**.

We next show that our model also encompasses *CML*, a formal discrete time modelling language for systems of systems. Sequential *CML* actions are characterised by six healthiness conditions that we enumerate below.

Definition A.5 (*CML* Healthiness Conditions)**.**

$$\begin{aligned}
\mathbf{RT1}(P) &\triangleq rt \leq rt' \wedge P[rt' - rt, \langle\rangle / rt', rt] \\
\mathbf{RT2}(P) &\triangleq \mathbf{RT1}(\mathbf{true}) \lhd \neg ok \rhd P \\
\mathbf{RT3}(P) &\triangleq (\exists\, v' \bullet \Pi) \lhd wait \wedge ok \rhd P \\
\mathbf{RT4}(P) &\triangleq P \,;\, \mathbf{J} \\
\mathbf{RT5}(P) &\triangleq \mathbf{SKIP} \,;\, P \\
\mathbf{RT6}(P) &\triangleq P \,;\, \mathbf{SKIP} \\
\mathbf{SKIP} &\triangleq \mathbf{RT2} \circ \mathbf{RT3}(\neg wait' \wedge rt' = rt \wedge v' = v \wedge ok')
\end{aligned}$$

Observational variables $rt, rt' : \mathrm{seq}\,(\mathcal{E} + \mathbb{P}(\mathcal{E}))$ represent the timed trace before and after execution. They correspond to tr and tr' from the theory of CSP, but in addition to events $\mathcal{E}$, a timed trace can contain "tock" events that denote the passage of time. Thus a trace is divided into a sequence of time instants, each of which is characterised by a sequence of effectively instantaneous events. A tock event is represented by the set of events that were refused at the end of a particular instant. Thus, *CML* does not have require the observational variable ref, as refusals are encoded in the trace itself. Since

our notion of trace is polymorphic, we will replace rt by tr with an appropriate event type.

The *CML* **SKIP** is different, again, to the CSP *SKIP*. However, it is in fact equivalent to the reactive skip operator $\mathit{II}_{\mathbf{R}}$, precondition true, a pericondition false (since $wait' = false$), and the postcondition simply equates each undashed variable with its dashed version, including the state st. Thus, we can prove the following theorem.

Theorem A.12 (*CML* processes are **NSRD** healthy)**.** *If P is* **RT1**-**RT6** *healthy, then P is* **NSRD** *healthy*

Proof. **RT1**-**RT2** is equivalent to $\mathbf{R}_s$. **RT3** and **RT4** are equivalent to **RD1** and **RD2**, respectively. Since **SKIP** = **Skip**, **RT6** is a theorem for **SRD** predicates. Finally, **RT7** is equivalent to **SRD3** so we are done. □ □

Thus, our theory unifies and subsumes both untimed *Circus* and timed *CML*, which demonstrates its applicability. All the laws we have proved for **NSRD** predicates can be used for reasoning about both languages, along with other members of the *Circus* language family with similarly formulated healthiness conditions.

Finally, our theory can also be used to unify languages with hybrid computation. As we have previously shown [18], our trace algebra allows us to account for continuous timed traces [25], which can in turn be applied to model continuous variables and differential equations [19]. The inclusion of continuous variables thus allows us to consider hybrid reactive design contracts, where the assumptions and guarantees constrain the possible behaviours of the continuous variables. Such a foundation has previously been applied to give a UTP semantics to Simulink [56], and our contracts can be applied similarly.

A.3 Linking with Imperative Specifications

Aside from unifying reactive languages, our contracts also allow us to link with imperative specification languages like VDM and Z. In such notations, systems are specified in terms of operations that are formulated as relations on state variables. The theory of designs can be applied to unifying such specification formalisms. Thus, in this section we formulate a link between the theory of designs and reactive designs, which enables embedding imperative programs and specifications into reactive contracts.

We specify functions for converting designs to reactive designs, and vice-versa.

