Unifying Theories of Reactive Design Contracts

Simon Foster, Ana Cavalcanti, Samuel Canham, Jim Woodcock and Frank Zeyda

Abstract

Design-by-contract is an important technique for model-based design in which a composite system is specified by a collection of contracts that specify the behavioural assumptions and guarantees of each component. In this paper, we describe a unifying theory for *reactive design* contracts that provides the basis for modelling and verification of reactive systems. We provide a language for expression and composition of contracts that is supported by a rich calculational theory. In contrast with other semantic models in the literature, our theory of contracts allow us to specify both the evolution of state variables and the permissible interactions with the environment. Moreover, our model of interaction is abstract, and supports, for instance, discrete time, continuous time, and hybrid computational models. Being based in Unifying Theories of Programming (UTP), our theory can be composed with further computational theories to support semantics for multi-paradigm languages. Practical reasoning support is provided via our proof framework, Isabelle/UTP, including a proof tactic that reduces a conjecture about a reactive program to three predicates, symbolically characterising its assumptions and guarantees about intermediate and final observations. This allows us to verify programs with a large or infinite state space. Our work advances the state-of-the-art in semantics for reactive languages, description of their contractual specifications, and compositional verification.

1. Introduction

Verification of large-scale systems of systems and cyber-physical systems is challenging due to the size and complexity of the underlying models [50]. The design-by-contract paradigm [42, 36, 3] provides a precise approach for compositional verification. In this approach, the verifier defines contracts with two parts: (1) the required behaviour that a constituent guarantees to implement, and (2) the assumptions the constituent can make about its environment [4].

A composite system can thus be characterised by a number of contracts, one for each constituent, the composition of which fulfils the overall system-level contract that specifies the behaviour of the system as a whole. Each constituent system can be shown to fulfil its required behaviour under certain assumptions. Violation of a constituent's assumptions leads to unpredictable behaviour of the entire system. To enable verification, we also need a theory of contracts applicable to a wide range of paradigms and accompanied by automated tool support.

In this article, we provide a novel unifying theory of contracts for a wide-spectrum of stateful reactive languages [46, 60, 17], with practical verification support provided in the Isabelle/HOL theorem prover [45, 19]. Our contracts are supported by a rich algebraic theory with composition operators and laws to calculate the overall assumptions and guarantees of composite contracts. Whilst contracts can be applied as a specification mechanism for the purpose of verification, they can also function as a denotational model for reactive languages according to the "programs-as-predicates" philosophy [27, 46]. This means that our contract theory is a semantic model sufficient to characterise both specifications and implementations, and thus avoids a formalisation gap between different languages.

Our theory is based in Hoare and He's Unifying Theories of Programming [33, 10] (UTP): a meta-model framework for describing denotational semantics in terms of an alphabetised predicate calculus that acts as a *lingua franca*. Different semantic models can be encoded into this common domain, and compared and linked with one another. Unlike other works [25, 64], our theory of contracts is wholly embedded into the alphabetised relational calculus: this gives a direct route to automated reasoning. We build on previous

UTP theories of reactive processes [33] and reactive designs [10, 46], whilst making several improvements and generalisations.

Our UTP theory is highly extensible in several ways. Whilst the previous work supports only discrete sequence-based traces, we adopt our work on algebraic trace models [16]. This allows us to describe contracts for a spectrum of languages, from untimed and discrete time languages, through to continuous time and hybrid systems with differential equations. Moreover, our theory can be extended with additional semantic information, such as refusals as employed in *Circus* [61, 46], but also potentially other models such as timed testing traces [60, 17].

Finally, our theory has been mechanised in our Isabelle-based proof assistant for the UTP, which we call Isabelle/UTP [19, 63, 18]. This provides us with theory construction and verification facilities, and also the ability to develop verification tools from UTP-based semantic models. Notably, we demonstrate a prototype proof tactic for proving refinements between reactive contracts by re-expressing refinement conjectures as three implications between the pre-, peri-, and postconditions that are expressed purely in relational calculus. These proof obligations can then be discharged using relational and predicate calculus tactics in Isabelle/HOL, which greatly improves the potential for automation. Moreover, our contract theory allows us to characterise and reason about reactive programs purely symbolically, which allows us to verify programs with a very large or infinite state.

All the theorems proved here are proved in Isabelle/UTP; these proofs can be found in our Isabelle/UTP repository[1]. Additionally, most theorems and definitions in the paper are accompanied by a small Isabelle icon (). In the electronic version, each icon is hyperlinked to the corresponding mechanised artefact in the repository. This, we hope, will convince the reader of the level of rigour employed in this work.

In summary, the novel contributions of this paper are as follows:

1. a novel UTP theory of stateful reactive contracts based on a generalised semantic trace model;
2. a contract notation with assumptions, and guarantees of intermediate and final states;
3. theorems for calculating composite contracts, including parallel composition;
4. an automated proof method for proving contractual refinement conjectures.

The remainder of the paper is organised as follows. In Section 2, we provide the necessary context for our work, including details of UTP, reactive processes, and reactive designs. In Section 3, we describe our theories of reactive relations and conditions, which form an important building block of our contracts. In Section 4, we introduce reactive design contracts; including their notation, algebraic laws, and a number of small illustrative examples. In Section 5, we use the laws described in Section 4 to demonstrate the use of contracts in reasoning about a small case study, which has also been mechanised in Isabelle/UTP. In Section 6, we provide a detailed overview of our UTP theory, including its healthiness conditions, signature, and justification for the laws presented in Section 4. In particular we show how recursion and parallel composition are handled. In Section 7, we give an overview of our mechanisation of reactive designs in Isabelle, highlight idiosyncrasies needed for the encoding, and describe our reactive-design proof tactics. In Section 8, we give a survey of related work on design-by-contract and explain the distinguishing features of our work. Finally, in Section 9 we conclude.

2. Preliminaries

In this section we review preliminaries of UTP and its core theories, and introduce a number of foundational theorems, including novel results regarding the class of continuous healthiness conditions (Theorem 2.5). All theorems in this and following sections have been mechanically proved in Isabelle/UTP [19]; for details please see Section 7 and the accompanying Isabelle proofs. This, we believe, adds a substantially more rigorous basis for UTP and the presented results than the previous works.

[1] Isabelle/UTP Repository: https://github.com/isabelle-utp/utp-main

2.1. UTP

UTP [33] seeks to identify the fundamental computational paradigms that exist as foundations of programming language semantics and formalises them using UTP theories. UTP theories can describe what it means for a language to be concurrent [33, 10], real-time [53], or object-oriented [51]. The UTP thus promotes reuse of theoretical building blocks that underlie programming languages.

UTP is based on an alphabetised relational calculus [33] with operators of higher-order predicate calculus and relation algebra. An alphabetised predicate P consists of a set of typed variables $\alpha(P)$, and a predicate which may refer to only those variables in $\alpha(P)$. Alphabetised relations are alphabetised predicates whose alphabet consists of pairs of variables that denote initial values (x) and later values (x'). The relations are ordered by refinement, $P \sqsubseteq Q$, which is denoted below:

Definition 2.1. $P \sqsubseteq Q \triangleq [Q \Rightarrow P]$ where $\alpha(P) = \alpha(Q)$

Here, $[P]$ denotes the universal closure of P; if $\alpha(P) = \{x_1 \cdots x_n\}$, then $[P] \triangleq (\forall\, x_1 \cdots x_n \bullet P)$. Refinement states that the predicate Q implies P over all variables, and thus P sets an upper bound on the possible observations exhibited by Q. Consequently, $\sqsubseteq$ is a partial order on the set of alphabetised relations.

The domain of alphabetised relations forms a number of important algebraic structures including (1) a complete lattice [33], where the order is refinement ($\sqsubseteq$), ***false*** is top, and ***true*** is bottom; (2) a relation algebra [56, 18]; (3) a cylindric algebra [28, 19]; and (4) a quantale [19], which induces (5) a Kleene algebra [1]. Together these provide a rich set of base properties supporting program verification [1, 19].

UTP follows the "programs-as-predicates" approach [27], where a program is modelled as a relation in the alphabetised predicate calculus. For example, an assignment $x := x + 1$, which increments variable x whilst leaving all other variables unchanged, can be denoted using the predicate $x' = x + 1 \land y' = y$, where y denotes the collection of variables other than x. Programming operators, such as sequential composition ($P \;;\; Q$), if-then-else conditional ($P \lhd b \rhd Q$), non-deterministic choice ($P \sqcap Q$), and recursion ($\mu\, F$), are denoted as predicates [33, 9], as shown below.

Definition 2.2 (UTP Programming Operators)**.**

$$P \;;\; Q \triangleq \exists\, v_0 \bullet P[v_0/v'] \land Q[v_0/v] \qquad \alpha(P) = \alpha(Q) = \{v, v'\}$$

$$x := v \triangleq x' = v \land y' = y \land \cdots \land z' = z$$

$$\mathit{II} \triangleq x := x$$

$$\prod_{i \in I} P(i) \triangleq \bigvee i \in I \bullet P(i)$$

$$P \lhd b \rhd Q \triangleq (b \land P) \lor (\neg b \land Q) \qquad \alpha(P) = \{v, v'\}, \alpha(b) = \{v\}$$

$$\mu\, F \triangleq \bigsqcap \{X \mid F(X) \sqsubseteq X\}$$

Moreover, the empty relation ***false*** is usually used to denote either a program that fails to terminate, or else is "miraculous", having no possible behaviours. This embedding of programs into logic naturally provides great opportunities for verification by automated proof [19]. Moreover, the standard laws of programming [32] are all theorems with respect to the operator denotations. We also emphasise that these operators are all alphabet polymorphic, and can therefore be used to compose predicates of varying types, so long as the side conditions are satisfied. A selection of theorems of Definition 2.2 is shown below.

Theorem 2.3 (Relational Calculus Laws)**.**

$$P \;;\; \mathit{II} = \mathit{II} \;;\; P = P$$

$$P \;;\; \textbf{\textit{false}} = \textbf{\textit{false}} \;;\; P = \textbf{\textit{false}}$$

$$(P \;;\; Q) \;;\; R = P \;;\; (Q \;;\; R)$$

$$(P \lhd b \rhd Q) \;;\; R = P \;;\; R \lhd b \rhd Q \;;\; R$$

$$\left(\prod_{i \in I} P(i)\right) \;;\; Q = \prod_{i \in I} P(i) \;;\; Q$$

$$P \;;\; \left(\prod_{i \in I} Q(i)\right) = \prod_{i \in I} P \;;\; Q(i)$$

The relational skip ($\mathbb{I}$) is a left and right identity, and ***false*** is a left and right annihilator. Sequential composition is associative, it distributes through conditional from the right (but not the left), and it distributes through internal choice from the left and right.

UTP theories are well-defined subsets of the alphabetised relations that satisfy certain properties desirable for a particular computational paradigm. For example, to model real-time programs, we need a way of recording how much time has passed since execution began, and ensuring that the passage of time is well-behaved by, for instance, forbidding reverse time travel. In UTP, this can be achieved by extending our alphabet with special observational variables, such as $clock, clock' : \mathbb{N}$, which can be used to model time, and imposing invariants, such as $clock \leq clock'$. UTP theories are therefore 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***), that specifies the invariants as a function from predicates to predicates with the above alphabet;
3. a *signature*: that is, a set of constructors and operators that are 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 a notion of subtyping exists. This also means that UTP theories can readily be combined by merging the alphabets and composing the healthiness conditions.

If a relation is a fixed-point of the healthiness conditions, $P = \mathbf{HC}(P)$, then it is said to be ***HC***-healthy. For our theory of real-time, we can specify a healthiness condition $\mathbf{HT}(P) \triangleq (P \wedge clock \leq clock')$, which conjoins a relation with the invariant. Then we have it that

$$\mathsf{delay}(n : \mathbb{N}) \triangleq clock' = clock + n$$

is a ***HT***-healthy relation, since it always satisfies $clock \leq clock'$, and therefore is is a fixed-point of ***HT***.

A UTP theory's domain is the set of healthy predicates: $[\![\mathbf{HC}]\!]_{\mathsf{H}} \triangleq \{P \mid \mathbf{HC}(P) = P\}$. For this reason, it is necessary that a healthiness function is idempotent ($\mathbf{HC} \circ \mathbf{HC} = \mathbf{HC}$), and usually also monotonic. Monotonicity ensures that the UTP theory forms a complete lattice, substantiated by the Knaster-Tarski theorem [55]. This gives rise to a theory top ($\top_{\mathbf{T}}$), a bottom ($\bot_{\mathbf{T}}$), an infimum ($\bigsqcap_{\mathbf{T}} A$), for $A \subseteq [\![\mathbf{HC}]\!]_{\mathsf{H}}$, a supremum ($\bigsqcup_{\mathbf{T}} A$), a least fixed-point operator $\mu_{\mathbf{T}} F$, for $F : [\![\mathbf{HC}]\!]_{\mathsf{H}} \to [\![\mathbf{HC}]\!]_{\mathsf{H}}$ and a greatest fixed-point operator $\nu_{\mathbf{T}} F$. 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 of **continuity**, which gives rise to additional properties.

Definition 2.4 (Continuous Healthiness Conditions)**.**
HC is said to be continuous if it satisfies $\mathbf{HC}(\bigsqcap A) = \bigsqcap\{\mathbf{HC}(P) \mid P \in A\}$ for $A \neq \emptyset$.

This notion of continuity, also know as universal disjunctivity, is stronger than the related notion of Scott-continuity [52], 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's infimum ($\bigsqcap_{\mathbf{T}}$) is the same operator as the alphabetised predicate infimum ($\bigsqcap$) for non-empty sets. So, a number of additional laws can be imported into the theory, some of which are illustrated below.

Theorem 2.5 (Continuous Theory Laws)**.**

$$\bot_{\mathbf{T}} = \mathbf{HC}(\mathit{true}) \tag{2.5.1}$$

$$\top_{\mathbf{T}} = \mathbf{HC}(\mathit{false}) \tag{2.5.2}$$

$$\bot_{\mathbf{T}} \sqcap P = \bot_{\mathbf{T}} \quad \textit{if } P \textit{ is } \mathbf{HC}\textit{-healthy} \tag{2.5.3}$$

$$\top_{\mathbf{T}} \sqcap P = P \quad \textit{if } P \textit{ is } \mathbf{HC}\textit{-healthy} \tag{2.5.4}$$

$$\bigsqcap_{\mathbf{T}} A = \bigsqcap A \quad \textit{if } A \neq \emptyset \textit{ and } A \subseteq [\![\mathbf{HC}]\!]_{\mathsf{H}} \tag{2.5.5}$$

$$\mu_{\mathbf{T}} X \bullet F(X) = \mu\, X \bullet F(\mathbf{HC}(X)) \quad \textit{if } F : [\![\mathbf{HC}]\!]_{\mathsf{H}} \to [\![\mathbf{HC}]\!]_{\mathsf{H}} \tag{2.5.6}$$

For the first four these identities, monotonicity of $\mathbf{HC}$ is actually a sufficient assumption. Of particular interest is (2.5.6) that shows how a theory's weakest fixed-point operator ($\mu_{\mathbf{T}}$) can be rewritten to the alphabetised-predicate weakest fixed-point (μ). The requirement is that the continuous healthiness condition $\mathbf{HC}$ can be applied after each unfolding of the fixed-point to ensure that the function F is only ever presented with a healthy predicate. It should be noted that this requirement for continuous healthiness functions does not similarly restrict fixed-point functions. Specifically, our laws apply for any monotonic function F and thus at this level there is no restriction in modelling of unbounded non-determinism. Indeed, we have encountered very few healthiness conditions that are monotonic but not continuous.

Healthiness conditions in UTP are often built by composition of several component functions. That being the case, continuity and idempotence properties of the overall healthiness condition can also be obtained by composition.

Though UTP was originally a purely theoretical framework for denotational semantics [33], more recently it has been adapted into an implementation within the Isabelle proof assistant [45], called Isabelle/UTP [18, 19]. This proof tool can be used develop UTP theories, by defining healthiness conditions and proving algebraic laws, in order to support mechanically verified denotational semantics. Moreover, such a mechanised denotational semantics can be used to construction verification tools for different languages by harnessing Isabelle's powerful automated proof facilities [6]. In this article, we use Isabelle/UTP to both define and verify our UTP theory of reactive contracts, and to produce an associated verification technique.

2.2. Designs

The UTP theory of designs [33, 9] has two observational variables, $ok, ok' : \mathbb{B}$, flags that denote whether a program was started and whether it terminated, respectively. The design, $P_1 \vdash P_2$, states that if a program is started and the state satisfies precondition P_1, then it will terminate and satisfy postcondition P_2. This is encoded in the following predicative definition.

Definition 2.6. $P_1 \vdash P_2 \triangleq (ok \wedge P_1) \Rightarrow (ok' \wedge P_2)$

Here, P_1 and P_2 are relations on variables excluding ok and ok'. Effectively, this encoding allows a pair of predicates to be encoded as a single predicate. An simple example is the design $D \triangleq \mathbf{true} \vdash x' = 1$, where $x : \mathbb{N}$ is a program variable. If D is given permission to execute ($ok = true$), then the program terminates ($ok' = true$) with $x = 1$. If the program is not given permission to execute ($ok = false$), then x can taken any value.

Designs have a natural notion of refinement which requires that the precondition is weakened, and the postcondition strengthened within the window of the precondition, as shown by the theorem below.

Theorem 2.7. $P_1 \vdash P_2 \sqsubseteq Q_1 \vdash Q_2 \Leftrightarrow (P_1 \Rightarrow Q_1) \wedge (Q_2 \wedge P_1 \Rightarrow P_2)$

Design relations are closed under sequential composition, disjunction, and conjunction [33, 9], all of which retain the denotations given in Definition 2.2 with the alphabet containing ok and ok'. The main design healthiness conditions are $\mathbf{H}$ and $\mathbf{N}$, which are given below [33, 9].

Definition 2.8 (Design Healthiness Conditions)**.**

$$\mathbf{H1}(P) \triangleq (ok \Rightarrow P) \qquad \mathbf{J} \triangleq (ok \Rightarrow ok') \wedge v' = v$$
$$\mathbf{H2}(P) \triangleq P \,;\, \mathbf{J} \qquad \mathbb{I}_D \triangleq \mathbf{true} \vdash \mathbb{I}$$
$$\mathbf{H3}(P) \triangleq P \,;\, \mathbb{I}_D \qquad \mathbf{H} \triangleq \mathbf{H1} \circ \mathbf{H2}$$
$$\mathbf{N} \triangleq \mathbf{H1} \circ \mathbf{H3}$$

$\mathbf{H1}$ states that until a design has been given permission to execute, as recorded via ok, no observations are possible. $\mathbf{H2}$ states that no design can require non-termination. A more intuitive characterisation of the $\mathbf{H2}$ fixed-points is $P[false/ok'] \Rightarrow P[true/ok']$: every non-terminating behaviour of P for which $ok' = false$ has an equivalent terminating behaviour for which $ok' = true$. The composition, $\mathbf{H}$, precisely characterises the set of design relations constructed using $P_1 \vdash P_2$ [9].

H3 designs additionally require that P is a condition: it does not refer to dashed variables. This subclass of designs 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* [21], 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 the theories they define are both complete lattices. The bottom element $\bot_D$ is abortive, arising, for instance due to a violated precondition, and the top $\top_D$ is miraculous. The infimum $P \sqcap Q$ is a non-deterministic choice between two designs, and refinement reduces non-determinism: $P \sqcap Q \sqsubseteq P$.

2.3. Reactive Processes

The theory of reactive processes [33, 10] 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 healthiness conditions **R1** and **R2**, and (2) introduce intermediate observations, which is achieved through healthiness condition **R3**. In addition to ok and ok', the theory has three pairs of observational variables:

1. $wait, wait' : \mathbb{B}$ that determine whether a process (or its predecessor) is waiting for interaction with its environment, that is, it is quiescent, or else has correctly terminated;
2. $tr, tr' : \mathrm{seq}\ Event$ that describes the trace before and after the process' execution; and
3. $\mathit{ref}, \mathit{ref}' : \mathbb{P}\ Event$ that describe the events being refused during a quiescent state, as required by the failures-divergences model of CSP [31, 48]. If all events are being refused, the process is in a deadlock situation.