Definition A.6 (Lifting Functions)**.**

$$\mathbf{R}_D(P) \triangleq \mathbf{R}_s(pre_{\mathbf{D}}(P) \vdash false \diamond (\mathbf{tt} = \langle\rangle \wedge post_{\mathbf{D}}(P))$$
$$\mathbf{D}_R(Q) \triangleq (pre_{\mathbf{R}}(Q))[tr'/tr] \vdash (post_{\mathbf{R}}(Q))[tr'/tr]$$

Here, P is a normal UTP design on state variables alone: its alphabet consists of ok, ok', st, and st' and it is **N** healthy. Q is any reactive design predicate. The function $\mathbf{R}_D$ embeds a state variable design into a reactive design by copying the precondition, setting the pericondition to **false**, since there are no intermediate observations, and conjoining the postcondition with $\mathbf{tt} = \langle\rangle$ so that no events occur. If P is a normal design, then it

follows that $\boldsymbol{R}_D(P)$ is a normal stateful reactive design, in particular since the precondition will consider only unprimed state variables and not the trace.

We also introduce the function $\boldsymbol{D}_R$ as its inverse: it constructs a normal design which ignores the pericondition and permits no trace behaviour by substituting tr for tr'. We can show that, for any normal design $p \vdash Q$, $\boldsymbol{D}_R$ is the inverse of $\boldsymbol{R}_D$.

Theorem A.13 (Lifting Theorems)**.**

$$\boldsymbol{D}_R(\boldsymbol{R}_D(p \vdash Q)) = p \vdash Q \tag{A.13.1}$$

$$P \sqsubseteq Q \Rightarrow \boldsymbol{R}_D(P) \sqsubseteq \boldsymbol{R}_D(Q) \tag{A.13.2}$$

A corollary of this theorem is that $\boldsymbol{R}_D$ is an injective function: every normal design has a corresponding unique reactive design. The $\boldsymbol{R}_D$ is also monotonic, meaning refinement of a lifted imperative design can be determined by refinement of the design itself. Finally, we demonstrate some homomorphism laws of $\boldsymbol{R}_D$.

Theorem A.14 (Lifting Homomorphisms)**.**

$$\boldsymbol{R}_D(\top_D) = \textbf{\textit{Miracle}}$$

$$\boldsymbol{R}_D(\bot_D) = \textbf{\textit{Chaos}}$$

$$\boldsymbol{R}_D(\text{I\!I}_{\boldsymbol{D}}) = \text{I\!I}_{\boldsymbol{R}}$$

$$\boldsymbol{R}_D(P \sqcap Q) = \boldsymbol{R}_D(P) \sqcap \boldsymbol{R}_D(Q)$$

$$\boldsymbol{R}_D(P \lhd b \rhd Q) = \boldsymbol{R}_D(P) \lhd b \rhd \boldsymbol{R}_D(Q)$$

$$\boldsymbol{R}_D(P \mathbin{;} Q) = \boldsymbol{R}_D(P) \mathbin{;} \boldsymbol{R}_D(Q)$$

$$\boldsymbol{R}_D(\langle\sigma\rangle_{\boldsymbol{D}}) = \langle\sigma\rangle_{\boldsymbol{R}}$$

Lifting the design top and bottom yields the reactive design top and bottom, as expected, and lifting the design identity yields the reactive identity. Moreover, $\boldsymbol{R}_D$ as expected distributes through internal choice, conditional, and sequential composition. Finally, lifting the design assignment yields the reactive assignment.

A.4 Recursion

In this section we will show how to calculate reactive contracts for a restricted class of recursive models, with the particular aim of substantiating Theorem 5.3 in Section 5. In Section A.1 we showed that generalised reactive designs form a complete lattice. Thus for any monotonic process construction F we can be sure there exists fixed-points $\mu\, F$ and νF. However, in order to reason about reactive contract generally, we need to calculate the pre, peri, and postconditions of such constructions.