Since reactive programs often run indefinitely, the theory of reactive processes distinguishes good and bad non-termination, the being characterised by divergence. This is achieved by reinterpreting ok to indicate divergence. Specifically, if ok' is false then a reactive process has diverged, meaning it is exhibiting unpredictable or erroneous behaviour. If ok' is true, and $wait'$ is false, the process has not terminated, but neither has it diverged.

The theory of reactive processes is used to provide a UTP denotational semantics for both CSP [32, 10], based on relational encoding of the failures-divergences model [48], and also the stateful process language *Circus* [61, 47, 46]. *Circus* provides all the usual operators of CSP for expressing networks of communicating processes, together with state-based constructs such as variable assignment. *Circus* processes encapsulate a number of state variables, operations that act on those variables, and actions that encode the reactive behaviour of the process using channels.

In previous work [16], we have generalised the standard UTP theory of reactive processes [33]. Our generalised theory [16] removes the ref and ref' variables, which allows us to characterise behavioural semantic models other than failures-divergences. Moreover, we add $\mathit{st}, \mathit{st}' : \Sigma$ to explicitly model state as suggested by [7], where Σ is a state space type. In our previous work [16], 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". We characterise traces with an abstract set $\mathcal{T}$ equipped with two operators: trace concatenation $\frown : \mathcal{T} \to \mathcal{T} \to \mathcal{T}$, and the empty trace $\varepsilon : \mathcal{T}$, which obey the following axioms [16].

Definition 2.9. A trace algebra $(\mathcal{T}, \frown, \varepsilon)$ is a cancellative monoid satisfying the following axioms:

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

An example model is formed by finite sequences, $\langle a, b, \cdots, c\rangle$, that is $(\mathrm{seq}\, A, \frown, \langle\rangle)$ forms a trace algebra, where $\frown$ is sequence concatenation. Using the two trace algebra operators, we can also define a trace prefix operator $(x \leq y)$, and trace difference $(x - y)$, which removes a prefix y from x. 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 [16]. We thus generalise the type of tr and tr' to be an instance of a trace algebra $\mathcal{T}$, and recreate the three reactive healthiness conditions [33, 10].

Definition 2.10 (Stateful Reactive Healthiness Conditions)**.**

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

R1 states that tr is monotonically increasing; processes are not permitted to undo past events. $\mathbf{R2}_c$ is a version of **R2** [33], created to overcome an issue with definedness of sequence difference [16], but semantically equivalent in the context 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 $\mathbf{R2}_c$ deletes the history only when $tr \leq tr'$, which ensures that $\mathbf{R2}_c$ does not depend on **R1**, and thus commutes with it. Intuitively, an **R1**-$\mathbf{R2}_c$ healthy predicate syntactically does not constrain the trace history (tr), but only the trace contribution expression (tt), as the following theorem illustrates.

Theorem 2.11 (**R1**-$\mathbf{R2}_c$ trace contribution)**.**

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

Finally, we have $\mathbf{R3}_h$, a version of **R3** from [7] that introduces the concept of intermediate observations, whilst ensuring that state variables are not included. $\mathbf{R3}_h$ states that if a process observes *wait* to be true, then its predecessor has not yet terminated and thus it should behave like the reactive identity, II_R. For example, in a composition $P\,;\,Q$, if P has not terminated then Q, if $\mathbf{R3}_h$ healthy, will behave as II_R.

The reactive identity maintains the present value of all variables, other than the state st, when the predecessor is in an intermediate state, or behaves like $\mathbf{R1}(\mathit{true})$ if ok is false. The latter scenario means that the predecessor has diverged and thus we can guarantee nothing other than that the trace increases. Intuitively, an $\mathbf{R3}_h$ 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 the theories for both CSP [33, 10] and *Circus* [46].

Definition 2.12 (**R3** Healthiness Condition)**.**

$$\mathbf{R3}(P) \triangleq \mathit{II}_{\mathrm{rea}} \lhd \mathit{wait} \rhd P \qquad \mathit{II}_{\mathrm{rea}} \triangleq \mathit{II} \lhd \mathit{ok} \rhd \mathbf{R1}(\mathit{true})$$

The only difference from $\mathbf{R3}_h$ is that the identity $\mathit{II}_{\mathrm{rea}}$ 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. As discussed in detail in Section 6, $\mathbf{R3}_h$ allows a simpler treatment of state variables, supports additional algebraic laws for assignment and state substitution, and solves the problem with

external choice for which it was originally designed [7], though at the cost of losing McEwan's interruption operator [40]. Thus, the use of $\mathbf{R3}_h$ instead of $\mathbf{R3}$ is a design decision based on the particular modelling facilities of interest.

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

Theorem 2.13 (Reactive Process Theory Properties)**.**

- $\mathbf{R}_s$ *is idempotent:* $\mathbf{R}_s(\mathbf{R}_s(P)) = \mathbf{R}_s(P)$*;*
- $\mathbf{R}_s$ *is 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 2.5. Thus we now have a UTP theory of stateful reactive processes to use as the foundation for reactive design contracts.

3. Reactive Relations and Conditions

In this section, we begin the main novel contributions of our paper, by introducing a theory of reactive relations that we use to describe assumptions and guarantees in our reactive contracts in Section 4. A reactive relation is an $\mathbf{R1}$-$\mathbf{R2}_c$-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 3.1 (Reactive Relations)**.**

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

$\mathbf{RR}$ restricts access to ok and $wait$ through existential quantification. In general, if $(\exists\, x \bullet P) = P$ then it must be the case that P does not refer to x. With the help of Theorem 2.13, we can show that $\mathbf{RR}$ is both idempotent and continuous.

Theorem 3.2. $\mathbf{RR}$ *is idempotent and continuous.*

We can therefore also show that reactive relations form a complete lattice.

Theorem 3.3. $[\![\mathbf{RR}]\!]_H$ *forms a complete lattice, with bottom element* $\mathbf{R1}(\mathbf{true})$*, and top element* $\mathbf{false}$*.*

Proof. We obtain a complete lattice by Knaster-Tarski [28], and by Theorem 2.5 the bottom and top elements are $\mathbf{RR}(\mathbf{true})$ and $\mathbf{RR}(\mathbf{false})$, respectively. For illustration, we give the following calculation that shows how the former reduces to $\mathbf{R1}(\mathbf{true})$.

$$\begin{aligned}
\mathbf{RR}(\mathbf{true}) &= (\exists\, ok, ok', wait, wait' \bullet \mathbf{R1}(\mathbf{R2}_c(\mathbf{true}))) && [3.1]\\
&= (\exists\, ok, ok', wait, wait' \bullet \mathbf{R1}(\mathbf{true}[\varepsilon, tt/tr, tr'] \lhd tr \leq tr' \rhd \mathbf{true})) && [2.10]\\
&= (\exists\, ok, ok', wait, wait' \bullet \mathbf{R1}(\mathbf{true} \lhd tr \leq tr' \rhd \mathbf{true})) && [\text{vacuous substitution}]\\
&= (\exists\, ok, ok', wait, wait' \bullet \mathbf{R1}(\mathbf{true})) && [\text{relational calculus}]\\
&= (\exists\, ok, ok', wait, wait' \bullet tr \leq tr') && [2.10]\\
&= (tr \leq tr') && [\text{predicate calculus}]\\
&= \mathbf{R1}(\mathbf{true}) && [2.10]
\end{aligned}$$

Calculation of $\mathbf{RR}(\mathbf{false}) = \mathbf{false}$ follows a similar form. □

Here, $\mathbf{R1}(\mathbf{true})$ is the most non-deterministic relation where the trace is monotonically increasing. As for relations, $\mathbf{false}$ is miraculous reactive relation with no possible observations.

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 $\mathbf{R1}$ and thus it is necessary to redefine negation, and also implication, for similar reasons, for reactive relations.

Definition 3.4 (Reactive Relation Logical Operators)**.**

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

The universal relation $\mathbf{true}$ is not $\mathbf{RR}$-healthy, since it allows any combination of tr and tr'. Consequently, we define $\mathbf{true}_r$, which is the bottom element. Reactive negation, $\neg_r P$, negates P and then applies $\mathbf{R1}$. Effectively this yields a predicate whose corresponding set of trace extensions does not satisfy P. Since $\mathbf{RR}$ is already closed under the other Boolean operators, such as $\vee$, $\wedge$, and $\mathbf{false}$, we can apply them directly and prove the following theorem.

Theorem 3.5. $(\mathbf{RR}, \wedge, \vee, \neg_r, \mathbf{true}_r, \mathbf{false})$ *forms a Boolean algebra.*

We can also prove the following closure properties for the standard relational operators of Definition 2.2:

Theorem 3.6 (Relational Operators Closure Properties)**.**

- *If P and Q are $\mathbf{RR}$ then $P \,;\, Q$ is $\mathbf{RR}$;*
- *If $I \neq \emptyset$ and $\forall\, i \bullet P(i)$ is $\mathbf{RR}$, then $\bigsqcap_{i \in I} P(i)$ is $\mathbf{RR}$.*

$\mathbf{RR}$ is closed under sequential composition (;) and nondeterministic choice ($\bigsqcap$). Consequently, we can reuse many of the corresponding algebraic laws of the alphabetised relational calculus listed in Section 2.1. This is a significant advantage to the UTP approach of conservatively extending existing theories. However, the relational assignment operator is not healthy, because we can use it to perform arbitrary updates to the trace; $tr := \varepsilon$ is not $\mathbf{R1}$ healthy, for instance. Consequently, we need to define new operators for manipulating and querying a reactive program's state (Σ) via observational variable $st : \Sigma$. We therefore also define the following operators:

Definition 3.7 (Reactive Relational State Operators Assignment)**.**

$$\langle \sigma \rangle_r \triangleq (tr' = tr \wedge st' = \sigma(st) \wedge v' = v)$$
$$\mathit{II}_r \triangleq \langle id \rangle_r$$
$$[s]_r \triangleq \mathbf{R1}(s) \qquad \text{provided } s \text{ refers to undashed state variables only}$$

$\langle \sigma \rangle_r$ is 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. Since the alphabet is open, we use the shorthand v to refer to the variable set excluding ok, $wait$, tr, and st. Substitutions functions can be constructed using the notation $\{x_1 \mapsto v_1, \cdots, x_n \mapsto v_n\}$ which associates n expressions (v_i) to corresponding variables (x_i). A singleton assignment can be denoted as $x :=_r v \triangleq \langle \{x \mapsto v\} \rangle_r$, where v is an expression on undashed state variables.

As usual, we also introduce the degenerate form II_r, which simply retains the values of all variables. We also define a state condition operator $[s]_r$, where s is a predicate over undashed state variables only: it is a condition not mentioning variables ok, $wait$, or tr. The operator requires that s holds on the state variables, whilst leaving the trace unconstrained. We can demonstrate the following healthiness properties for these operators.

Theorem 3.8. $\langle \sigma \rangle_r$, II_r, *and* $[s]_r$ *are* $\mathbf{RR}$*-healthy*

Assignment is **RR**, since we conjoin with $tr' = tr$, which is **R1** and $\mathbf{R2}_c$ healthy, and do not refer to *ok* or *wait*. $\mathbb{I}_r$ is **RR** for the same reasons. The state condition is **RR** healthy as it is clearly **R1**, and also it is $\mathbf{R2}_c$ 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 b is a relation that does not refer to dashed variables. Such conditions can be characterised as fixed points of the idempotent function $\mathbf{C}(b) \triangleq b \mathbin{;} \mathbf{true}$. For example, the precondition of a **H3** design is a condition on the initial state variables only, and so is **C**-healthy. For reactive relations, we cannot exclude all dashed variables as we wish to express trace constraints using tt, which includes tr and tr'. Consequently, reactive conditions are characterised by the following healthiness condition, **RC**.

Definition 3.9 (Reactive Conditions)**.**

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

We require that $\mathbf{true}_r$ is a right unit of the predicate's negated form, which means, firstly, that it can refer only to undashed state and observational variables other than tr. Secondly, the behaviour of tr is restricted by having $tr \leq tr'$ as a right unit. Intuitively, this means that a reactive condition's complement is extension closed [48]: if trace t_1 is permitted by $\neg_r P$ then for any trace t_2, $t_1 \frown t_2$ is also permitted. Extension closure is here characterised by effectively requiring that $\mathbf{true}_r$ is a right unit of $\neg_r P$.

The reason for this constraint is that if a trace violates a reactive condition, that is the precondition of a reactive contract, then any extension should also violate it. A reactive condition is technically a relation, but can be considered as a condition on the state variables and the trace variable (tt). We can show, for example, that any state condition $[s]_r$, which does not constrain tt, is **RC1** by the following calculation:

Example 3.10 (State Condition is **RC1**)**.**

$$\begin{array}{lr}
\mathbf{RC1}([s]_r) = \neg_r ((\neg_r [s]_r) \mathbin{;} \mathbf{true}_r) & [3.9] \\
= \neg_r ([\neg s]_r \mathbin{;} \mathbf{true}_r) & [\text{predicate calculus}] \\
= \neg_r ((tr \leq tr' \wedge \neg s) \mathbin{;} tr \leq tr') & [\mathbf{R1}, \mathbf{true}_r, [-]_r \text{ definitions}] \\
= \neg_r (tr \leq tr' \mathbin{;} tr \leq tr' \wedge (\neg s) \mathbin{;} tr \leq tr') & [\text{distribution: } \neg s \text{ is condition}] \\
= \neg_r (tr \leq tr' \wedge (\neg s) \mathbin{;} tr \leq tr') & [\text{transitivity of } \leq] \\
= \neg_r (tr \leq tr' \wedge (\exists\, t_0 \bullet (\neg s) \wedge t_0 \leq tr')) & [\mathbin{;} \text{ definition, substitution}] \\
= \neg_r (tr \leq tr' \wedge \neg s) & [\text{predicate calculus}] \\
= \neg_r (\neg_r [s]_r) & [\neg_r, \mathbf{R1} \text{ definitions}] \\
= [s]_r & [\text{double negation}]
\end{array}$$

The calculation first pushes the negation into the state condition, to yield $[\neg s]_r$. Since this is **R1**, but does not otherwise constrain tr and tr', any extension of the trace is permitted. Consequently, $\mathbf{true}_r$ is a right unit of $[\neg s]_r$, and then, by relational calculus, $[s]_r$ is **RC1** healthy. □

Reactive conditions can also constrain tr', but only if the corresponding trace extension tt refers only to a prefix of the trace, leaving the suffix unconstrained. Consider, for example, $\neg_r (\langle a \rangle \leq tt)$, a reactive relation that forbids a from being the first element of tt. It permits the empty trace ε, and any trace $\langle b, \cdots \rangle$, where $b \neq a$. It forbids the trace $\langle a \rangle$ and any extension thereof. It is **RC** healthy, because its negated form is extension closed, as confirmed below.

Example 3.11 (Constrained Prefix is **RC1**)**.**

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

Crucially, the relation $tr \frown \langle a \rangle \leq tr'$ is extension closed, and consequently has $\mathbf{true}_r$ as a right unit. □

Reactive conditions thus serve to restrict permissible initial behaviours in the trace; the previous example states that the event a must not be performed initially. Thus, an alternative characterisation of reactive conditions is that the trace is prefix closed, which can be characterised by the following healthiness condition.

Definition 3.12. $\mathbf{RC2}(P) \triangleq \mathbf{R1}(P \,;\, tr' \leq tr)$

$\mathbf{RC2}$ first sequentially composes P with $tr' \leq tr$, which is the converse of $\mathbf{true}_r$, and states that the trace monotonically decreases. This has the effect of abstracting references to variables other than tr, and recording every trace which is a prefix of the traces tr' produced by P. Then, $\mathbf{R1}$ is applied to remove traces that are shorter than those of the initial tr passed to P. The intuition is given by the following theorem:

Theorem 3.13. *If P is $\mathbf{RR}$ then $\mathbf{RC2}(P) = (\exists (t_0, t_1) \bullet (\exists\, st' \bullet P[\varepsilon, t_1/tr, tr'] \wedge t_0 \leq t_1 \wedge tr' = tr \frown t_0))$*

Here, t_1 is one of the traces contributes by P, and t_0 is arbitrary prefix of t_1. Application of $\mathbf{RC2}$ inserts all such t_1 traces for every t_0 trace, and constructs the overall trace to be the tr extended by t_0. If adding these prefixes as observations has no effect, because they are present already, then the reactive relation is $\mathbf{RC2}$ healthy. Though $\mathbf{RC2}$ is not identical to $\mathbf{RC1}$, we can show that it has the same set of fixed points.

Theorem 3.14. *If P is $\mathbf{RR}$, then P is $\mathbf{RC1}$ if and only if P is $\mathbf{RC2}$*

The theorem shows that a reactive relation is prefix closed when its complement is extension closed, and vice-versa. The intuition is that if reactive condition P admits t_1, then it must also admit any prefix $t_0 \leq t_1$. If t_0 was excluded from P then any extension, including t_1, would also be excluded, since $\neg_r P$ is extension closed, contradicting our assumption. We can therefore use $\mathbf{RC2}$ healthiness to demonstrate $\mathbf{RC1}$ healthiness. Both healthiness conditions are idempotent and continuous, and consequently $\mathbf{RC}$ predicates form a complete lattice. In particular, we retain the lattice top and bottom elements $\mathbf{false}$ and $\mathbf{true}_r$, and also the connectives $\wedge$ and $\vee$. However, $\mathbf{RC}$ predicates are not closed under reactive negation, since this does not preserve prefix closure. The unrestricted use of negation in this context, however, is not necessary for the purposes of this paper.

We also define a reactive weakest liberal precondition operator [14, 30].

Definition 3.15 (Reactive Weakest (Liberal) Precondition)**.**

$$P \;\mathbf{wlp}_r\; Q \;\triangleq\; \neg_r (P \,;\, \neg_r Q) \qquad \text{provided } P \text{ is } \mathbf{RR} \text{ and } Q \text{ is } \mathbf{RC}$$

The predicate has the usual intuition: $P \;\mathbf{wlp}_r\; Q$ is the weakest reactive condition such that if reactive relation P terminates, it achieves a final observation satisfying reactive condition Q. The definition is similar to that given for relations in [33, 9], which effectively takes the complement of the observations under which P fails to establish Q. We have simply replaced relational negation with reactive negation.

The predicate $P \;\mathbf{wlp}_r\; Q$ is a reactive condition ($\mathbf{RC}$) provided that P is reactive relation and Q is a reactive condition. Although, we are using complement, which does not retain prefix closure, we apply it twice which leads to restoration of prefix closure in the final form.

From this definition, we can prove a number of standard $\mathbf{wlp}$ laws [14, 30], which we enumerate below.

Theorem 3.16 (Reactive Weakest Precondition Laws)**.**

$$P \ \mathbf{wlp}_r \ \mathbf{true}_r = \mathbf{true}_r \tag{3.16.1}$$

$$P \ \mathbf{wlp}_r \ (Q \wedge R) = (P \ \mathbf{wlp}_r \ Q \wedge P \ \mathbf{wlp}_r \ R) \tag{3.16.2}$$

$$(P \lhd b \rhd_r Q) \ \mathbf{wlp}_r \ R = (P \ \mathbf{wlp}_r \ R \lhd b \rhd_r Q \ \mathbf{wlp}_r \ R) \tag{3.16.3}$$

$$(P \ ; \ Q) \ \mathbf{wlp}_r \ R = P \ \mathbf{wlp}_r \ (Q \ \mathbf{wlp}_r \ R) \tag{3.16.4}$$

$$\mathit{II}_r \ \mathbf{wlp}_r \ R = R \tag{3.16.5}$$

$$\langle \sigma \rangle_r \ \mathbf{wlp}_r \ R = \sigma \dagger R \tag{3.16.6}$$

$$\mathbf{false} \ \mathbf{wlp}_r \ P = \mathbf{true}_r \tag{3.16.7}$$

$$(P \sqcap Q) \ \mathbf{wlp}_r \ R = (P \ \mathbf{wlp}_r \ R) \wedge (Q \ \mathbf{wlp}_r \ R) \tag{3.16.8}$$

$$\left(\prod i \in A \bullet P(i)\right) \ \mathbf{wlp}_r \ R = (\forall \, i \in A \bullet P(i) \ \mathbf{wlp}_r \ R) \tag{3.16.9}$$

These laws are similar to those given by Dijkstra [14] and Hoare [30]. In particular, we note that the miraculous reactive relation ***false*** has the weakest liberal precondition $\mathbf{true}_r$, which is why it is "liberal": we can make no judgements about a non-terminating reactive relation. The assignment law (Theorem 3.16.6) uses a substitution operator $\sigma \dagger R$ to apply substitution function σ to predicate R. In words, $\langle \sigma \rangle_r$ achieves R provided that R holds when all its variables are replaced by those given in the assignment σ.

We have now constructed a model for simple reactive programs containing both traces, assignments, and conditions. Like UTP relations, reactive relations do not have the expressivity to account for non-terminating reactive behaviours. These are accounted for by our theory of reactive contracts, which we define the next section.

4. Reactive Design Contracts

In this section, we describe the signature of our theory of reactive design contracts and algebraic theorems. Due to its complexity, we defer the definition of the UTP theory's healthiness condition (***NSRD***) until Section 6. As we have mentioned in the introduction, reactive programs can be denoted as contracts that represent their assumptions and guarantees. Our goal is to provide a general method for calculating the contract of reactive program, supported by equational theorems that can reduce a composition of multiple contracts into a single unified contract specification, which can then be subjected to verification. All the laws we present are mechanically proven theorems of our UTP theory; here we also provide some intuition for why they hold. We illustrate the use of our contract notation with a number of *Circus*-based [61] examples, which give intuition, though stateful failure-divergences is not the only applicable semantic model.

4.1. Contracts and Refinement

Reactive program components normally proceed through three phases during execution:

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 state updates are, however, 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. Since reactive programs often do not terminate, this phase may never be reached.

In this view, 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 [46]. Shared variables can, nevertheless, be modelled by encoding them within traces.

Reactive programs can also diverge [48], meaning they exhibit erroneous behaviour, such as engaging in an infinite sequence of internal activity without any communication. Divergence corresponds to violation of a contract's assumptions. A reactive design contract is a triple [8] of the form

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

the three parts of which 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 tt, and potentially other (unprimed) observational variables in the alphabet (r), but not observational variables ok or $wait$, or primed variables other then tr'. Access to tr' is usually indirect through tt.
2. the **pericondition** Q, with commitments the contract guarantees to fulfil during its intermediate execution steps. Often, it used to represent "quiescent" observations, where the program is awaiting interaction with it environment. It is a reactive relation only on the initial states, tt, and any other variables (r, r').
3. the **postcondition** R, with commitments that are 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 specifications, 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 4.1 (Reactive Design Contract)**.**

$$[\, P_1 \vdash P_2 \mid P_3 \,] \triangleq \boldsymbol{R}_s(P_1 \vdash P_2 \diamond P_3) \quad \text{where } P_1 \textit{ is } \boldsymbol{RC}, \text{ and } P_2 \text{ and } P_3 \text{ are both } \boldsymbol{RR}$$

This definition assumes that P_1, P_2, and P_3 are as specified above. This is formalised by requiring that P_1 is a reactive condition, and P_2 and P_3 are both reactive relations, using the theory developed in Section 3. The reactive contract is a form of UTP design which is made reactive using $\boldsymbol{R}_s$. In previous work [46], reactive designs are often written in just two parts ($\boldsymbol{R}_s(P \vdash Q)$), the assumption and guarantee, with the intermediate and final behaviours intertwined. Here, we adopt the triple notation first developed in [8] as it allows us to consider these separately and simplifies many laws. The diamond $P_2 \diamond P_3$ [8] is simply an abbreviation for $P_2 \lhd wait' \rhd P_3$, which distinguishes intermediate and final non-divergent observations.

Our theory supports contract refinement, which is characterised by the following theorem:

Theorem 4.2 (Reactive Design Refinement)**.**

$$[\, P_1 \vdash P_2 \mid P_3 \,] \sqsubseteq [\, Q_1 \vdash Q_2 \mid Q_3 \,] \Leftrightarrow [P_1 \Rightarrow Q_1] \wedge (P_2 \sqsubseteq (Q_2 \wedge P_1)) \wedge (P_3 \sqsubseteq (Q_3 \wedge P_1))$$

This is not a definition, but a theorem of the UTP refinement operator from Definition 2.1 that is supported by the UTP theory (elaborated in Section 6). Theorem 4.2 shows that contract refinement reduces to three proof obligations:

1. the precondition is weakened $(P_1 \Rightarrow Q_1)$;
2. the pericondition of the first contract (P_2) is strengthened by the pericondition of the second (Q_2), conjoined with the precondition of the first (P_1);
3. the postcondition of the first contract (P_3) is strengthened by the postcondition of the second (Q_3), conjoined with the precondition of the first (P_1).

Such a weakening of the assumption and strengthening of the guarantees, of course, is a defining feature of most contract theories [42, 2, 4, 5]. The particular value of this theorem in our case is to provide a foundation for a standard verification procedure for contract-based reactive languages. If a language can be given a contractual denotational semantics, meaning that every operator can be assigned a reactive contract, then we can solve a verification problem, $[P_1 \vdash P_2 \mid P_3] \sqsubseteq Q$, for a specification P and reactive program Q.

We first calculate the program's contact $Q = [Q_1 \vdash Q_2 \mid Q_3]$, and then use Theorem 4.2 to produce the three proof obligation predicates. Then, we can utilise theorem proving technology for relational calculus in Isabelle/UTP [18, 19] to attempt discharge of the proof obligations. In Isabelle/HOL, this can be supported by the sledgehammer proof method [6] that harnesses external automated theorem provers. Consequently, our theorem of contracts can be used to support an automated verification technique for reactive programs. This allows us, in particular, to support verification of programs and models with a very large or infinite state space, since the calculated contracts are symbolic rather than explicit entities, which allows us to overcome the state explosion problem. This, then, is the utility of the complex theory that follows.

In addition to the refinement law, we also have a similar theorem for proving equivalences:

Theorem 4.3 (Reactive Design Equivalence)**.**

$$[\, P_1 \vdash P_2 \mid P_3 \,] = [\, Q_1 \vdash Q_2 \mid Q_3 \,] \Leftrightarrow (P_1 = Q_1) \wedge ((P_2 \wedge Q_1) = (Q_2 \wedge P_1)) \wedge ((P_3 \wedge Q_1) = (Q_3 \wedge P_1))$$

This theorem is a consequence of Theorem 4.3 and the fact that refinement is antisymmetric. Two reactive contracts are equivalent if, and only if, (1) their preconditions are equivalent, and (2) their peri- and postconditions are equivalent modulo the precondition. With this theorem we can similarly automate proving equivalences.

4.2. Denoting Reactive Programs

A crucial requirement of the verification strategy outlined above is that the target language is equipped with a denotational semantics in terms of reactive contracts. We illustrate the use of contracts in giving a denotational semantics to a reactive language by denoting several of the operators from the *Circus* language [61, 46]. For this, we first need to specialise the semantic model to failure-divergences [48]. We thus specialise the trace algebra to finite sequences, $tr : \mathrm{seq}\ Event$, for some suitable set of events, and add the observational variable $ref' : \mathbb{P}\ Event$, as usual [10]. This allows us to record the set of events which are refused when an action is in a quiescent state, or equivalently the set of events that the action is willing to engage in. It is equivalent to encoding CSP failure traces [48], which consist of a sequence of events and a refusal set. Adding observational variables is possible because the alphabet of the reactive design theory is extensible; consequently our verification strategy is effectively parametric in a specialised semantic model.

We begin with the example of *Circus* event prefix [46], which is denoted by a reactive design triple:

Example 4.4 (Event Prefix Reactive Design)**.**

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

This simple event prefix represents a program that, when enabled, waits for the environment to permit an a event, and following this, terminates. It is denoted by a contract with a true precondition since it can never diverge; every environment is a valid context. Its pericondition encodes a single quiescent observation: the event a is not refused and no events has been contributed to the trace as yet. The postcondition states that, when the program terminates, 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: $a \rightarrow P \triangleq (a \rightarrow \textbf{\textit{Skip}})\ ;\ P$. □

Other examples are the ***Skip*** action, which represents a terminating process, and the ***Stop*** action, which represents a deadlock.

Example 4.5 (Terminated and Deadlocked Actions)**.**

$$\mathbf{Skip} \triangleq [\, \mathbf{true}_r \vdash \mathbf{false} \mid tt = \langle\rangle \land st' = st \,]$$
$$\mathbf{Stop} \triangleq \left[\, \mathbf{true}_r \vdash tt = \langle\rangle \land ref' \subseteq Event \;\middle|\; \mathbf{false} \,\right]$$

The terminated action ***Skip*** has a true precondition. It has no intermediate observations, so the pericondition is ***false***, as it is essentially instantaneous and never pauses for interaction. In the postcondition, it is specified that the action makes no contribution to the trace, and leaves the state variables unchanged. The deadlocked action (***Stop***) likewise has a true precondition. No state is a final state, indicated by the ***false*** postcondition, since the process does not terminate. In the quiescent states it is simply required that the trace is unchanged, and any refusal set is observable, since no event is enabled. □

Our final example is external choice over a contract indexed by set A.

Example 4.6 (External Choice)**.**

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

This is the first example of a contract composition law: it shows how a collection of contracts, in this case an indexed set, can be composed in a single contract. Such composition laws can then be combine with definitions, like those in Examples 4.4 and 4.5 for contract calculation. The overall contract permits internal activity in the choice branches, but the choice itself is not resolved until an external event occurs. The precondition requires that the preconditions of 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 ($tt = \langle\rangle$), 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 the three example definitions look different from the standard presentation of *Circus* [46], they are largely equivalent. Indeed, the definitions given above are largely theorems of the original *Circus* definitions, and therefore our encoding is conservative. The exception is event prefix, in which we conceal the state whilst waiting for the event, following previous work [7].

We now have a contractual denotational semantics for simple *Circus* actions. For verification, we also need to specify properties for reactive programs using specification contracts. A common desirable property of *Circus* actions and CSP processes is deadlock-freedom [48], which states that a process never reaches a quiescent state where no event is enabled. It can be specified using the following reactive contract:

Definition 4.7 (Deadlock-freedom Contract)**.** $\mathbf{CDF} \triangleq \left[\, \mathbf{true}_r \vdash \exists\, e \bullet e \notin ref' \;\middle|\; \mathbf{true}_r \,\right]$

This reactive contract has a $\mathbf{true}_r$ precondition, which by Theorem 4.2, means that the precondition of the implementation contract must also be $\mathbf{true}_r$. This is because we must weaken the precondition, and $\mathbf{true}_r$ is the weakest possible reactive condition. Intuitively, this means that to refine ***CDF***, a reactive program must also be free of divergence. The postcondition is also $\mathbf{true}_r$, but since we must strengthen the postcondition, any postcondition for the implementation is admitted. The pericondition contains the main specification formula; it states that in every quiescent observation there must be an event which is not being refused. In other words, only programs that do not admit the observation $ref' = Event$ are deadlock-free.

We can show, for example, that $a \rightarrow \mathbf{Skip}$ is deadlock-free:

Example 4.8 ($\mathbf{CDF} \sqsubseteq a \rightarrow \mathbf{Skip}$)**.** Since the precondition of $a \rightarrow Skip$ is $\mathbf{true}_r$, it suffices to consider the pericondition, and show that the following refinement holds:

$$(\exists\, e \bullet e \notin ref') \sqsubseteq a \notin ref' \land tt = \langle\rangle$$

Recall that refinement is reverse implication. Therefore, we need to show that $a \notin \mathit{ref}' \Rightarrow (\exists\, e \bullet e \notin \mathit{ref}')$, which straightforwardly holds when we set $e = a$. Consequently, we have proved deadlock-freedom.

Conversely, we cannot show that ***Stop*** is deadlock-free, because its pericondition includes $\mathit{ref}' \subseteq \mathit{Event}$, which allows the possibility of refusing everything. □

4.3. Calculational Laws

Though language-specific operators like those above for *Circus* can be expressed, many core contract operators can be introduced generically. We can therefore develop a large body of laws for calculating contracts that do not depend on a particular semantic model, but can be instantiated with any trace algebra $\mathcal{T}$, and additional observational variables. We begin by denoting some basic reactive operators for this generic theory.

Theorem 4.9 (Reactive Design Core Operators)**.**

$$\begin{aligned} \mathbb{I}_R &= [\, \mathit{true}_r \vdash \mathit{false} \mid \mathbb{I}_r \,] \\ \langle \sigma \rangle_R &= [\, \mathit{true}_r \vdash \mathit{false} \mid \langle \sigma \rangle_r \,] \\ \mathit{Miracle} &= [\, \mathit{true}_r \vdash \mathit{false} \mid \mathit{false} \,] \\ \mathit{Chaos} &= [\, \mathit{false} \vdash \mathit{false} \mid \mathit{false} \,] \end{aligned}$$

Operator $\mathbb{I}_R$ is reactive design identity. 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, by using $\mathbb{I}_r$ as a postcondition, we also add the conjunct $r' = r$ which is shorthand for saying all additional variables are unchanged. This distinguishes $\mathbb{I}_R$ from the *Circus*-specific ***Skip*** operator from Example 4.5, which leaves ref' unconstrained.

Operator $\langle \sigma \rangle_R$ is a generalised assignment, 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 $x :=_R v$ can be expressed as $\langle \{x \mapsto v\} \rangle_R$. The generalised assignment also enables us to easily define multiple-variable assignment constructs.

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. It is the top element of the refinement lattice:

Theorem 4.10. $[\, P_1 \vdash P_2 \mid P_3 \,] \sqsubseteq \mathit{Miracle}$

This follows, by Theorem 4.2, since the precondition true_r is the least reactive condition, and ***false*** is the greatest reactive relation. ***Chaos***, in contrast to ***Miracle***, is the contract with an unsatisfiable precondition and thus always yields a program error. It is the bottom of the refinement lattice, and is the least deterministic contract:

Theorem 4.11. $\mathit{Chaos} \sqsubseteq [\, P_1 \vdash P_2 \mid P_3 \,]$

Chaos can be used to identify interactions that are erroneous, and thus the context should avoid them, as illustrated by the following example.

Example 4.12 (Divergent Process)**.**

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

Here, we have utilised Examples 4.4 and 4.6, together with Definition 4.9 to calculate the composite contract. 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 of the form illustrated in Example 3.11. If the program performs a, then the behaviour is unpredictable. The pericondition states that the trace has not yet been extended, and the action does not refuse a or 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 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 4.13 (Reactive Design Compositions)**.** 

$$[\,P_1 \vdash P_2 \mid P_3\,] \sqcap [\,Q_1 \vdash Q_2 \mid Q_3\,] = [\,P_1 \wedge Q_1 \vdash P_2 \vee Q_2 \mid P_3 \vee Q_3\,] \tag{4.13.1}$$

$$\prod_{i \in I} [\,P_1(i) \vdash P_2(i) \mid P_3(i)\,] = \left[\, \bigwedge_{i \in I} P_1(i) \;\middle|\; \bigvee_{i \in I} P_2(i) \;\middle|\; \bigvee_{i \in I} P_3(i) \,\right] \tag{4.13.2}$$

$$[\,P_1 \vdash P_2 \mid P_3\,] \sqcup [\,Q_1 \vdash Q_2 \mid Q_3\,] = \left[\, \begin{array}{c} P_1 \\ \vee\ Q_1 \end{array} \;\middle|\; \begin{array}{c} P_1 \Rightarrow_r P_2 \wedge \\ Q_1 \Rightarrow_r Q_2 \end{array} \;\middle|\; \begin{array}{c} P_1 \Rightarrow_r P_3 \wedge \\ Q_1 \Rightarrow_r Q_3 \end{array} \,\right] \tag{4.13.3}$$

$$[\,P_1 \vdash P_2 \mid P_3\,] \lhd b \rhd [\,Q_1 \vdash Q_2 \mid Q_3\,] = \left[\, \begin{array}{c} P_1 \\ \lhd\, b \,\rhd \\ Q_1 \end{array} \;\middle|\; \begin{array}{c} P_2 \\ \lhd\, b \,\rhd \\ Q_2 \end{array} \;\middle|\; \begin{array}{c} P_3 \\ \lhd\, b \,\rhd \\ Q_3 \end{array} \,\right] \tag{4.13.4}$$

$$[\,P_1 \vdash P_2 \mid P_3\,] \,;\, [\,Q_1 \vdash Q_2 \mid Q_3\,] = \left[\, \begin{array}{c} P_1 \wedge \\ P_3 \ \mathbf{wlp}_r\ Q_1 \end{array} \;\middle|\; \begin{array}{c} P_2 \vee \\ P_3 \,;\, Q_2 \end{array} \;\middle|\; P_3 \,;\, Q_3 \,\right] \tag{4.13.5}$$

$$[\,P \vdash Q \mid R\,]^{n+1} = \left[\, \bigwedge_{i \leq n} \left(R^i \ \mathbf{wlp}_r\ P\right) \;\middle|\; \bigvee_{i \leq n} R^i \,;\, Q \;\middle|\; R^{n+1} \,\right] \tag{4.13.6}$$

These are theorems, rather than definitions, since they formulate the semantics of contracts that are composed by the UTP operators defined previously in Definition 2.2. We do not need to redefine them for our theory of reactive designs, but rather prove laws that show how to calculate composite contracts. This is a key contribution of our work, since it means the existing theorems of operators like ; and $\sqcap$ can be directly imported into our theory of reactive designs, and applied to algebraic reasoning.

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 non-deterministic, and thus there must be no possibility of divergence in any of the possible branches. Internal choice can, alternatively, be viewed as a disjunction operator for contracts similar to that in [4]. 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, $P \ \mathbf{wlp}_r\ Q$. 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, $P_3 \,;\, Q_3$.

As a corollary, we prove 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 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 $i \leq n$, followed 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.

Finally we present a law for calculating the contract of a tail-recursive program of the form $\mu_{\mathbf{R}}\, X \bullet P \mathbin{;} X$, where $\mu_{\mathbf{R}}$ is the weakest fixed-point operator, which allows us to formulate iterative contracts. This is subject to P being a productive [13] contract, that is, one that extends the trace when it terminates.

Definition 4.14. $[\, P_1 \vdash P_2 \mid P_3 \,]$ is said to be productive if P_3 is a fixed-point of $\mathbf{R4}(P) \mathrel{\widehat{=}} P \wedge tr < tr'$, that is if we establish termination then it is necessary that the trace strictly increases.

For example, $a \rightarrow \mathbf{Skip}$ is productive because it always produces an a event upon termination. Its postcondition is $\mathbf{R4}$ healthy because it contains the conjunct $tt = \langle a \rangle$, which strictly increases the trace ($tr < tr'$). On the other hand, $\langle \sigma \rangle_{\mathbf{R}}$ is not productive because it contributes no events to the trace. Productivity is related to, but not the same as the common notion of "guardedness" [33], which, as explained in Section 6.5, applies to a function on contracts rather than a contract itself. If a contract's postcondition is productive, then we have the following theorem.

Theorem 4.15 (Recursive Reactive Design)**.** *If R is $\mathbf{R4}$ healthy, then*

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

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 R 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 give some of the algebraic laws of reactive design contracts.

Theorem 4.16 (Reactive Design Laws)**.**

$$\begin{aligned}
\mathbf{Miracle} \sqcap [\, P_1 \vdash P_2 \mid P_3 \,] &= [\, P_1 \vdash P_2 \mid P_3 \,] && \text{(RD1)}\\
\mathbf{Chaos} \sqcap [\, P_1 \vdash P_2 \mid P_3 \,] &= \mathbf{Chaos} && \text{(RD2)}\\
\mathbb{I}_{\mathbf{R}} \mathbin{;} [\, P_1 \vdash P_2 \mid P_3 \,] &= [\, P_1 \vdash P_2 \mid P_3 \,] && \text{(RD3)}\\
[\, P_1 \vdash P_2 \mid P_3 \,] \mathbin{;} \mathbb{I}_{\mathbf{R}} &= [\, P_1 \vdash P_2 \mid P_3 \,] && \text{(RD4)}\\
[\, P_1 \vdash P_2 \mid \mathbf{false} \,] \mathbin{;} [\, Q_1 \vdash Q_2 \mid Q_3 \,] &= [\, P_1 \vdash P_2 \mid \mathbf{false} \,] && \text{(RD5)}\\
\mathbf{Miracle} \mathbin{;} [\, P_1 \vdash P_2 \mid P_3 \,] &= \mathbf{Miracle} && \text{(RD6)}\\
\mathbf{Chaos} \mathbin{;} [\, P_1 \vdash P_2 \mid P_3 \,] &= \mathbf{Chaos} && \text{(RD7)}\\
[\, \mathbf{false} \vdash P_2 \mid P_3 \,] &= \mathbf{Chaos} && \text{(RD8)}\\
[\, P_1 \vdash P_2 \mid P_3 \,] \mathbin{;} \mathbf{Miracle} &= [\, P_1 \vdash P_2 \mid \mathbf{false} \,] && \text{(RD9)}\\
[\, P_1 \vdash P_2 \mid P_3 \,] \mathbin{;} \mathbf{Chaos} &= [\, P_1 \wedge (P_3 \mathbin{\mathbf{wlp}_r} \mathbf{false}) \vdash P_2 \mid \mathbf{false} \,] && \text{(RD10)}
\end{aligned}$$

All of these laws can be proved by calculation using the definitions in Theorem 4.9 and laws in Theorem 4.13. RD1 establishes that a choice between a $\mathbf{Miracle}$ and P yields P, since $\mathbf{Miracle}$ is the top of the lattice. Similarly, RD2 establishes that a choice between $\mathbf{Chaos}$ and P yields $\mathbf{Chaos}$. The reactive skip is a left and right identity for any contract P, as this is stated by RD3 and RD4, respectively.