In general, we are most interested in the weakest fixed-point for reactive designs, $\mu\, F$, as the strongest fixed-point yields miraculous behaviour for erroneous processes [8]. For example, $(\nu X \bullet X) = \textbf{\textit{Miracle}}$, whereas in reality an infinite loop is a programmer error that should yield ***Chaos***, which $\mu\, X \bullet X$ does. In order to calculate the reactive design of a weakest fixed-point we employ two results: (1) Hoare and He's proof that guarded

processes yield unique fixed-points [29, theorem 8.1.13, page 206], and (2) Kleene's fixed-point theorem [34]. The latter allows us to convert from a recursive construction with a strongest fixed-point to an iterative construction, using a replicated internal choice of power constructions. Since we can calculate the reactive design of replicated processes, we can therefore tackle recursion.

Hoare and He's theorem states, informally, that guarded reactive processes yield a unique fixed-point. That is, if for any X, $F(X)$ is guarded then $\mu\, F = \nu F$. Guardedness is defined as follows.

Definition A.7 (Guarded Reactive Designs)**.** A fixed-point function on reactive designs $F : [\![\textbf{\textit{SRD}}]\!] \to [\![\textbf{\textit{SRD}}]\!]$ is guarded provided that, for any $P \in [\![\textbf{\textit{SRD}}]\!]$ and $n \in \mathbb{N}$,

$$(F(P) \wedge gv(n+1)) = (F(P \wedge gv(n)) \wedge gv(n+1))$$

where $gv(n) \triangleq (tr \leq tr' \wedge \#tt < n)$ and $\#tt : \mathbb{N}$ is a suitable discrete trace measure.

Given a reactive design $\mu\, X \bullet F(X)$, if F is guarded then intuitively before recursion variable X can be reached, F must have produced a non-empty portion of the trace. For example, in CSP we may have a process $\mu\, X \bullet a \to X$ which is guarded since it must perform an a before recursing. This is ensured by requring that the trace contribution of the function applied to a reactive design $F(P)$ yields a trace strictly longer than that produced by P. This is the purpose of gv: if we observe $F(P)$ in a context where the trace is longer than $n+1$ (enforced by $gv(n+1)$) then we can conclude that the trace contributed by P must be no longer than n, and thus we can conjoin it with $gv(n)$. We can then use this to prove Hoare and He's theorem.

Theorem A.15 (Unique Fixed-Points)**.** *If F is guarded then $\mu\, F = \nu F$.*

Technically, the μ and ν operators here specified are the operators of the relational calculus lattice, and not an arbitrary UTP theory. Thus, in order to employ this theorem in the context of reactive designs we first have to employ Theorems A.7 and 2.1 to convert the reactive design fixed-point operator. Continuity of ***SRD*** is thus an important property. Thus, for any guarded process, we can convert a recursive construction using the weakest fixed-point to one using the strongest fixed-point.

In order to make use of Theorem A.15 it is necessary to prove guardedness theorems for the operators of the target language. In general, this can be quite complicated and so for the purposes of this paper we shortcut guardness and instead focus on tail recursive fixed-point constructions of the form $\mu_R\, X \bullet P \fatsemi X$, where X is not mentioned in P, as employed by Theorem 5.3. This pattern, though restrictive, covers a large number of specifiable *Circus* processes, for instance. In this case, guardedness can be shown simply by showing that P always produces events before it terminates. Of course P may not terminate at all, but in this case the recursion variable X is unreachable and thus $\mu_R\, X \bullet P \fatsemi X$ would reduce to P. Whether or not P terminates, we define productivity as the criterion we need for well-defined recursion.

Definition A.8 (Productivity)**.** A reactive design $[\, P \vdash Q \mid R \,]$ is said to be productive if R is a fixed-point of $\lambda\, X \bullet X \wedge tr < tr'$. That is, if we establish termination then it is necessary that the trace strictly increases.

We can then prove the following theorem.