Law RD5 states that any non-terminating contract – that is where the postcondition is $\mathbf{false}$ – is a left zero for sequential composition, as clearly then the successor is unreachable. Thus, in particular $\mathbf{Miracle}$ and $\mathbf{Chaos}$ are both left zeros for sequential composition, as shown by RD6 and RD7. Moreover, RD7 shows that any reactive contract with a $\mathbf{false}$ precondition is $\mathbf{Chaos}$.

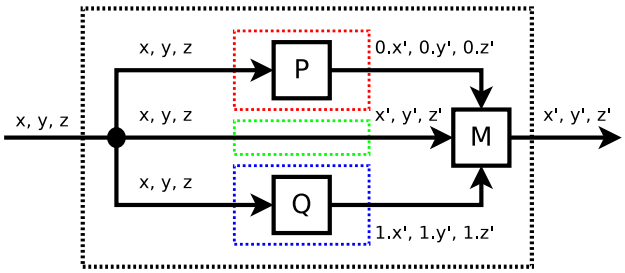

Figure 1: Parallel-by-merge Dataflow

Law RD9 is a property first observed in [59]: placing a ***Miracle*** after a reactive design eliminates final states, and yields a non-terminating process. Since it is impossible to reach a miraculous state, inserting one prunes transitions that lead to it.

Finally, RD10 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 $\mathbf{wlp}_r$ ***false***, which effectively states that postcondition P_3 should not be established, because otherwise chaos will ensue. This explains Example 4.12: 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 4.17 (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.

4.4. Parallel Contracts

The final operator we tackle in this section is parallel composition, written $P \parallel_R^M Q$. Our definition of parallel composition, elaborated in Section 6.6, uses the parallel-by-merge scheme developed as part of UTP [33]. Since different concurrency schemes are possible for reactive contracts, depending on the underlying notion of trace, we cannot define a single parallel composition operator and so $P \parallel_R^M Q$ is parametric over M. This is a merge predicate that defines how the state, traces, and any other observational variables should be merged following execution of P and Q.

We illustrate parallel-by-merge in Figure 1, where we assume the programs act on three variables, x, y, and z. Parallel-by-merge splits the observation space into three identical segments: one for P, one for Q, and a third that is identical to the original input. Relation M then takes the outputs from P, Q, and the original inputs, and merges them into a single output.

For reactive designs, the variables we need to merge are the state variables (st), trace (tt), and any additional observational variables r. As an example, we present below the merge predicate for interleaving the events of two *Circus* processes [33, 47].

Example 4.18 (Interleaving Merge)**.**

$$
\begin{aligned}
M_c &\triangleq tt \in (0.tt \mid\mid\mid_t 1.tt) \land ref' \subseteq 0.ref \cap 1.ref \quad \text{where} \\
\langle\rangle \mid\mid\mid_t xs &= xs \mid\mid\mid_t \langle\rangle = \{xs\} \\
\langle x\rangle \frown xs \mid\mid\mid_t \langle y\rangle \frown ys &= \{\langle x\rangle \frown zs \mid zs \in (xs \mid\mid\mid_t \langle y\rangle \frown ys)\} \cup \\
&\quad\ \{\langle y\rangle \frown zs \mid zs \in (\langle x\rangle \frown xs \mid\mid\mid_t ys)\} \\
P \mid\mid\mid Q &\triangleq P \parallel_{\mathbf{R}}^{M_c} Q
\end{aligned}
$$

In the definition of $P \parallel_{\mathbf{R}}^{M} Q$, the trace and refusal variables are decorated with an index 0 or 1 that determine whether the quantity originates from P or Q, respectively. The binary operator $\mid\mid\mid_t$ interleaves two traces; it is a recursive function on sequences, returning a set of possible traces. The merge predicate, M_c, firstly constructs the overall trace tt as one of all possible interleavings, and secondly states that an event is only refused if it is refused by both processes. Our merge predicate leaves the state variable st unspecified as the *Circus* interleaving operator hides any internal state, since they can not in general be merge without further machinery for shared variables. We then define the interleaving operator $P \mid\mid\mid Q$ using the parametric reactive design parallel composition operator $\parallel_{\mathbf{R}}^{\cdot}$, which is formally defined in Section 6.6. □

For the purpose of generic laws, we assume that any merge predicate yields well-formed traces, and furthermore is symmetric. Symmetry in this context means that the merge predicate has no bias towards either of its operands and yields the same result if they are swapped. This is clearly the case in Example 4.18, since both the traces and refusals are composed by symmetric operators, namely $\mid\mid\mid_t$ and $\cap$. The following theorem describes the result of composing two contracts.

Theorem 4.19 (Reactive Design Parallel Composition)**.**

$$
\begin{aligned}
&[\, P_1 \vdash P_2 \mid P_3 \,] \parallel_{\mathbf{R}}^{M} [\, Q_1 \vdash Q_2 \mid Q_3 \,] = \\
&\left[\begin{array}{l|l|l}
(P_1 \Rightarrow_r P_2)\ \mathbf{wr}_M\ Q_1 \land & P_2 \parallel_{\mathbf{E}}^{M} Q_2 \lor & \\
(P_1 \Rightarrow_r P_3)\ \mathbf{wr}_M\ Q_1 \land & P_3 \parallel_{\mathbf{E}}^{M} Q_2 \lor & P_3 \parallel_M Q_3 \\
(Q_1 \Rightarrow_r Q_2)\ \mathbf{wr}_M\ P_1 \land & P_2 \parallel_{\mathbf{E}}^{M} Q_3 & \\
(Q_1 \Rightarrow_r Q_3)\ \mathbf{wr}_M\ P_1 & &
\end{array}\right]
\end{aligned}
$$

The precondition is expressed in terms of a reactive condition combinator $\mathbf{wr}_M$. This is a form of weakest rely condition: $A\ \mathbf{wr}_M\ B$ describes the weakest context in which reactive relation A does not violate reactive condition B. This is necessary because in a composition like $P \parallel_{\mathbf{R}}^{M} Q$, the reactive processes P and Q interfere and this can lead to the violation of their preconditions. Thus, the overall precondition of a parallel composition itself assumes that such interferences do not occur. Interferences cannot occur in Example 4.18, since there is no synchronisation, but, in general, of course it can happen when more specialised merge predicates are employed. The definition of $\mathbf{wr}_M$ is elided for now, as it requires further elaboration of the UTP theory, but is given in Section 6, Definition 6.26. It obeys several theorems that are shown below:

Theorem 4.20 (Weakest Rely Laws)**.**

$$
\mathbf{false}\ \mathbf{wr}_M\ P = \mathbf{true}_r \qquad P\ \mathbf{wr}_M\ \mathbf{true}_r = \mathbf{true}_r \qquad \left(\prod_{i \in I} P(i)\right)\ \mathbf{wr}_M\ Q = \left(\bigwedge_{i \in I} (P(i)\ \mathbf{wr}_M\ Q)\right)
$$

The laws show, respectively, that (1) a miraculous reactive relation satisfies any precondition, (2) any reactive relation satisfies a true precondition, and (3) the weakest rely condition of a disjunction of relations is the conjunction of their weakest rely conditions.

The parallel composition precondition in Theorem 4.19 states that the respective preconditions of the two contracts, P_1 and Q_1, are not violated, neither by the opposing periconditions – respectively Q_2 and P_2 – under their respective preconditions, or the opposing postconditions – respectively Q_3 and P_3. The pericondition of the parallel reactive contract is written in terms of operator $P \parallel^M_{\mathbf{E}} Q$ that merges the traces but not the states, since this is concealed in intermediate observations. A parallel process is in an intermediate state if either of the composed processes is. Finally, the postcondition merges the final trace and state of each process, directly using the parallel by merge operator ($\parallel_M$). We formally define all these operators in Section 6.6.

We can use Theorems 4.19 and 4.20 to prove the following theorem; for illustration we elaborate the complete calculation.

Theorem 4.21. $\mathbf{Miracle} \parallel^M_{\mathbf{R}} P = \mathbf{Miracle}$

Proof.

$$
\begin{aligned}
\mathbf{Miracle} \parallel^M_{\mathbf{R}} P &= [\, \mathbf{true}_r \vdash \mathbf{false} \mid \mathbf{false} \,] \parallel^M_{\mathbf{R}} [\, P_1 \vdash P_2 \mid P_3 \,] \\
&= \left[\begin{array}{l|l|l} (\mathbf{true}_r \Rightarrow_r \mathbf{false})\ \mathbf{wr}_M\ P_1 \wedge \\ (\mathbf{true}_r \Rightarrow_r \mathbf{false})\ \mathbf{wr}_M\ P_1 \wedge \\ (P_1 \Rightarrow_r P_2)\ \mathbf{wr}_M\ \mathbf{true}_r \wedge \\ (P_1 \Rightarrow_r P_3)\ \mathbf{wr}_M\ \mathbf{true}_r \end{array} \middle| \begin{array}{l} \mathbf{false} \parallel^M_{\mathbf{E}} P_2 \vee \\ \mathbf{false} \parallel^M_{\mathbf{E}} P_2 \vee \\ \mathbf{false} \parallel^M_{\mathbf{E}} P_3 \end{array} \middle| \mathbf{false} \parallel_M P_3 \right] && [4.19] \\
&= \left[\begin{array}{l} \mathbf{false}\ \mathbf{wr}_M\ P_1 \wedge \mathbf{false}\ \mathbf{wr}_M\ P_1 \wedge \\ \mathbf{true}_r \wedge \mathbf{true}_r \end{array} \middle| \ \mathbf{false} \vee \mathbf{false} \vee \mathbf{false} \ \middle| \ \mathbf{false} \right] && [4.20] \\
&= [\ \mathbf{true}_r \wedge \mathbf{true}_r \mid \mathbf{false} \mid \mathbf{false}\ ] && [4.20] \\
&= [\, \mathbf{true}_r \vdash \mathbf{false} \mid \mathbf{false} \,] \\
&= \mathbf{Miracle} \qquad \square
\end{aligned}
$$

This law shows that composition of any predicate with a miracle always yields a miracle, regardless of the merge predicate. The intuitive property that ***Chaos*** is similarly an annihilator depends on the form of merge predicate, and so this cannot be proved in general. We perform a further example calculation using the interleaving operator from Example 4.18.

Example 4.22 (Interleaving Calculation)**.**

$$
\begin{aligned}
& a \rightarrow \mathbf{Skip} \mid\mid\mid b \rightarrow \mathbf{Stop} \\
&= [\, \vdash a \notin ref' \wedge tt = \langle\rangle \mid st' = st \wedge tt = \langle a \rangle \,] \mid\mid\mid [\, \vdash a \notin ref' \wedge tt = \langle\rangle \vee tt = \langle b \rangle \mid \mathbf{false} \,] \\
&= \left[\begin{array}{l|l|l} (a \notin ref' \wedge tt = \langle\rangle) \parallel^{M_C}_{\mathbf{E}} (b \notin ref' \wedge tt = \langle\rangle) \vee \\ (a \notin ref' \wedge tt = \langle\rangle) \parallel^{M_C}_{\mathbf{E}} (tt = \langle b \rangle) \vee \\ (st' = st \wedge tt = \langle a \rangle) \parallel^{M_C}_{\mathbf{E}} (b \notin ref' \wedge tt = \langle\rangle) \vee \\ (st' = st \wedge tt = \langle a \rangle) \parallel^{M_C}_{\mathbf{E}} (tt = \langle b \rangle) \end{array} \middle| \ (st' = st \wedge tt = \langle a \rangle) \parallel_M \mathbf{false} \right] \\
&= \left[\begin{array}{l|l} (a \notin ref' \wedge b \notin ref' \wedge tt = \langle\rangle) \vee \\ (a \notin ref' \wedge tt \in \{\langle\rangle, \langle b \rangle\}) \vee \\ (b \notin ref' \wedge tt \in \{\langle\rangle, \langle a \rangle\}) \vee \\ (tt = \langle a, b \rangle) \end{array} \middle| \ \mathbf{false} \right] \\
&= \left[\begin{array}{l|l} (a \notin ref' \wedge b \notin ref') \lhd tt = \langle\rangle \rhd \left(\begin{array}{l} a \notin ref' \wedge tt = \langle b \rangle \\ \vee\ b \notin ref' \wedge tt = \langle a \rangle \\ \vee\ tt = \langle a, b \rangle \end{array}\right) \end{array} \middle| \ \mathbf{false} \right] \\
&= a \rightarrow b \rightarrow \mathbf{Stop} \ \Box\ b \rightarrow a \rightarrow \mathbf{Stop}
\end{aligned}
$$

We use the abbreviation $[\, \vdash P_2 \mid P_3 \,]$ for a contract with a $\mathbf{true}_r$ precondition. In the first step, we calculate the meaning of the two sequential processes using Examples 4.4 and 4.5, with Theorem 4.13. We then employ

Theorem 4.19 to expand out the overall parallel reactive contract. There is no possibility of divergence, so the preconditions remain trivial. For the pericondition we need to merge each quiescent observation with every opposing quiescent or final observation. The result is the four conjuncts displayed. In the postcondition, we have to merge the two overall postconditions. The overall postcondition becomes ***false***, because ***Stop*** prevents the overall operator from successfully terminating. In the pericondition, we calculate the four merged quiescent observations. These characterise the states when (1) no event has yet occurred, and we are accepting a or b; (2) either $\langle\rangle$, or else $\langle b\rangle$ has occurred and a remains enabled, which includes the situation when the right hand action has performed a transition; (3) the symmetric case to (2) with a potentially having occurred; and (4) both events have occurred and no event is enabled: the refusal set is unconstrained. In common with the semantics of CSP[48], we observe that the set of possible refusals is downward closed under $\subseteq$ for every trace combination. We can then in fact show that this contract is equivalent to an external choice using the definition given in Example 4.6. □

This completes our exposition of the calculational laws for our reactive contract theory. All the theorems presented have been mechanically checked, as we discuss in Section 7. In the next section we illustrate their use in verification.

5. Automated Verification using Contracts

In this section, we exemplify the use of reactive contracts to verify properties of a small cash-card system described in *Circus*, using the verification procedure outlined in Section 4.1. We will explicitly calculate the implementation contracts, and show how these are then verified. Though the corresponding calculations are complex, the crucial detail is that they symbolically characterise reactive programs with potentially infinite state, and can be produced automatically in Isabelle/UTP.

In this example, cards can independently perform transfers to one another, provided sufficient balance exists. A key requirement is that there is no loss or increase of the value shared across the cards. For simplicity, we will construct a purely sequential specification of this system, and show how these properties can be discharged. We use a modified version of the model described in [62], which describes a network of a number of cards, each of which is identified by a natural number $\mathbb{N}$. Monetary amounts are represented as integers, $\mathbb{Z}$, so that we can additionally represent negative balances.

The central process has a single state variable $accts : \mathbb{N} \pfun \mathbb{Z}$, a partial function that represents the set of accounts: the balance on each card. We also introduce three channels:

- $pay : \mathbb{N} \times \mathbb{N} \times \mathbb{Z}$, to initiate a transfer request of a given value between two cards;
- $reject : \mathbb{N}$, to indicate rejection of a transfer from the given card identifier;
- $accept : \mathbb{N}$, to indicate acceptance.

In order to update a particular account stored in $accts$, we need a form of assignment that applies to a single entry. We therefore introduce the following indexed assignment operator:

Definition 5.1 (Indexed Assignment)**.**

$$x(i) :=_c v \mathrel{\widehat{=}} [\, i \in \mathrm{dom}(\mathit{st}.x) \vdash \mathbf{false} \mid \mathit{st}' = \mathit{st}(x \mapsto x(i \mapsto v)) \land \mathit{tt} = \langle\rangle \,]$$

Indexed assignment $x(i) :=_c v$, as employed by the Pay action, is unlike regular assignment in that it is a partial operator and can only be executed when the collection x has the index i defined. Consequently, it must be guarded by an assumption $i \in \mathrm{dom}(\mathit{st}.x)$, which states that the index $i : A$ is in the domain of variable $x : A \pfun B$.

We give indexed assignment a denotational semantics using a *Circus* contract, thus extending the available operators, which also illustrates the extensibility of our approach. The construct has similar semantics to regular assignment, but has the precondition that the given collection index must exist. The postcondition states that the collection state variable x is updated so that the index i maps to v. If the precondition is violated, then the result is divergence as the following calculation demonstrates:

Example 5.2 (Divergent Indexed Assignment)**.**

$$
\begin{array}{lll}
& x :=_R \emptyset \,;\, x(i) :=_c v & \\
= & [\, \mathit{true}_r \vdash \mathit{false} \mid st' = st(x \mapsto \emptyset) \wedge tt = \langle\rangle \,] \,; & \\
& [\, i \in \mathrm{dom}(st.x) \vdash \mathit{false} \mid st' = st(x \mapsto x(i \mapsto v)) \wedge tt = \langle\rangle \,] & [4.9, 5.1] \\
= & [\, \mathit{true}_r \wedge (st' = st(x \mapsto \emptyset) \wedge tt = \langle\rangle)\ \mathit{wlp}_r\, (i \in \mathrm{dom}(st.x)) \vdash \mathit{false} \mid \cdots] & [4.13.5] \\
= & [\, i \in \mathrm{dom}(\emptyset) \vdash \mathit{false} \mid \cdots] & [\text{relational calculus}] \\
= & \mathit{Chaos} & [RD8]
\end{array}
$$

If we assign $\emptyset$, the empty partial function, to x and then attempt to manipulate the ith element of x, the result is always ***Chaos***. □

If we use an indexed assignment within a reactive program, then it is necessary to show that the applied indices are always in scope. We can now define the key actions for the cash card system.

Definition 5.3 (Card System)**.**

$$
\begin{array}{rcl}
Pay(i, j : \mathbb{N}, n : \mathbb{Z}) & \triangleq & pay.i.j.n \rightarrow \\
& & \left(\begin{array}{l} reject.i \rightarrow \mathit{Skip} \\ \quad \lhd\, i = j \vee i \notin \mathrm{dom}(accts) \vee n \leq 0 \vee n > accts(i) \,\rhd \\ \left(\begin{array}{l} accts(i) :=_c accts(i) - n \,; \\ accts(j) :=_c accts(j) + n \,; \\ accept.i \rightarrow \mathit{Skip} \end{array} \right) \end{array} \right) \\
PaySet(cs : \mathbb{P}\,\mathbb{N}) & \triangleq & \{(i : \mathbb{N}, j : \mathbb{N}, n : \mathbb{Z}) \mid i \in cs \wedge j \in cs \wedge i \neq j\} \\
SomePay(cs : \mathbb{P}\,\mathbb{N}) & \triangleq & \prod (i, j, n) \in PaySet(cs) \bullet Pay(i, j, n) \\
Cycle(cs : \mathbb{P}\,\mathbb{N}) & \triangleq & \mu\, X \bullet SomePay(cs) \,;\, X \\
System & \triangleq & accts :=_c \langle 100, 100, 100, 100, 100 \rangle \,;\, Cycle(\{0..4\})
\end{array}
$$

Action $Pay(i, j, n)$ defines the protocol to execute a payment request between cards i and j of amount n. If $i = j$, and thus the cards are the same, or card i does not exist, then the transfer is rejected, by offering $reject.i$ and then terminating. Likewise, if there is insufficient balance or a negative transfer is requested, then the transfer is also rejected. If the transfer can be performed, then two indexed assignments lower the balance of card i and raise the balance of card j, respectively. Finally, the $accept.i$ event is offered and then the action terminates.

The main behaviour of the system is described by the *Cycle* action, which takes the set of card identifiers $cs : \mathbb{P}\,\mathbb{N}$ as a parameter. It iterates the action $SomePay(cs)$, which consists of an internal choice over all possible payments between all possible cards, $PaySet(cs)$. The behaviour of an example card system is described by the action *System* that creates 5 cards, each with a balance of 100, and then begins the cycle.

In order to verify properties of the processes, we first need to calculate the reactive design contract of the system. For the purposes of illustration, we focus on the *Pay* action. The other contracts can be calculated in terms of the *Pay* contract using the laws presented in Section 4. The following theorem provides the result of the calculation. Like all other theorems in this paper, it is proved, not by hand, but by use of our tactics in Isabelle/UTP.

Theorem 5.4 (Pay Action Contract Calculation)**.**

$Pay(i, j, n) = [\, P_1 \vdash P_2 \mid P_3 \,]$ *where*

$$
\begin{array}{rcl}
P_1 = & \langle pay.i.j.n \rangle \leq tt \wedge i \neq j \wedge i \notin \mathrm{dom}(accts) \wedge 0 < n \wedge n < accts(i) \\
\Rightarrow & j \in \mathrm{dom}(accts)
\end{array}
$$

$$
\begin{aligned}
P_2 = \;& \mathit{tt} = \langle\rangle \wedge \mathit{pay.i.j.n} \notin \mathit{ref}' \\
\vee \;& \mathit{tt} = \langle \mathit{pay.i.j.n} \rangle \wedge (i = j \vee n < 0 \vee n > \mathit{accts}(i)) \wedge \mathit{reject.i} \notin \mathit{ref}' \\
\vee \;& \mathit{tt} = \langle \mathit{pay.i.j.n} \rangle \wedge i \neq j \wedge n < 0 \wedge n > \mathit{accts}(i) \wedge \mathit{accept.i} \notin \mathit{ref}'
\end{aligned}
$$

$$
\begin{aligned}
P_3 = \;& \mathit{tt} = \langle \mathit{pay.i.j.n}, \mathit{reject.i} \rangle \wedge (i = j \vee n < 0 \vee n > \mathit{accts}(i)) \wedge \mathit{st}' = \mathit{st} \\
\vee \;& \mathit{tt} = \langle \mathit{pay.i.j.n}, \mathit{accept.i} \rangle \wedge i \neq j \wedge n < 0 \wedge n > \mathit{accts}(i) \\
& \qquad \wedge \mathit{accts} := \mathit{accts}(i \mapsto \mathit{accts}(i) - n, j \mapsto \mathit{accts}(j) + n)
\end{aligned}
$$

Theorem 5.4 gives the result of the calculation of the precondition P_1, pericondition P_2, and postcondition P_3 of the implementation contract for the *Pay* action. Intuitively, this complex contract symbolically characterises the potentially infinite state space and possible transitions that the *Pay* action can make, depending on the initial state. The precondition specifies the circumstances under which behaviour is predictable, the pericondition specifies the possible quiescent observation, and the postcondition specifies the terminating observations.

Precondition P_1 requires that, if the *pay.i.j.n* event occurs, that is, it is at the head of the trace, and the conditions for a valid transfer are all satisfied by the state, then it must be the case that card j exists in the state space. This precondition arises directly from the indexed assignment: its violation leads to unpredictable behaviour. Thus, if the transfer is not offered or the payment is not valid then this assignment is not reached, and so the precondition precisely identifies the state in which divergence is possible. We alternatively could remove this precondition altogether by altering the definition of *Pay* so that if $j \notin \mathrm{dom}(\mathit{accts})$ then also a *reject* event is issued. However, for illustration purposes we leave the precondition in place. This precondition is a reactive condition because it only restricts a prefix of the trace to the left of the implication and only refers to initial state variables.

The pericondition, P_2, specifies the three possible intermediate observations for the action. Firstly, we have the scenario in which nothing has happened yet, so the trace is empty and the *pay.i.j.n* is being offered – it is not being refused. Secondly, it is possible that the transfer request occurred, and so the trace has a singleton event, but one of the conditions for a valid transfer is violated, and thus the *reject.i* event is being offered. Thirdly, we can have that a valid transfer request occurred and so the *accept.i* event is being offered. At this point the state update has happened internally, but it cannot yet be observed as the action is still in an intermediate state.

The postcondition, P_3, specifies two possible final states, one for an invalid transfer request, in which case the state remains the same, and one for a valid transfer, in which case the two balances are updated. In both cases the trace is updated with two events, and no refusals are recorded since we have terminated.

The next step in verification is to specify some holistic properties of our system that we would like to show. We chose three properties: (1) there is no increase or decrease in the overall balance across cards, (2) no overdrafts on the card balances are permitted, and (3) if a valid transfer request is made it must be executed. It is not difficult to see that these properties hold, but the purpose is to show how they can specified and verified using reactive contracts.

Each of these properties is assigned a contract which $\mathit{Pay}(i, j, n)$ must refine. We therefore demonstrate the three properties as three corresponding theorems, which have been discharged in Isabelle/UTP using the rdes-refine tactic (see Section 7), which employs Theorem 4.2 and the contract calculation laws. For the purpose of illustration, we give high-level informal proofs that correspond to the mechanised proofs.

Theorem 5.5 (No Increase or Decrease in Value)**.** *A payment between card i and j, where $\{i, j\} \subseteq cs$ and $i \neq j$, does not lead to an overall change in balance for the system of cards. Formally, as a reactive contract:*

$$
[\,\mathrm{dom}(\mathit{accts}) = \mathit{cs} \vdash \mathbf{true}_r \mid \mathit{sum}(\mathit{accts}) = \mathit{sum}(\mathit{accts}')\,] \sqsubseteq \mathit{Pay}(i, j, n)
$$

where sum is a function that sums up the range of the given partial function. We set the pericondition $\mathbf{true}_r$ as we do not need to constrain quiescent states in this specification.

Proof. We first apply Theorem 4.2 to split the refinement into three proof obligations. We need to show that the precondition is weakened, and the peri- and postconditions are strengthened, which we tackle one at a time:

1. $\mathrm{dom}(accts) = cs \Rightarrow P_1$
2. $P_2 \wedge \mathrm{dom}(accts) = cs \Rightarrow \mathbf{true}_r$
3. $P_3 \wedge \mathrm{dom}(accts) = cs \Rightarrow sum(accts) = sum(accts')$

Case (1) follows because $i, j \in cs$ and since the domain of $accts$ is cs, then clearly both $i, j \in \mathrm{dom}(accts)$. Case (2) follows trivially. Case (3) requires that we consider both cases of postcondition P_3. If the payment request is invalid, then $accts' = accts$ since $\mathbf{st}' = \mathbf{st}$, and thus clearly $sum(accts) = sum(accts')$. If the payment request is valid, then we need to show that $sum(accts) = sum(accts(i \mapsto accts(i) - n, j \mapsto accts(j) + n))$. This is equal to $sum(accts) - n + n = sum(accts)$, and so we are done. □

Theorem 5.6 (No overdrafts)**.** *No card i is permitted to have a negative balance:*

$$[\,\mathrm{dom}(accts) = cs \vdash \mathbf{true}_r \mid \forall\, k \in cs \bullet accts(k) \geq 0 \Rightarrow accts'(k) \geq 0\,] \sqsubseteq Pay(i, j, n)$$

Proof. The argument for the pre and periconditions is the same as in the previous theorem. For the postcondition we need to show that after a valid payment the balance of any valid card k is not less than 0, that is, $accts(k) \geq 0$. The key property to prove here is $(accts(i \mapsto accts(i) - n, j \mapsto accts(j) + n))(k) \geq 0$. We can do this by case analysis: $k = i$, $k = j$, and $k \neq i \wedge k \neq j$. In the former two cases, the fact that the transfer happens indicates that the balances following must be no less than zero. In the final case, the balance remains the same, and so we are done. □

Theorem 5.7 (Transfer Acceptance)**.** *If a payment is initiated and we have enough money in the account, then the transfer is not rejected:*

$$\left[\mathrm{dom}(accts) = cs \;\middle|\; \begin{array}{l} tt \neq \langle\rangle \wedge last(tt) = pay.i.j.k \wedge \\ n \leq accts(i) \Rightarrow accept.i \notin ref' \end{array} \;\middle|\; \mathbf{true}_r\right]$$

Proof. This property is different as it involves the pericondition rather than the postcondition. This is because we are reasoning about offered events in intermediate observations. We need to show in the pericondition that if the last event to occur is $pay.i.j.n$, and a sufficient amount is in account i then we must not refuse to the accept the payment. This can be achieved by case analysis on pericondition P_2. □

Thus we have shown how our design contracts can be used to verify properties of simple *Circus* actions. In the next section we explore the UTP theory's healthiness conditions behind our contracts — which provides support for this verification.

6. Theory of Generalised Reactive Designs

In this section, we present our UTP theory of reactive design contracts in detail. We describe the healthiness conditions, core signature definitions, and algebraic laws, which substantiate those given in Section 4. In particular, we define healthiness conditions for two UTP theories: ***SRD*** in Section 6.2, which is a recasting of the previous reactive design theory from [46], and ***NSRD*** in Section 6.3, a novel healthiness condition that refines ***SRD*** with additional constraints to support reactive preconditions and invisible intermediate states. This latter healthiness condition is the foundation of our reactive contract theory. We consider the algebraic law this theory supports, and the restrictions it places on expressible operators. In Section 6.5, we consider the formalisation of recursion, and show how Kleene's fixed-point theorem [39] can be applied to calculate tail recursive reactive designs. Finally in Section 6.6, we detail our results in the formalisation of parallel composition.

6.1. UTP Theory Preliminaries

The theorems presented in Section 4, present reactive contracts using the syntactic form of $[\, P_1 \vdash P_2 \mid P_3 \,]$. It is often inconvenient to rely on such a specific syntactic form to reason about contracts, for example when stating algebraic theorems. It is much more convenient to a reactive contract P as a relation that admits certain properties. Consequently, as usual in the UTP approach, we define healthiness conditions that characterise well-formed contract predicates. This allows us to obtain a large number of algebraic laws from lattice theory and related domains, and also allows us to reason about contracts without the need for the syntactic form.

In order to reason about contracts as opaque relations, we need a way of extracting the three parts from a contract. We therefore define the following three functions:

Definition 6.1 (Pre-, Peri-, and Postcondition Extraction 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 characteristic predicates. In order to illustrate this, we first expand out the definition of reactive contract from Definition 4.1, and also using Definitions 2.6 and 2.10:

$$[\, P_1 \vdash P_2 \mid P_3 \,] = \mathbf{R1}(\mathbf{R2}_c(\mathbb{I}_{\mathbf{R}} \lhd wait \rhd ((ok \wedge P_1) \Rightarrow (ok' \wedge (P_2 \lhd wait' \rhd P_3)))))$$

From this, we can see that the crucial variables that dictate the various behaviours are $wait$, ok, ok', and $wait'$. Variable $wait$ flags whether a sequential predecessor is in an intermediate state or has terminated, ok flags whether a predecessor diverged, and ok' and $wait'$ record this information for the present relation.

The precondition (P_1) 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 reached when the reactive design was started and did not diverge (ok'), but has not yet reached its final state ($\neg wait'$), and is thus intermediate. The postcondition is similarly obtained, but $wait'$ becomes $false$. With these definitions we can prove theorems that allow us to extract the constituent reactive relations from a syntactic reactive contract.

Theorem 6.2 (Extracting Pre, Peri, and Postconditions)**.**

$$\begin{aligned} pre_{\mathbf{R}}([\, P_1 \vdash P_2 \mid P_3 \,]) &= P_1 \\ peri_{\mathbf{R}}([\, P_1 \vdash P_2 \mid P_3 \,]) &= P_1 \Rightarrow_r P_2 \\ post_{\mathbf{R}}([\, P_1 \vdash P_2 \mid P_3 \,]) &= P_1 \Rightarrow_r P_3 \end{aligned}$$

*Provided P_1, Q_1, and Q_2 are all **RR** healthy.*

The pericondition and postcondition are viewed through the prism of the precondition being satisfied, which is why the implications are present in the result above. This is an important assumption of reactive designs. Although it is possible to specify behaviours in the peri and postcondition outside the precondition, once the contract is constructed these behaviours are pruned, so that the behaviour of a contract when its precondition is violated is always ***Chaos***.

6.2. Stateful Reactive Designs

In this section, we introduce the first of our two reactive design theories: stateful reactive designs. Motivated by the previous work on CSP and *Circus* in UTP [33, 10, 46], we define the following healthiness conditions for our generalised theory of reactive designs. We generalise the previous works [33, 10, 46] due to the underlying abstract trace algebra, and the extensible alphabet of our theory [16].

Definition 6.3 (Stateful Reactive Designs Healthiness Conditions)**.**

$$\begin{aligned} \mathbf{RD1}(P) &\triangleq ok \Rightarrow_r P \\ \mathbf{RD2}(P) &\triangleq P \,;\, \mathbf{J} \\ \mathbf{SRD}(P) &\triangleq \mathbf{RD1} \circ \mathbf{RD2} \circ \mathbf{R}_s \end{aligned}$$

RD1 and **RD2** are analogous to **H1** and **H2** from the theory of designs. Moreover, **RD1** and **RD2** correspond to **CSP1** and **CSP2** from the theories of CSP [33, 10] and *Circus* [46]. We rename them, firstly because our theory has a different alphabet founded upon our trace algebra, and secondly because we do not 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 $\mathbf{true}_r$; that is, the trace must monotonically increase, but the behaviour is otherwise unpredictable. **RD2** is identical to **H2** and thus also **CSP2**.

Our overall healthiness condition for stateful reactive designs is then called **SRD**, which includes **RD1**, **RD2**, and $\mathbf{R}_s$. Like **RD1** and **RD2**, it is a generalisation of the overall *Circus* healthiness condition **CSP** [46]. We can next prove that **SRD** relations admit a certain syntactic formulation using reactive contracts, as shown in the following theorem.

Theorem 6.4 (Stateful Reactive Design Formulation)**.**

$$\mathbf{SRD}(P) = [\, pre_{\mathbf{R}}(P) \vdash peri_{\mathbf{R}}(P) \mid post_{\mathbf{R}}(P) \,]$$

If a relation is **SRD** healthy, then it takes the form of a reactive contract with a pre-, peri-, and postcondition. Theorem 6.4 thus allows us to take an **SRD** predicate and deconstruct it into its three parts, using the functions defined in Definition 6.1, which can then be manipulated separately. This theorem shows an equivalence between the syntactic formulation, and elements of $[\![\mathbf{SRD}]\!]_{\mathsf{H}}$: any element of the latter can be constructed using the contract triple notation. Conversely, a well-formed reactive design triple always yields a healthy reactive design. This is confirmed by the following closure theorem, which is a corollary of Theorems 6.4 and 6.2.

Corollary 6.4.1. *If P_1, P_2, and P_3 are all* **RR** *healthy then $[\, P_1 \vdash P_2 \mid P_3 \,]$ is* **SRD** *healthy.*

In order to show that a reactive contract is **SRD**, it suffices to show that its pre-, peri-, and postcondition are all **RR**. A further corollary is the three triple extraction functions all produce healthy reactive relations when applied to an **SRD** relation.

Corollary 6.4.2. *If P is* **SRD** *then $pre_{\mathbf{R}}(P)$, $peri_{\mathbf{R}}(P)$, and $post_{\mathbf{R}}(P)$ are all* **RR***-healthy.*

Having defined a candidate theory for our reactive contracts, we will now explore its algebraic properties. The class of **SRD** relations admits a number of useful identities that we outline below.

Theorem 6.5 (**SRD** Laws)**.** *If P is* **SRD** *healthy then*

$$\mathbb{I}_{\mathbf{R}} \,;\, P = P \tag{6.5.1}$$

$$\mathbf{Chaos} \,;\, P = \mathbf{Chaos} \tag{6.5.2}$$

$$\mathbf{Miracle} \,;\, P = \mathbf{Miracle} \tag{6.5.3}$$

$\mathbb{I}_{\mathbf{R}}$ is a left unit of any **SRD** relation, and **Chaos** and **Miracle** are both left annihilators. However, in general $\mathbb{I}_{\mathbf{R}}$ is not a right unit, which the following theorem illustrates.

Theorem 6.6. *If P_1, P_2, and P_3 are* **RR** *then $[P_1 \vdash P_2 \mid P_3] \,;\, \mathbb{I}_{\mathbf{R}} = [\, \neg_r (\neg_r P_1) \,;\, \mathbf{true}_r \vdash \exists\, st' \bullet P_2 \mid P_3 \,]$*

Right composition with $\mathbb{I}_{\mathbf{R}}$ effectively imposes two additional requirements on the reactive contract: (1) that the negated precondition does not refer to dashed variables other than tr', which must be extension closed — it is **RC1** (cf. Definition 3.9); and (2) that the pericondition does not refer to st', as characterised by

the existential quantifier. The latter follows as a direct consequence of $\mathbf{R3}_h$ [7] (cf. Definition 2.10), which requires that st must not be restricted in an intermediate state. Thus, composition of P with $\mathbb{I}_R$ has the same effect on st' in P since $\mathbb{I}_R$ ignores any intermediate states ($wait' = true$), rather than passing them forward to a successor process.

The former restriction is a direct consequence of healthiness condition **RD1**. To explain why, we consider the reactive design $P \mathbin{;} \mathbb{I}_R$ in the situation when P has started ($ok \wedge wait$), but diverged ($\neg ok'$). The following calculation shows the result.

Example 6.7 (Divergence and **RD1**).

$$
\begin{array}{lr}
& (P \wedge ok \wedge \neg ok' \wedge \neg wait) \mathbin{;} \mathbb{I}_R \\
= & P[true, false, false/ok, ok', wait'] \mathbin{;} \mathbb{I}_R[false/ok] & \text{[relational calculus]} \\
= & (\neg_r pre_R(P)) \mathbin{;} \mathbb{I}_R[false/ok] & \text{[Definition 6.1]} \\
= & (\neg_r pre_R(P)) \mathbin{;} (\mathbf{RD1}(\mathbb{I}_R))[false/ok] & \text{[Theorem 6.4.1]} \\
= & (\neg_r pre_R(P)) \mathbin{;} (ok \Rightarrow_r \mathbb{I}_R)[false/ok] & \text{[Definition 6.3]} \\
= & (\neg_r pre_R(P)) \mathbin{;} \mathbf{true}_r & \text{[predicate calculus]}
\end{array}
$$

When P diverges with $\neg ok'$, its behaviour is $\neg_r pre_R(P)$: the precondition has been violated. Moreover, if ok' is false in P then ok is false in $\mathbb{I}_R$, which yields the relation $\mathbf{true}_r$. Actually, this follows for any **RD1** relation, not just $\mathbb{I}_R$, since $\mathbf{RD1}[false/ok] = \mathbf{true}_r$. Consequently, right composition with any **SRD** relation yields the same result: the precondition should be **RC1**.

Consequently, the right composition law motivates that we need to consider a refined theory for reactive contracts, in order to support the desirable algebraic and calculational laws of Section 4. This is the objective of the following section.

6.3. Normal Stateful Reactive Designs

We refine **SRD** by identifying the subclass of normal stateful reactive designs, **NSRD**, which provides the theory domain of our reactive contracts.

Definition 6.8 (Normal Stateful Reactive Designs Healthiness Conditions).

$$
\begin{array}{rcl}
\mathbf{RD3}(P) & \triangleq & P \mathbin{;} \mathbb{I}_R \\
\mathbf{NSRD}(P) & \triangleq & \mathbf{RD1} \circ \mathbf{RD3} \circ \mathbf{R}_s
\end{array}
$$

RD3 is analogous to **H3**: it requires that skip is a right unit. This ensures that the precondition is a reactive condition and the pericondition does not depend on st'. The use of the word "normal" here is therefore by analogy with normal designs [21]. **NSRD** does not explicitly invoke **RD2** as it is subsumed by **RD3**, as the following theorem demonstrates.

Theorem 6.9 (**RD3** subsumes **RD2**). $\mathbf{RD2}(\mathbf{RD3}(P)) = \mathbf{RD3}(\mathbf{RD2}(P)) = \mathbf{RD3}(P)$

Proof. This follows since $\mathbf{J} \mathbin{;} \mathbb{I}_R = \mathbb{I}_R \mathbin{;} \mathbf{J} = \mathbb{I}_R$. □

Consequently, it is easy to show that every **NSRD**-healthy relation is also **SRD**-healthy, and therefore the theorems of Section 6.2 remain valid. We can also prove the following theorem that follows as a consequence of Theorems 6.9 and 6.6.