Theorem A.16. *If P is productive then $\lambda\, X \bullet P \mathbin{;} X$ is guarded.*

Then, by Theorem A.15 we know it is safe to map the weakest fixed-point to the strongest fixed-point. This then brings us to Kleene's fixed-point theorem, which allows us to calculate an iterative construction for a strongest fixed-point of F, provided that F is continuous.

Theorem A.17 (Adapted Kleene's fixed-point theorem)**.** *If F is a continuous function then the strongest fixed-point can be calculated by iteration:*

$$\nu F \;=\; \prod_{i \in \mathbb{N}} F^i(\textbf{\textit{false}})$$

Kleene's fixed-point theorem actually often employs Scott-continuity as its antecedent, which is a weaker notion than our continuity on complete lattices. The theorem allows us to calculate the fixed-point by iterating F, starting from ***false***, the lattice $\top$. Now, for our simplified pattern $\nu X \bullet P \mathbin{;} X$ we automatically have continuity since relational composition is continuous. Combining this with Theorem 5.2.6, that includes the calculation for a power construction, we are now in the position to substantiate Theorem 5.3.

Proof.

$$\begin{aligned}
&\mu_R\, X \bullet [\, P \vdash Q \mid R \,] \mathbin{;} X \\
&= \mu\, X \bullet [\, P \vdash Q \mid R \,] \mathbin{;} \textbf{\textit{SRD}}(X) && \text{[Theorem 2.1.8]} \\
&= \nu X \bullet [\, P \vdash Q \mid R \,] \mathbin{;} \textbf{\textit{SRD}}(X) && \text{[Theorems A.15 and A.16]} \\
&= \prod_{i \in \mathbb{N}} (\lambda\, X \bullet [\, P \vdash Q \mid R \,] \mathbin{;} \textbf{\textit{SRD}}(X))^i(\textbf{\textit{false}}) && \text{[Theorem A.17]} \\
&= \prod_{i \in \mathbb{N}} (\lambda\, X \bullet [\, P \vdash Q \mid R \,] \mathbin{;} \textbf{\textit{SRD}}(X))^{i+1}(\textbf{\textit{false}}) && \text{[Unfold: } f^0(\textbf{\textit{false}}) = \textbf{\textit{false}}\text{]} \\
&= \prod_{i \in \mathbb{N}} ([\, P \vdash Q \mid R \,]^{i+1} \mathbin{;} \textbf{\textit{SRD}}(\textbf{\textit{false}})) && \text{[Induction on } i\text{]} \\
&= \prod_{i \in \mathbb{N}} ([\, P \vdash Q \mid R \,]^{i+1} \mathbin{;} \textbf{\textit{Miracle}}) && \text{[Theorem 2.1.5]} \\
&= \prod_{i \in \mathbb{N}} \left[\bigwedge_{i \le n} (R^i \;\textbf{\textit{wp}}_r\; P) \;\middle|\; \bigvee_{i \le n} R^i \mathbin{;} Q \;\middle|\; R^{n+1} \right] \mathbin{;} \textbf{\textit{Miracle}} && \text{[Theorem 5.2.6]} \\
&= \prod_{i \in \mathbb{N}} \left[\bigwedge_{i \le n} (R^i \;\textbf{\textit{wp}}_r\; P) \;\middle|\; \bigvee_{i \le n} R^i \mathbin{;} Q \;\middle|\; \textbf{\textit{false}} \right] && \text{[Theorem RD8]} \\
&= \left[\bigwedge_{i \in \mathbb{N}} (R^i \;\textbf{\textit{wp}}_r\; P) \;\middle|\; \bigvee_{i \in \mathbb{N}} R^i \mathbin{;} Q \;\middle|\; \textbf{\textit{false}} \right] && \left[\begin{array}{c}\text{Theorem 5.2.2 and} \\ \text{relational calculus}\end{array}\right]
\end{aligned}$$

□ □

This proof demonstrates the necessity of all the body of theorems we have proved from the UTP theories and reactive designs in order to reason about recursion. We therefore now have a complete constructive approach for calculating the reactive contract for a variety of recursive specifications.