Theorem 6.10 (Normal Stateful Reactive Design Formulation).

$$\mathbf{NSRD}(P) = [\, \mathbf{RC1}(pre_R(P)) \vdash (\exists\, st' \bullet peri_R(P)) \mid post_R(P) \,]$$

In addition to ensuring that the elements of the triple of **RR** healthy, **NSRD** also ensures that the precondition is a reactive condition, and that the pericondition does not refer to st'.

Corollary 6.10.1. *P is **NSRD** healthy provided that the following conditions hold:*

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

A further corollary finally justifies the syntactic formulation given in Section 4:

Corollary 6.10.2 (**NSRD** contract closure)**.**
*If P_1 is **RC**, P_2 and P_3 are both **RR**, and P_2 does not refer to st', then $[\, P_1 \vdash P_2 \mid P_3 \,]$ is **NSRD** healthy.*

We now, therefore, have an adequate formulation of the reactive design theory. In spite of its restrictions, **SRD** remains useful as a means of obtaining algebraic theorems. In the remainder of this section we will therefore explore the algebraic properties of the two theories.

6.4. Algebraic Properties

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

Theorem 6.11 (Reactive Design Lattices)**.** ***SRD** and **NSRD** healthy predicates form complete lattices with $\top_{\mathsf{R}} \triangleq \mathbf{SRD}(\mathbf{false}) = \mathbf{Miracle}$ and $\bot_{\mathsf{R}} \triangleq \mathbf{SRD}(\mathbf{true}) = \mathbf{Chaos}$.*

Proof. Standard proofs of idempotency for **CSP1** and **CSP2** [33, 10] apply also to **RD1** and **RD2**, respectively. **RD3** is idempotent since $\mathbb{I}_{\mathsf{R}} \,;\, \mathbb{I}_{\mathsf{R}} = \mathbb{I}_{\mathsf{R}}$. **RD1** is continuous since it is disjunctive. **RD2** and **RD3** 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 6.10.2, are both **NSRD** healthy. Through the Knaster-Tarski theorem [55], we also obtain weakest fixed-point operators μ_{R} and μ_{N} for the two theories, which we will further explore in Section 6.5. Since **SRD** and **NSRD** are also continuous, by Theorem 2.5 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 6.12 (Reactive Design Composition)**.** *If P and Q are **NSRD** healthy then*

$$P \,;\, Q = \mathbf{R}_s \left(\begin{array}{ll} & pre_{\mathsf{R}}(P) \wedge (post_{\mathsf{R}}(P) \ \mathbf{wlp}_r \ pre_{\mathsf{R}}(Q)) \\ \vdash & peri_{\mathsf{R}}(P) \vee (post_{\mathsf{R}}(P) \,;\, peri_{\mathsf{R}}(Q)) \\ \diamond & post_{\mathsf{R}}(P) \,;\, post_{\mathsf{R}}(Q) \end{array} \right)$$

Theorem 6.12 is essentially the same as Theorem 4.13.5, 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 and pericondition, following the form given in Theorem 6.6. We finally show closure of the theory under the main programming operators.

Theorem 6.13 (Reactive Designs Closure)**.** ***SRD** and **NSRD** healthy predicates are closed under $\sqcap$, $\sqcup$, $\lhd\, b \,\rhd$, $\,;\,$, $\mathbb{I}_{\mathsf{R}}$, $\langle\sigma\rangle_{\mathsf{R}}$, **Miracle**, and **Chaos**.*

This closure theorem means that we can import the algebraic laws for the relational calculus operators from core UTP [33], such as several equations of Theorem 2.3. Finally, we note that our theory admits the following familiar laws from relational calculus for assignment.

Theorem 6.14 (Reactive Design Assignment Compositions)**.** *If P is **NSRD** healthy, x is a state variable, and v is an expression containing only state variables, then:*

$$(x :=_{\mathsf{R}} v) \,;\, P = P[v/x] \tag{6.14.1}$$

$$\langle\sigma\rangle_{\mathsf{R}} \,;\, P = \sigma \dagger P \tag{6.14.2}$$

Theorem 6.14.1, the more usual law [32], shows that a sequential assignment can be turned into a substitution applied to its successor. It is an instance of the more general Theorem 6.14.2 when $\sigma = \{x \mapsto v\}$, crucially, provided that x is a state variable and not an arbitrary UTP variable.

Theorem 6.14 depends directly on the use of $\mathbf{R3}_h$, and the resulting requirement that periconditions do not refer to after state. It would not hold if allowed such references, for example by substituting $\mathbf{R3}_h$ with the original healthiness condition **R3** [33]. In order to understand why, consider the following corollary of Theorem 6.14.1 involving a *Circus* event prefix:

Theorem 6.15 (Assignment and Events)**.**

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

Theorem 6.15 allows us to push assignment through event prefixes, whilst making an appropriate substitution. It can be found in reactive languages like Occam [49]. This law does not hold in previous *Circus* semantics [46] as st' (there called v') is revealed in intermediate states. Thus, whilst the left hand side of this equation admits an intermediate observation where x is updated to e, because the pericondition does characterise state updates, the right hand side does not.

There are, however, costs to this simplification. A side effect is to prevent encoding of McEwan's *Circus* interruption operator [40], which has the form $P \triangle_i Q$. This operator is similar to $P \,|||\, (i \to Q)$, except that if the interruption event i occurs before P terminates, and whilst it is quiescent, then the remaining behaviour of P is pruned. If P terminates, then conversely the behaviours of Q are lost. Crucially, the internal state of P is retained following interruption [7], and is passed on to Q, which is unique to McEwan's work (cf. [53, 58, 60]). For example, consider the following action:

$$(x := 5 \,;\, a \to \mathbf{Skip}) \triangle_b (x := x + x).$$

First, we note that the leading assignment $x := 5$ cannot be interrupted, since it does not have a quiescent state. However, once this has occurred the left-hand side enters a quiescent state where a is enabled. Through the interruption operator, b is also enabled at this point. If a occurs then the entire action terminates in a final state with $x = 5$. However, if b occurs then the value of x at this point is retained, and the action $x := x + x$ occurs, leading to final value of 10 for x.

We see, therefore, how the value of x is retained and used by the interruption action. Such a retention of the state, however, cannot be represented in the presence of **RD3** as no intermediate state variables are recorded. Therefore, if such an interruption operator is required, the loss of Theorem 6.15 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, as **R3** reactive designs carry more information and so are more distinguishing than $\mathbf{R3}_h$ reactive designs. **R3** is supported in Isabelle/UTP if its use is desired.

6.5. 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 4.15. In Section 6.4, we have shown that generalised reactive designs form a complete lattice. Thus, for any monotonic process constructor F we can be sure there exists fixed-points $\mu_R F$ and $\nu_R F$. However, in order to reason about recursive reactive contracts 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, $\mu_R F$, as the strongest fixed-point yields miraculous behaviour for erroneous constructions [9]. For example, $(\nu_R X \bullet X) = \mathbf{Miracle}$, whereas in reality an infinite loop is a programmer error that should yield **Chaos**, which $\mu_R X \bullet X$ 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 [33, theorem 8.1.13, page 206], and (2) Kleene's fixed-point theorem [39]. 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 functions on reactive processes have a unique fixed-point, that is, if, for any relation X, $F(X)$ is guarded, then $\mu\, F = \nu F$. Guardedness is defined as follows.

Definition 6.16 (Guarded Reactive Designs)**.** A function on reactive designs $F : [\![\mathbf{SRD}]\!]_H \to [\![\mathbf{SRD}]\!]_H$ is guarded provided that, for any $P \in [\![\mathbf{SRD}]\!]_H$ and $n \in \mathbb{N}$,

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

where $gv(n) \triangleq (tr \leq tr' \wedge \#\mathit{tt} < n)$ and $\# : \mathcal{T} \to \mathbb{N}$ is a discrete trace measure function. We extend the trace algebra $(\mathcal{T}, \hat{}, \varepsilon)$ with the following axioms for the measure function; for $s, t \in \mathcal{T}$:

$$\#\varepsilon = 0 \qquad s > 0 \Rightarrow \#s > \varepsilon \qquad \#(s \hat{} t) = \#s + \#t$$

Definition 6.16 is similar to the one given in [33], but is generalised to allow a variety of different discrete measure functions that satisfy the measure function axioms. An example measure function that satisfies these axioms is the length function on sequences.

Given a reactive design $\mu\, X \bullet F(X)$, if F is guarded according to Definition 6.16 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 requiring 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 use this to prove Hoare and He's theorem.

Theorem 6.17 (Unique Fixed-Points)**.** *If F is guarded then $\mu\, F = \nu F$.*

Technically, the μ and ν operators we use here are those of the relational calculus lattice, and not of an arbitrary UTP theory (hence the lack of subscripts). The proof given in [33, theorem 8.1.13, page 206], which our mechanised proof follows, omits the step that transitions the theory fixed-point operator (μ_R) to the relational one (μ). However, this step is necessary in order to employ their approximation chain theorem [33, theorem 2.7.6, page 63]. Thus, in order to employ Theorem 6.17 for reactive designs, we first have to use Theorems 6.11 and 2.5 to convert the reactive design fixed-point operator. Continuity of $\mathbf{SRD}$ is thus an important property. Using that, for any guarded process, we can convert a recursive construction using a weakest fixed-point to one using a strongest fixed-point.

In order to make use of Theorem 6.17, it is necessary to prove guardedness theorems for the operators of the target language. In general, this can be quite complicated; in many cases, however, we can shortcut guardedness and instead focus on tail-recursive fixed-point constructions of the form $\mu_R\, X \bullet P \mathbin{;} X$, where X is not mentioned in P, as employed by Theorem 4.15. 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 $\mu_R\, X \bullet P \mathbin{;} X$ reduces to P. Whether or not P terminates, productivity is the criterion needed (see Definition 4.14), and from this we can prove the following theorem.

Theorem 6.18. *If P is productive then the function $\lambda\, X \bullet P \mathbin{;} X$ is guarded.*

So, by Theorem 6.17 we can map the weakest fixed-point to the strongest fixed-point. This brings us to Kleene's fixed-point theorem, which allows us to calculate an iterative construction for the strongest fixed-point of a continuous function.

Theorem 6.19 (Kleene's fixed-point theorem)**.** *If F is a continuous function then the strongest fixed-point can be calculated by iteration:*

$$\nu F \;=\; \bigsqcap_{i \in \mathbb{N}} F^i(\mathbf{false})$$

Kleene's fixed-point theorem often employs Scott-continuity as its antecedent, which is based on complete partial orders rather than complete lattices. We employ our stronger notion of continuity in Theorem 6.19, since we do have a complete lattice in our setting. Theorem 6.19 allows us to calculate the fixed-point by iterating F, starting from false, which is the top ($\top$) of the relational lattice. Now, for our simplified pattern $\nu X \bullet P \mathbin{;} X$ we automatically have continuity since relational composition is continuous, that is

$$(\bigsqcap i \bullet P(i)) \mathbin{;} Q = (\bigsqcap i \bullet P(i) \mathbin{;} Q)$$

is a theorem of relational calculus (see Theorem 2.3). Combining this property with Theorem 4.13.6, which includes the calculation for a power construction, we are now in the position to substantiate Theorem 4.15 for calculating iterative reactive contracts. We give the calculational proof below explicitly since it uses and illustrates several of our results.

Proof of Theorem 4.15.

$$\begin{aligned}
&\mu_{\mathsf{R}}\, X \bullet [\, P \vdash Q \mid R \,] \mathbin{;} X \\
&= \mu\, X \bullet [\, P \vdash Q \mid R \,] \mathbin{;} \mathit{SRD}(X) && \text{[Theorem 2.5.6]} \\
&= \nu X \bullet [\, P \vdash Q \mid R \,] \mathbin{;} \mathit{SRD}(X) && \text{[Theorems 6.17 and 6.18]} \\
&= \bigsqcap_{i \in \mathbb{N}} (\lambda\, X \bullet [\, P \vdash Q \mid R \,] \mathbin{;} \mathit{SRD}(X))^i(\mathit{false}) && \text{[Theorem 6.19]} \\
&= \bigsqcap_{i \in \mathbb{N}} (\lambda\, X \bullet [\, P \vdash Q \mid R \,] \mathbin{;} \mathit{SRD}(X))^{i+1}(\mathit{false}) && \text{[Unfold: } f^0(\mathit{false}) = \mathit{false}\text{]} \\
&= \bigsqcap_{i \in \mathbb{N}} ([\, P \vdash Q \mid R \,]^{i+1} \mathbin{;} \mathit{SRD}(\mathit{false})) && \text{[Induction on } i\text{]} \\
&= \bigsqcap_{i \in \mathbb{N}} ([\, P \vdash Q \mid R \,]^{i+1} \mathbin{;} \mathit{Miracle}) && \text{[Theorem 2.5.2]} \\
&= \bigsqcap_{i \in \mathbb{N}} \left[\bigwedge_{i \le n} (R^i \mathbin{\mathit{wlp}_r} P) \;\middle|\; \bigvee_{i \le n} R^i \mathbin{;} Q \;\middle|\; R^{n+1} \right] \mathbin{;} \mathit{Miracle} && \text{[Theorem 4.13.6]} \\
&= \bigsqcap_{i \in \mathbb{N}} \left[\bigwedge_{i \le n} (R^i \mathbin{\mathit{wlp}_r} P) \;\middle|\; \bigvee_{i \le n} R^i \mathbin{;} Q \;\middle|\; \mathit{false} \right] && \text{[Theorem RD9]} \\
&= \left[\bigwedge_{i \in \mathbb{N}} (R^i \mathbin{\mathit{wlp}_r} P) \;\middle|\; \bigvee_{i \in \mathbb{N}} R^i \mathbin{;} Q \;\middle|\; \mathit{false} \right] && \left[\begin{array}{r}\text{Theorem 4.13.2 and}\\ \text{relational calculus}\end{array}\right]
\end{aligned}$$

□

This proof demonstrates the necessity of a large corpus of theorems we have proved from the UTP theories and reactive designs in order to reason about recursion. We now have a complete constructive approach for calculating the reactive contract for a variety of recursive specifications. Extension to deal with mutual recursion is laborious, but not challenging.

6.6. Parallel Composition

In this section we introduce the parametric parallel composition operator for reactive designs, and substantiate a number of preliminary results for the operator, including well-formedness conditions supported by a novel healthiness condition. Parallel composition in UTP is expressed in terms of the parallel-by-merge scheme $P \parallel_M Q$, whereby the final states of concurrent separated processes P and Q are merged by predicate M. We adopt a slightly simplified definition of parallel-by-merge, originally presented in [19], which assumes

that the alphabet of both P and Q is the same, but is otherwise semantically equivalent to the standard UTP definition [33, chapter 7].

Definition 6.20 (Parallel-by-Merge)**.**

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

The intuition for parallel-by-merge is given in Figure 1. It splits the state space into three parts, one of which is passed to P, the second to Q, and the third is simply copied from the original input state. Assuming both P and Q produce an output, the merge predicate M computes the overall output that should be given by merging the three states. For imperative programs, a simple example merge predicate may pick two disjoint subsets of the variables output by P and Q to produce an overall output. The value of such a scheme for parallel composition is that different concurrency schemes can be supported, and properties of the parallel composition operator often reduce to properties of the merge predicate [33, theorem 7.2.10, page 173].

To formalise this in Definition 6.20, we require that P and Q are homogeneous relations with the alphabet $\{v, v'\}$, where v is a vector of variables. $\lceil P \rceil_0$ and $\lceil Q \rceil_1$ rename the dashed variables by adding indices 0 and 1, respectively, so they can be distinguished by the merge predicate M[2]. M is a heterogeneous alphabetised relation whose alphabet is $\{0.v, 1.v, v, v'\}$. It takes three copies of the variables: the renamed dashed variables of P and Q, and all undashed variables. The separated processes are conjoined, which is acceptable because of their disjoint output alphabets, along with a predicate that copies all initial variables ($v' = v$). The resulting conjunctive relation is then composed in sequence with M. This merge predicate calculates the overall final state in terms of the initial state, and final states of P and Q. Thus, the overall composition $P \parallel_M Q$ has alphabet $\{v, v'\}$.

For our theory of reactive designs, the objective, as indicated by Section 4.4 is to specialise parallel-by-merge so that it acts only on the trace, state, and other semantic observational variables (like *ref*). We therefore define the parametric merge predicate $\mathcal{M}_R(M)$ that merges *ok* and *wait* variables, so handling divergence and intermediate observations. It defers merging of the states and traces to an "inner merge predicate" M, such as M_c described in Example 4.18. We require that M does not refer to *ok* or *wait* nor decorations thereof.

We define three auxiliary merge operators that construct the "outer merge predicate", by showing how *ok* and *wait* are merged, and imposition the reactive design healthiness conditions. These operators are then used to define the reactive design parallel composition operator.

Definition 6.21 (Reactive Designs Parallel Composition)**.**

$$\begin{aligned}
\mathcal{N}_0(M) &\triangleq (wait' = (0.wait \vee 1.wait) \wedge tr \leq tr' \wedge M) \\
\mathcal{N}_1(M) &\triangleq (ok' = (0.ok \wedge 1.ok) \wedge \mathcal{N}_0(M)) \\
\mathcal{M}_R(M) &\triangleq \mathbf{RD3}(\mathbf{RD1}(\mathbf{R3}_h(\mathcal{N}_1(M)))) \\
P \parallel_{\mathbf{R}}^{M} Q &\triangleq P \parallel_{\mathcal{M}_R(M)} Q
\end{aligned}$$

The auxiliary merge functions $\mathcal{N}_0$ and $\mathcal{N}_1$ conjoin the inner merge predicate M with three conjuncts. $\mathcal{N}_0$ handles merging of the *wait* variables. If either P or Q admits an intermediate observation ($0.wait$ or $1.wait$), then also the composite observation is intermediate, and thus we take the disjunction of the *wait* variables to determine $wait'$. The third conjunct ensures the resulting merge is **R1** healthy. $\mathcal{N}_1$ handles merging of the *ok* variables. If either P or Q diverges, then we require that their composition also diverges and thus ok' takes the conjunction of both *ok* variables. Using $\mathcal{N}_1$, we then define the overall merge predicate $\mathcal{M}_R$ by application of three healthiness conditions to construct a stateful reactive design. The parametric merge of two reactive designs, $P \parallel_{\mathbf{R}}^{M} Q$ with inner merge predicate M, is simply a parallel-by-merge using $\mathcal{M}_R(M)$.

It remains to prove that **NSRD** is closed under reactive design parallel composition. In order to prove this, we need to restrict the form of the merge predicate using new healthiness conditions. Firstly, we need a modified version of $\mathbf{R2}_c$ that is applicable to merge predicates, first defined in [16].

[2]These are called "separating simulations" in [33, page 172], and are denoted using special relations called $U0$ and $U1$.

Definition 6.22 ($\mathbf{R2}_c$ for Merge Predicates)**.**

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

Merge predicates have three ways of accessing the trace history through the three respective copies of the trace variable. Thus, it is necessary to delete the history in $0.tr$, $1.tr$, and tr in the healthiness condition $\mathbf{R2}_m$ to ensure that this does not occur. This allows us to prove the following theorem

Theorem 6.23. *$P \parallel_M Q$ is **R1** and $\mathbf{R2}_c$ healthy provided that P and Q are both **R1** and $\mathbf{R2}_c$, and M is **R1** and $\mathbf{R2}_m$.*

This theorem provides the circumstances under which a parallel-by-merge constructs a healthy reactive process. From this, we define the following healthiness condition for reactive-design inner merge predicates.

Definition 6.24 (Healthy Inner Merge Predicate)**.**

$$\mathbf{RDM}(M) \triangleq \mathbf{R2}_m(\exists\, 0.ok, 1.ok, ok, ok', 0.wait, 1.wait, wait, wait' \bullet M)$$

RDM contains $\mathbf{R2}_m$ and additionally formalises the requirement that the inner merge predicate does not contain *ok*, *wait*, and decorations thereof, through existential quantification. These variables have already been handled by the outer merge predicate, and so the inner merge predicate should not refer to them. M_c in Example 4.18 is clearly **RDM**, for example, since it refers only to $0.tt$ and $1.tt$, thus satisfying $\mathbf{R2}_m$, and does not mention *ok* or *wait* in any way.

Using this definition of healthy reactive inner merges, we can then prove the following closure theorem for parallel composition:

Theorem 6.25 (Parallel Composition Closure)**.** *If P and Q are **SRD** healthy, and M is **RDM** healthy, then $P \parallel_M Q$ is **NSRD** healthy.*

As seen in Definition 6.21, parallel composition, like sequential composition, is not defined explicitly as a reactive design using the contract syntax. However, in order to calculate the meaning of a parallel reactive program it is necessary to know how to calculate a contract form for it. By Theorem 6.25 we know that parallel composition is a normal stateful reactive design (**NSRD**), and therefore we can invoke Theorem 6.10 to split it into a reactive design triple, toward substantiation of Theorem 4.19. It then suffices to calculate its pre-, peri-, and postcondition. In order to do this, we need to characterise interference between parallel processes using a notion of weakest rely condition. This is the weakest context under which two parallel composed processes do not violate one another's assumptions.

Definition 6.26 (Weakest Rely Condition)**.**

$$P \ \mathbf{wr}_M \ Q \triangleq \neg_r \left((\neg_r Q) \parallel_{M;\mathbf{true}_r} P \right)$$

The operator $P \ \mathbf{wr}_M \ Q$ is essentially the concurrent case of the reactive weakest precondition $\mathbf{wlp}_r$ (cf. Definition 3.15). It represents the weakest context where reactive relation P does not lead to the violation of reactive condition Q. We achieve this by first merging the possible traces of negated Q with those of P using the merge predicate M composed with $\mathbf{true}_r$. This determines all the behaviours permitted by merge predicate M that are enabled by P and yet violate Q. The composition with $\mathbf{true}_r$ ensures that resulting reactive relation is extension closed. We then negate the resulting relation to obtain the overall reactive precondition.

We can show that the weakest rely condition constructor forms a reactive condition, which is essential to our using it in the reactive contract precondition:

Theorem 6.27 (Weakest Rely Condition forms a Reactive Condition)**.** *If P and Q are **RR** healthy, and M is **RDM** healthy, then $P \ \mathbf{wr}_M \ Q$ is a reactive condition.*

Proof. By Theorem 6.23 we can show that $P\ \mathbf{wr}_M\ Q$ is **R1** and $\mathbf{R2}_c$. It therefore suffices to show that it is also **RC1** healthy, which we do below.

$$\begin{aligned}
\mathbf{RC1}(P\ \mathbf{wr}_M\ Q) &= \mathbf{RC1}(\neg_r((\neg_r P) \parallel_{\mathrm{M};\mathbf{true}_r} Q)) && [\mathbf{wr}_M \text{ definition}]\\
&= \neg_r((\neg_r \neg_r((\neg_r P) \parallel_{\mathrm{M};\mathbf{true}_r} Q)) ; \mathbf{true}_r) && [\mathbf{RC1} \text{ definition}]\\
&= \neg_r(((\neg_r P) \parallel_{\mathrm{M};\mathbf{true}_r} Q) ; \mathbf{true}_r) && [\text{predicate calculus}]\\
&= \neg_r(((\neg_r P) \parallel_{\mathrm{M}} Q) ; \mathbf{true}_r ; \mathbf{true}_r) && [\parallel_{\mathrm{M}} \text{ definition}]\\
&= \neg_r(((\neg_r P) \parallel_{\mathrm{M}} Q) ; \mathbf{true}_r) && [\text{relational calculus}]\\
&= \neg_r((\neg_r P) \parallel_{\mathrm{M};\mathbf{true}_r} Q) && [\parallel_{\mathrm{M}} \text{ definition}]\\
&= P\ \mathbf{wr}_M\ Q && [\mathbf{wr}_M \text{ definition}]
\end{aligned}$$

This essentially follows due the imposition of extension closure in the merge predicate. □

We can now use the weakest rely condition to calculate the precondition for parallel composition. We enumerate all the ways that divergence could arise from the composition, and request that this not happen using weakest rely conditions.

Theorem 6.28 (Parallel Precondition)**.**

$$pre_{\mathbf{R}}(P \parallel^M_{\mathbf{R}} Q) = \left(\begin{array}{l} pre_{\mathbf{R}}(P)\ \mathbf{wr}_M\ peri_{\mathbf{R}}(Q) \wedge pre_{\mathbf{R}}(P)\ \mathbf{wr}_M\ post_{\mathbf{R}}(Q) \wedge \\ pre_{\mathbf{R}}(Q)\ \mathbf{wr}_M\ peri_{\mathbf{R}}(P) \wedge pre_{\mathbf{R}}(Q)\ \mathbf{wr}_M\ post_{\mathbf{R}}(P) \end{array}\right)$$

Divergence can arise in four possible possible ways: when an intermediate or final observation of Q leads to a state where the precondition of P is violated, and the converse situation for P and Q.

The theorem for calculating the pericondition of parallel composition requires that we merge the traces, but not the state variables as these are concealed in intermediate observations. So, we define the following derived parallel composition operator.

Definition 6.29 (Intermediate Merge)**.** $P \parallel^M_{\mathbf{E}} Q \triangleq P \parallel_{\exists\, st' \bullet M} Q$

$P \parallel^M_{\mathbf{E}} Q$ merges the traces of P and Q, whilst hiding the merged post-state using an existential quantifier. It is used in the following calculation of parallel pericondition.

Theorem 6.30 (Parallel Pericondition and Postcondition)**.**

$$peri_{\mathbf{R}}(P \parallel^M_{\mathbf{R}} Q) = pre_{\mathbf{R}}(P \parallel^M_{\mathbf{R}} Q) \Rightarrow_r \left(\begin{array}{l} peri_{\mathbf{R}}(P) \parallel^M_{\mathbf{E}} peri_{\mathbf{R}}(Q) \vee \\ post_{\mathbf{R}}(P) \parallel^M_{\mathbf{E}} peri_{\mathbf{R}}(Q) \vee \\ peri_{\mathbf{R}}(P) \parallel^M_{\mathbf{E}} post_{\mathbf{R}}(Q) \end{array}\right)$$

$$post_{\mathbf{R}}(P \parallel^M_{\mathbf{R}} Q) = pre_{\mathbf{R}}(P \parallel^M_{\mathbf{R}} Q) \Rightarrow_r (post_{\mathbf{R}}(P) \parallel_M post_{\mathbf{R}}(Q))$$

The merge predicate calculates the value of $wait'$ from the disjunction of $0.wait$ and $1.wait$, and so the overall observation of a parallel composition is intermediate if either P or Q is. The parallel postcondition, for the same reason, requires that both P and Q have reached their final states. Unlike for the pericondition, the normal parallel by merge operator is used, since the state is no longer concealed.

We have now shown the laws for calculating the pre-, peri-, and postconditions of parallel reactive contracts. Combining Theorems 6.28 and 6.30 with Theorem 6.2 allows us to finally substantiate Theorem 4.19. This then completes our preliminary results on parallel composition for reactive designs.

In this section, we have laid the foundations for our theory of reactive contracts, considering its foundations, links to other theories, and recursion and parallel composition. We next consider its mechanisation in Isabelle/HOL, which allowed us to mechanically prove the all of the laws previously presented.

7. Mechanised Proof with Isabelle/UTP

In this section, we give an overview of the mechanisation work in Isabelle/UTP [19, 63], which has supported the development of our theory of reactive contracts, and has allowed us to turn these theories into a prototype verification tool.

The theory hierarchy described in Sections 3, 4, and 6, has been developed almost entirely mechanically. Whilst we have planned the main high-level theorems using pen and paper prior to mechanisation, all the low-level lemmas have been proved automatically using the proof tactics we have developed for UTP. This has significantly helped our theory engineering. We have been able to progress at a speedy pace, with proof support aiding the exploration of theory properties and discovery of missing or implicit assumptions in our theorems. This has had a positive impact on the precision of our theories by preventing formalisation gaps in proofs, and improves our overall confidence in the correctness of our theorems. This is particularly the case due to the LCF architecture of Isabelle/HOL, which ensures that our theories are all sound with respect to the axioms of higher order logic. The most difficult technical challenge of this work has been mechanising the parallel-by-merge construct [33], which requires that we perform non-trivial manipulations on alphabets; for details please see the accompanying mechanised artefacts in Section 6.6.

We have also used our theory hierarchy to produce a refinement-based verification tactic for reactive contracts. This, in particular, allows us to harness a number of existing automated proof tactics and strategies [6] in verifying reactive programs, such as auto, which automates logical deduction, and sledgehammer, which integrates external automated theorems and SMT solvers.

Each of the rules for contract calculation given in Section 4 have been proved correct with respect to our underlying operator definitions of relational calculus. These theorems then act as inputs to the proof tactic, which proceeds by first calculating the contract for a reactive program, and then attempting to automatically prove that it refines a contractual specification. A proof goal about a potentially complex reactive program can be effectively reduced into three simpler reactive relations characterising assumptions, intermediate behaviours, and final behaviours, which can be more readily discharged and so favours greater automation. This is the approach we have prototyped in our verification example in Section 5.

The initial particular problem to be solved is bridging the gap between UTP predicates and HOL predicates, which are distinct types, albeit strongly related. In particular, the majority of predicate and relational operators of Isabelle/UTP are defined, via the lifting package [34], in terms of operators of HOL. The major difference is the addition of lenses [19], which accounts for the state variables and alphabet that is not explicitly present in HOL predicates. Nevertheless, because UTP predicates are effectively an enrichment of HOL predicates, laws and tactics applicable to the former often can be adapted to the latter. This, therefore, prevents the need to "reinvent the wheel" when conducting automated reasoning in Isabelle/UTP.

Our approach to reasoning about alphabetised predicates and relations therefore extends the approach we have first described in [18]. The basic approach is two step: firstly apply the transfer tactic of the lifting package [34] to interpret a UTP predicate as a HOL predicate, and secondly apply Isabelle/HOL's built in automated reasoning tactics. We have extended this approach to provide explicit support for lenses, such that each UTP variable can be collapsed and rewritten to a HOL variable with a similar name. Effectively this means that reasoning about UTP predicates and relations can often be entirely reduced to reasoning about HOL predicates, which improves proof automation.

In particular, for relations, proof reduces to a satisfiability problem, since sequential composition boils down to existential quantification, substitution, and conjunction. We provide the rel-auto tactic, which uses this approach, combined with HOL's auto tactic, to deal with conjectures in the predicate and relational calculus. It can discharge the majority of core Isabelle/UTP laws automatically, though in some cases a subsequent call to automated theorem provers using sledgehammer is also necessary.

Reasoning about predicates and relations in this way is very powerful and efficient, but it can be further optimised by making use of specific patterns imposed by a UTP theory. Each reactive contract is written in terms of the $\boldsymbol{R}_s$ healthiness condition, and the design turnstile, which adds a large relational overhead particularly when dealing with large composite definitions. As we indicated, this machinery can be collapsed using the theorems of Sections 4 and 6, so that proof is performed over the pre-, peri-, and postconditions, which are all simply reactive relations. Therefore, we have created a series of additional tactics to automate

this process, and in particular the tactics rdes-refine and rdes-eq that employ the calculational laws, and Theorems 4.2 and 4.3, to prove (or refute) refinement conjectures and equality conjectures.

In particular, this tactic can be used to the automate contract verification problem highlighted in Section 4, which takes the form of

$$[\, P_1 \vdash P_2 \mid P_3 \,] \sqsubseteq Q$$

and states that some reactive program Q, defined using operators of the object language, satisfies the given contractual specification.

This rdes-refine and rdes-eq tactics are generic, in that they can be applied to prove conjectures of any reactive contract based language, so long as denotational definitions and theorems are available that show how to expand the operators to contracts. The proof process has three stages:

1. **Calculate and Simplify Reactive Design Contracts**. The operators of the object language, for example *Circus*, are first expanded into their contractual specifications. For a large reactive program this could result in three very large predicates, which are addressed separately in the next stage. This stage requires that appropriate calculation theorems have been proved and are already available. The algebraic laws of reactive designs, for example Theorem 4.13, are applied to convert a composite reactive contract into a monolithic one of the syntactic form $[\, P_1 \vdash P_2 \mid P_3 \,]$. The application of these theorems and the associated simplifications can be performed separately by a tactic called rdes-simp.
2. **Apply the Refinement Theorem**. Theorem 4.2 is applied to break the refinement conjecture into three proof obligations about the pre-, peri-, and postconditions. As usual, we require that the precondition is weakened, and the peri- and postcondition is strengthened.
3. **Apply Relational Transfer and Reasoning Tactics**. The three proof obligations are purely relational, do not refer to observational variables *ok* and *wait*, and thus are relatively lightweight in nature. Thus we can now apply rel-auto, and potentially other Isabelle/HOL tactics, to try and complete the proof. Of course, we can also at this point try to generate a counterexample, which can be traced back to a fault in the pre-, peri-, or postcondition.

We have found this approach extremely useful for automated reasoning, and it has been used to validate all the example calculations, in particular those related to the cash-card example[3] from Section 5. The tactic is also very general, in that any language whose operators can be expressed using contracts, automatically receives proof support. In this sense, whilst so far we have applied it only to *Circus*, it can also be applied to a variety of related languages [58, 60, 24, 64], and therefore Isabelle/UTP constitutes a generic toolkit for verification tools.

8. Related Work

Meyer [42] coined the term "design-by-contract" when arguing the need for precisely specified assertions of a program's behaviour to ensure reliability of component-based software systems. These assertions come in three forms: preconditions, postconditions, and invariants, which are used to annotate methods and attributes within classes in his object-oriented programming language, Eiffel [41].

Meyer's work has its foundation in that of Floyd [15] and Hoare [29] on proving correctness of programs in terms of the Hoare triple $\{\, p \,\}\, Q \,\{\, r \,\}$. This asserts that if program Q is started in a state satisfying predicate p then, provided Q terminates, the final state satisfies predicate r. Effectively, the Hoare triple sets up a contract on state variables for Q that mandates a particular relationship between inputs and outputs.

Refinement calculi [2, 43, 44] promote such pre/postcondition specifications to statements of the language itself. A programming language is extended with an abstract specification statement, such as $w{:}[pre, post]$ [43], where *pre* is the precondition, *post* the postcondition, and w the "frame" – that is, the set of variables that are allowed to change. A specification statement can be transformed into a block

[3] See https://github.com/isabelle-utp/utp-main/blob/master/tutorial/utp_csp_mini_mondex.thy for our complete mechanised theory

of executable code through a series of correctness-preserving refinements that divide the specification into constituent parts, and eventually introduce atomic commands. Morgan [43] and Back [2] both refer to such specification statements as "contracts" between the specifier and the implementer, who is allowed to weaken the precondition and strengthen the postcondition. This makes the implementation more deterministic and thus predictable, whilst fulfilling the original contract that specifies it.

Hoare logic and refinement calculus can be unified either algebraically, using Kleene algebras [1, 20], or else denotationally [9, 26, 46], using the UTP theory of designs [33, 9], which is foundational for our reactive contracts. The unifying nature of UTP is achieved through encoding all conceivable programs, not as bespoke mathematical structures, but as elements of the alphabetised predicate calculus.

Benveniste et al. [4, 5] provide a formal contract framework that has been very influential in the area of contract-based design [50, 3, 57]. They give a denotational framework that describes contracts as pairs of predicates $C \triangleq (A, G)$ where A is the assumption, and G is the guarantee. A and G both characterise the set of program variable traces that prescribe valid behaviours of the context and given component, respectively. They define a notion of refinement $C_1 \preceq C_2$ called "dominance" — named to distinguish it from refinement to implementation, but otherwise technically the same — that requires that the assumption be weakened and the guarantee be strengthened. They also define an algebra for contracts including parallel composition $C_1 \parallel C_2$, disjunction $C_1 \sqcap C_2$, and conjunction $C_1 \sqcup C_2$. Dominance corresponds to the universal notion of refinement in UTP: $P \sqsubseteq Q$.

We note that Benveniste's theory has a striking resemblance to the theory of designs, although designs do not explicitly account for traces. Designs effectively encode the pair of assumption-guarantee predicates into a single relation via the ok and ok' variables. Thus, many of the laws in [4] have very similar design theorems, in particular those for the lattice operators. This implies that derivatives of the UTP design theory, including ours, rightly fit into their general contract framework [4, 5], and the resulting work stream [50, 3, 57].

Works building on Benveniste's framework [11, 57] allow step-wise refinement, and verification based on temporal logic properties. Previous works [38, 37] indicate that such temporal logics can be readily characterised in our domain. Our extensible semantic model is more expressive than a purely trace-based model, and we are limited to neither trace nor failures-divergences refinement. Our contract model also explicitly distinguishes terminating behaviours, and thus supports sequential as well as parallel composition. We also handle state variables, so that they need not be modelled as a sequence of updates in the trace.

On its own, the UTP theory of designs only accounts for imperative behaviour and thus various specialisations have been made for other paradigms. Notable is the theory of reactive designs [10, 46], which adds program histories represented by traces, alongside an account of non-terminating behaviours. Effectively, the theory of reactive designs combines the theory of designs with the theory of reactive processes. This integration of stateful and reactive behaviour enabled Oliveira et al. [47, 46] to give a UTP semantics to the *Circus* language, which combines CSP [31, 48], the Z notation [54], and guarded command language [14].

Reactive designs arise as a direct consequence of a result due to Hoare and He [33, Theorem 8.2.2], and was named as such by Cavalcanti and Woodcock [10] who highlighted a normal form: $\mathbf{R}(P_1 \vdash P_2)$, which is intuitively a design made reactive with assumption P_1 and guarantee P_2.. They can be used as a semantic domain for reactive languages with assumptions and guarantees, and is the foundational idea behind our theory. The innovation of our work is to subdivide P_2 into two parts, for intermediate and final observations, and prove a large set of calculational theorems that support automated verification. Our reactive designs theory also implements a contract framework with a generic trace model, based on trace algebra [16], similar to [4, 57]. However, we embed the traces directly into the design predicate through **R1** and $\mathbf{R2}_c$. This means that the relational calculus operators are directly applicable without redefinition, and substantiates the unifying nature of our theory.

Our work also overcomes a number of technical limitations in the existing reactive designs theory [47, 46]. Firstly, we explicitly characterise internal state with an observational variable, and require that the after state is invisible in intermediate states. This enables correct interaction with stateful behaviours for operators like external choice [7]. Secondly, we relax a restriction of [47, 46] on reactive design assumptions, which in their previous work cannot mention traces and thus are unsuitable to represent reactive assumptions.

Butterfield et al. [7] study state visibility in the theory of reactive designs, noting that the observation of program state is miraculous while waiting for an external choice to be resolved for **R3** predicates. This is

because in the original account [33] **R3** leaves intermediate valuations of state variables observable, however, CSP's external choice operator conjoins the intermediate observations since, until an event occurs, all such behaviours have potential to resolve. Therefore, inconsistent intermediate values for state variables lead to miraculous behaviour since, for example, $(x := 1 \land x := 2) = \textbf{\textit{false}}$. Thus, in [7], **R3** is replaced by a variant called $\textbf{R3}_h$, which causes internal state updates to be abstracted in intermediate observations. They note that this curtails the ability to interrupt an action and pass its intermediate state valuation onto the interrupting action [7, 40]. Nevertheless, we adopt $\textbf{R3}_h$ in our work, as it leads to a simpler handling of state and has several advantages for automated reasoning. We also note that it is not difficult to adapt our mechanisation to a version of reactive contracts based on **R3** instead.

Reactive designs have been further extended to account for discrete time in the UTP theories for the languages *CircusTime* [53, 58] and *CML* [60, 8]. They augment the trace with time events, albeit in slightly different ways. Moreover, a theory similar to that of reactive designs has been used to give a denotational semantics to a hybrid version of CSP [24] and the dynamic systems modelling language Modelica [17]. In that work, each continuous variable has a continuous-time trace. Our reactive contract theory unifies these discrete and hybrid theories. Canham and Woodcock [8] explicitly distinguish non-terminating and terminating behaviours in the postcondition, resulting in a triple of the form $\textbf{R}(P \vdash Q_1 \diamond Q_2)$ that we generalise and adopt. A key result of our trace algebraic foundations is that all the aforementioned extensions can be unified by our UTP theory.

Another extension of the theory of designs to account for contractual behaviour is found in rCOS [25, 26, 64], a refinement calculus and UTP theory for object-oriented and component systems. A contract in rCOS is expressed as a quadruple consisting of (1) the component's interface, that is, the types of its attributes and methods; (2) an initialiser or constructor for the component; (3) a collection of specifications for the methods; and (4) the valid traces of calls to methods that can be made. The initialiser and methods are specified using a form of UTP design also called a "reactive design" [25]. A contract's dynamic behaviour is given a semantics in the style of CSP's failures-divergences, with method names as events.

In rCOS [25, 26, 64], the valid traces are used to protect the method calls that can deadlock if their preconditions are violated, and thus act as a form of environmental assumption. Their notion of reactive design uses only the healthiness condition **R3**. Moreover, their contract notion is not embedded in UTP's relational calculus, but are explicit quadruples. This means they cannot reuse operators like sequential composition and nondeterministic choice, hampering composition with other UTP theories. Nevertheless, their language, being based on a CSP-style failures-divergence semantics augmented with stateful operations with design semantics, can be readily embedded into *Circus*, and hence our reactive design theory.

The rely-guarantee technique of Jones et al. [35, 36, 22] provides contract-based reasoning for programs with shared variable concurrency. A rely-guarantee quintuple $\{p, r\}c\{g, q\}$ states that, assuming the initial state satisfies predicate p, and each atomic step of the environment satisfies relation r – the rely condition – then program c terminates in a state satisfying q and guarantees g at each of its atomic steps. The rely condition specifies how much interference the process must tolerate from its siblings, and in return, the guarantee condition puts a limit on the interference the process can cause through its manipulation of the shared variables. A co-existence proof obligation requires that each parallel program is guaranteeing enough for what the other relies upon. Another proof obligation is that the postcondition is implemented by the steps taken by the two programs and their environment.

It might seem that the rely-guarantee technique, being based on shared variables, is not directly applicable to languages like *Circus*, which is based on a communication model with no variable sharing. However, our contracts are based on an abstract notion of trace that can encompass shared variable updates as events. In our reactive design contracts, the precondition can refer to both the trace and initial value of state variables, and therefore encompasses both a precondition p and rely condition r. Our pericondition broadly corresponds to a guarantee condition g, since it states what must be true at each intermediate step. However, our postcondition can also refer to the completed trace at the point of termination, along with the final valuation of state variables, which may afford us additional expressivity.

Comparison with rely-guarantee becomes easier if we consider a more abstract algebraic level. Hayes et al. have explored algebras to characterise rely-guarantee reasoning [22, 23, 12], which they call Concurrent Refinement Algebra (CRA). Like the theory of designs, CRA is based on a refinement lattice with nonde-

terministic choice. It also provides operators for sequential composition, parallel composition, and iteration. In [23], they identify a subalgebra of CRA that corresponds to atomic steps. In addition to characterising atomic state updates, the algebra is also used to characterise CSP-style events, which more readily compares with our work.

Hayes' semantics for CRA is based on Aczel traces [12], which explicitly distinguish passive environmental events and active program events. The former is used to encode that a process is willing to allow a particular event to occur, whilst the latter expresses active engagement. The resulting semantic model is very different to the standard failures-divergence model of CSP and *Circus*, in that an event prefix $a \rightarrow P$ explicitly prescribes that all other possible events can occur before a occurs in the trace, by using environment events. This allows a unified notion of parallel composition, which simply reduces the possible environmental and program events.

Their model does not currently account for external choice, though this should be representable with the choice being resolved only through a program event, and not environment events. Nevertheless, the relation to the standard model of CSP remains to be seen and a soundness result would be necessary to use their semantic model as a drop-in replacement. On the other hand, our semantic model is sufficiently general to encompass finite Aczel traces, though for this paper we focus on (though do not mandate) the standard failures-divergences model in examples.

9. Conclusions

We have developed a comprehensive and generalised UTP theory of reactive designs that can be applied to contract-based modelling and automated verification of sequential and parallel reactive systems. Our language of contracts allows to both compose specifications for reactive programs, and also give denotational semantics to reactive languages in the programs-as-predicates approach [27]. At a specification level, our contracts allow us to restrict permissible behaviours of the environment using preconditions, and specify possible behaviours in intermediate and final observations using peri- and postconditions. This is supported by a theory of reactive relations (Section 3), with which we are able to specify predicates and relations that refer to both state variable valuations and the trace of interactions. This theory provides an abstract algebraic account of traces, and therefore supports both discrete event sequences and also piecewise continuous functions. The theory is therefore applicable to both discrete time and hybrid systems.

From our theory, we derived a set of calculational laws in Section 4 for reactive contract composition using operators like sequential composition, non-deterministic choice, recursion, and parallel composition. We also proved theorems that demonstrate refinement and equivalence between two contracts, and use these to develop a prototype procedure for automated verification of reactive programs by calculation. This procedure was applied to a small cash-card verification example in Section 5 based in the *Circus* [61] language. The underlying healthiness conditions for our UTP theory have also been formulated in Section 6, and a number of key theorems for tail recursion and parallel composition have been expounded. Our contract theory is practically supported by a mechanisation in Isabelle/UTP, which provides both confidence in the proven theorems, and also automated verification facilities through a number of proof tactics. The vast majority of definitions and theorems from Section 3 onwards in this paper are novel; they exceptions are Theorems 6.17, 6.19, and 6.20.

There are a number of important areas for future work. In the area of assume-guarantee reasoning, there are a number of works that have much in common with our framework [4, 5, 57], particularly at the level of the fundamental UTP theory of designs. In the future, it would be interesting to fully explore these links, for example, through formalisation of their theories in UTP and formation of suitable Galois connections [33], especially with recent work on hybrid system contracts [57]. Such a unification could also consider formalisation of the Aczel trace model [12], a semantic model that facilitates event frames [23], which distinguish events *engaged in* from those simply permitted by a system constituent. This in turn could permit a much simpler, but no less expressive, definition of parallel composition provided by CSP and *Circus* and thus improve support for compositional reasoning with contracts.

Also with respect to parallel composition, the weakest-rely-condition calculus ($\boldsymbol{wr}_M$) will be further developed to capture interference between concurrent processes, and establish concrete links with rely-

guarantee algebra [22] and concurrent Kleene algebra [23]. There is also need for a mechanised refinement calculus, to support step-wise development of reactive programs. This should, in particular, support laws for parallel composition that facilitate compositional reasoning through distributing invariants, which can draw on previous work with *Circus* [47].

With respect to recursion, there is also potential future work. In this paper we have considered calculation of contracts for tail recursive programs utilising Hoare and He's guardedness theorem [33], and Kleene's fixed point theorem [39]. Whilst this encompasses a significant number of reactive programs, there is also potential for more general recursion schemes. For example, the *Circus* action

$$\mu\, X \bullet (a \rightarrow P \,;\, X) \,\Box\, b \rightarrow Q$$

which expresses a recursive body P guarded by event a, with escape event b leading to Q, is clearly productive and guarded, but does not fit our tail recursive pattern. Therefore, more general patterns could be identified and calculational laws proved utilising Theorems 4.13.6 and 6.17, and reusing our notion of productivity. One can potentially go further and consider non-continuous recursion schemes to which Kleene's fixed-point theorem is not applicable anymore, and can build on a further theorem by Hoare and He for calculating general recursive UTP designs [33, theorem 3.1.6, page 81].

In a different axis, we have yet to fully explore the use of our theory in the context of continuous-time trace models. Previously, we have used such a model to give a UTP semantics to the dynamical systems modelling language Modelica [17], but in a setting without formal contracts. Thus, in the future, we could consider a contract language for concurrent hybrid systems using the infrastructure present in this paper, together with a number of specialisations. We already have some preliminary results on modelling block-based control law diagrams using hybrid reactive contracts, and hope to publish this work in the near future.

Finally, more experience is needed in the use of our automated verification tool in order to explore applications from a variety of domains. We would thus like to complete our mechanisation of *Circus*, and apply it to verify a more substantial case study. Clearly, this would rely upon the availability of parallel reasoning facilities highlighted above. Moreover, verification tools for other UTP-based contract languages like *CML* will be developed. We can then start to gather results about the efficiency of our tool, and begin to work on improvements towards and applicability to large and realistic industrial case studies.

Acknowledgements

This research is funded by the CyPhyAssure project[4], EPSRC grant EP/S001190/1, the RoboCalc project[5], EPSRC grant EP/M025756/1, and the EU Horizon 2020 project "INTO-CPS", grant agreement 644047 [6]. We would like to thank our colleagues on this project, particularly those from Peter Gorm Larsen's group at Århus University, for their collaboration over the past three years, without which this work could not have come to fruition. We also thank the anonymous reviewers of this article, whose suggestions have greatly improved the presentation of our work.

[1] Armstrong, A., Gomes, V., Struth, G.: Building program construction and verification tools from algebraic principles. Formal Aspects of Computing **28**(2), 265–293 (2015)
[2] Back, R.J., Wright, J.: Refinement Calculus: A Systematic Introduction. Springer (1998)
[3] Bauer, S., David, A., Hennicker, R., Larsen, K., Legay, A., Nyman, U., Wasowski, A.: Moving from specifications to contracts in component-based design. In: Proc. 15th Intl. Conf. on Fundamental Approaches to Software Engineering (FASE), *LNCS*, vol. 7212, pp. 43–58. Springer (2012)
[4] Benveniste, A., Caillaud, B., Ferrari, A., Mangeruca, L., Passerone, R., Sofronis, C.: Multiple viewpoint contract-based specification and design. In: 6th Intl. Symp. on Formal Methods for Components and Objects (FMCO), *LNCS*, vol. 5382, pp. 200–225. Springer (2007)
[5] Benvenuti, L., Ferrari, A., Mangeruca, L., Mazzi, E., Passerone, R., Sofronis, C.: A contract-based formalism for the specification of heterogeneous systems. In: Proc. Forum on Specification, Verification, and Design Languages (FDL), pp. 142–147 (2008)

[4]CyPhyAssure Project: https://www.cs.york.ac.uk/circus/CyPhyAssure/
[5]RoboCalc Project: https://www.cs.york.ac.uk/circus/RoboCalc/
[6]INTO-CPS Project Website: http://projects.au.dk/into-cps/.

[6] Blanchette, J.C., Bulwahn, L., Nipkow, T.: Automatic proof and disproof in Isabelle/HOL. In: FroCoS, *LNCS*, vol. 6989, pp. 12–27. Springer (2011)
[7] Butterfield, A., Gancarski, P., Woodcock, J.: State visibility and communication in unifying theories of programming. Theoretical Aspects of Software Engineering **0**, 47–54 (2009)
[8] Canham, S., Woodcock, J.: Three approaches to timed external choice in UTP. In: Unifying Theories of Programming, *LNCS*, vol. 8963, pp. 1–20. Springer (2015)
[9] Cavalcanti, A., Woodcock, J.: A tutorial introduction to designs in unifying theories of programming. In: Proc. 4th Intl. Conf. on Integrated Formal Methods (IFM), *LNCS*, vol. 2999, pp. 40–66. Springer (2004)
[10] Cavalcanti, A., Woodcock, J.: A tutorial introduction to CSP in unifying theories of programming. In: Refinement Techniques in Software Engineering, *LNCS*, vol. 3167, pp. 220–268. Springer (2006)
[11] Cimatti, A., Tonetta, S.: A property-based proof system for contract-based design. In: 38th. Conf. on Software Engineering and Advanced Applications (SEAA), pp. 21–28. IEEE (2012)
[12] Colvin, R.J., Hayes, I.J., Meinicke, L.A.: Designing a semantic model for a wide-spectrum language with concurrency. Formal Aspects of Computing (2017)
[13] Coquand, T.: Infinite objects in type theory. In: Proc. 1st Intl. Workshop on Types for Proofs and Programs, *LNCS*, vol. 806, pp. 62–78. Springer (1993)
[14] Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Communications of the ACM **18**(8), 453–457 (1975)
[15] Floyd, R.W.: Assigning meanings to programs. In: Proc. Symp. in Applied Science, *Mathematical Aspects of Computer Science*, vol. 19, pp. 19–32. American Mathematical Society (1967)
[16] Foster, S., Cavalcanti, A., Woodcock, J., Zeyda, F.: Unifying theories of time with generalised reactive processes. Information Processing Letters **135**, 47–52 (2018)
[17] Foster, S., Thiele, B., Cavalcanti, A., Woodcock, J.: Towards a UTP semantics for Modelica. In: Proc. 6th Intl. Symp. on Unifying Theories of Programming, *LNCS*, vol. 10134. Springer (2016)
[18] Foster, S., Zeyda, F., Woodcock, J.: Isabelle/UTP: A mechanised theory engineering framework. In: UTP, *LNCS*, vol. 8963, pp. 21–41. Springer (2014)
[19] Foster, S., Zeyda, F., Woodcock, J.: Unifying heterogeneous state-spaces with lenses. In: Proc. 13th Intl. Conf. on Theoretical Aspects of Computing (ICTAC), *LNCS*, vol. 9965. Springer (2016)
[20] Gomes, V.B.F., Struth, G.: Modal kleene algebra applied to program correctness. In: 21st. Intl. Symp. on Formal Methods (FM), *LNCS*, vol. 9995, pp. 310–325. Springer (2016)
[21] Guttman, W., Möller, B.: Normal design algebra. Journal of Logic and Algebraic Programming **79**(2), 144–173 (2010)
[22] Hayes, I.J.: Generalised rely-guarantee concurrency: an algebraic foundation. Formal Aspects of Computing **28**(6), 1057–1078 (2016)
[23] Hayes, I.J., Colvin, R.J., Meinecke, L.A., Winter, K., Velykis, A.: An algebra of synchronous atomic steps. In: Proc. 21st Intl. Symp. on Formal Methods (FM), *LNCS*, vol. 9995, pp. 352–369. Springer (2016)
[24] He, J.: From CSP to hybrid systems. In: A.W. Roscoe (ed.) A classical mind: essays in honour of C. A. R. Hoare, pp. 171–189. Prentice Hall (1994)
[25] He, J., Xiaoshan, L., Zhiming, L.: Component-based software engineering: The need to link methods and their theories. In: Proc. 2nd Intl. Colloq. on Theoretical Aspects of Computing (ICTAC), *LNCS*, vol. 3722, pp. 70–95. Springer (2005)
[26] He, J., Xiaoshan, L., Zhiming, L.: rCOS: A refinement calculus of object systems. Theoretical Computer Science **365**, 109–142 (2006)
[27] Hehner, E.C.R.: A Practical Theory of Programming. Springer (1993)
[28] Henkin, L., Monk, J., Tarski, A.: Cylindric Algebras, Part I. North-Holland (1971)
[29] Hoare, C.A.R.: An axiomatic basis for computer programming. Communcations of the ACM **12**(10), 576–580 (1969)
[30] Hoare, C.A.R.: Some properties of predicate transformers. Journal of the ACM **25**(3), 461–480 (1978)
[31] Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall (1985)
[32] Hoare, C.A.R., Hayes, I., He, J., Morgan, C., Roscoe, A., Sanders, J., Sørensen, I., Spivey, J., Sufrin, B.: The laws of programming. Communications of the ACM **30**(8), 672–687 (1987)
[33] Hoare, C.A.R., He, J.: Unifying Theories of Programming. Prentice-Hall (1998)
[34] Huffman, B., Kunčar, O.: Lifting and transfer: A modular design for quotients in Isabelle/HOL. In: CPP, *LNCS*, vol. 8307, pp. 131–146. Springer (2013)
[35] Jones, C.B.: Development methods for computer programs including a notion of interference. Ph.D. thesis, Oxford University (1981)
[36] Jones, C.B.: Wanted: a compositional apporach to concurrency. In: Programming Methodology, Monographs in Computer Science, pp. 5–15. Springer (2003)
[37] von Karger, B.: A calculational approach to reactive systems. Science of Computer Programming **37**, 139–161 (2000)
[38] von Karger, B., Hoare, C.A.R.: Sequential calculus. Information Processing Letters **53**, 123–130 (1995)
[39] Lassez, J.L., Nguyen, V.L., Sonenberg, E.A.: Fixed point theorems and semantics: a folk tale. Information Processing Letters **14**(3), 112–116 (1982)
[40] McEwan, A.: Concurrent program development in Circus. Ph.D. thesis, Oxford University (2006)
[41] Meyer, B.: Eiffel: A language and environment for software engineering. Journal of Systems and Software **8**(3), 199–246 (1988)
[42] Meyer, B.: Applying "design by contract". IEEE Computer **25**(10), 40–51 (1992)
[43] Morgan, C.: Programming from Specifications. Prentice-Hall (1996)
[44] Morris, J.M.: A theoretical basis for stepwise refinement and the programming calculus. Science of Computer Programming

9(3), 287–306 (1987)
[45] Nipkow, T., Wenzel, M., Paulson, L.C.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic, *LNCS*, vol. 2283. Springer (2002)
[46] Oliveira, M., Cavalcanti, A., Woodcock, J.: A UTP semantics for Circus. Formal Aspects of Computing **21**, 3–32 (2009)
[47] Oliveira, M.V.M.: Formal Derivation of State-Rich Reactive Programs using *Circus*. Ph.D. thesis, Department of Computer Science - University of York, UK (2006). YCST-2006-02
[48] Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice-Hall (2005)
[49] Roscoe, A.W., Hoare, C.A.R.: The laws of Occam programming. Theoretical Computer Science **60**(2), 177–229 (1988)
[50] Sangiovanni-Vincentelli, A., Damm, W., Passerone, R.: Taming dr. frankenstein: Contract-based design for cyber-physical systems. European Journal of Control **3**, 217–238 (2012)
[51] Santos, T., Cavalcanti, A., Sampaio, A.: Object-Orientation in the UTP. In: S. Dunne, B. Stoddart (eds.) UTP 2006: First International Symposium on Unifying Theories of Programming, *LNCS*, vol. 4010, pp. 20–38. Springer-Verlag (2006)
[52] Scott, D.: Continuous lattices. In: Toposes, Algebraic Geometry and Logic, *LNM*, vol. 274, pp. 97–136. Springer (1971)
[53] Sherif, A., Cavalcanti, A., He, J., Sampaio, A.: A process algebraic framework for specification and validation of real-time systems. Formal Aspects of Computing **22**(2), 153–191 (2010)
[54] Spivey, M.: The Z-Notation - A Reference Manual. Prentice Hall, Englewood Cliffs, N. J. (1989)
[55] Tarksi, A.: A lattice-theoretical fixpoint theorem and its applications. Pacific Journal of Mathematics **5**(2), 285–309 (1955)
[56] Tarski, A.: On the calculus of relations. J. Symbolic Logic **6**(3), 73–89 (1941)
[57] Tonetta, S., Cimatti, A.: Contracts-refinement proof system for component-based embedded systems. Science of Computer Programming **97**, 333–348 (2015)
[58] Wei, K., Woodcock, J., Cavalcanti, A.: Circus Time with Reactive Designs. In: Unifying Theories of Programming, *LNCS*, vol. 7681, pp. 68–87. Springer (2013)
[59] Woodcock, J.: The Miracle of reactive programming. In: Proc. 2nd Intl. Symp. on Unifying Theories of Programming (UTP), *LNCS*, vol. 5713, pp. 202–217. Springer (2008)
[60] Woodcock, J.: Engineering UToPiA - Formal Semantics for CML. In: C. Jones, P. Pihlajasaari, J. Sun (eds.) FM 2014: Formal Methods, *Lecture Notes in Computer Science*, vol. 8442, pp. 22–41. Springer International Publishing (2014)
[61] Woodcock, J., Cavalcanti, A.: A concurrent language for refinement. In: A. Butterfield, G. Strong, C. Pahl (eds.) Proc. 5th Irish Workshop on Formal Methods (IWFM), Workshops in Computing. BCS (2001)
[62] Woodcock, J., Cavalcanti, A., Fitzgerald, J., Foster, S., Larsen, P.G.: Contracts in CML. In: ISoLA 2014, Part II, *LNCS*, vol. 8803, pp. 54–73. Springer (2014)
[63] Zeyda, F., Foster, S., Freitas, L.: An axiomatic value model for Isabelle/UTP. In: Proc. 6th Intl. Symp. on Unifying Theories of Programming (UTP), *LNCS*, vol. 10134, pp. 155–175. Springer (2016)
[64] Zhan, N., Kang, E.Y., Liu, Z.: Component publications and compositions. In: 2nd Intl. Symp. on Unifying Theories of Programming (UTP), *LNCS*, vol. 5713, pp. 238–257. Springer (2008)