
Article:
Foster, Simon orcid.org/0000-0002-9889-9514, Ye, Kangfeng, Cavalcanti, Ana orcid.org/0000-0002-0831-1976 et al. (1 more author) (2021) Automated verification of reactive and concurrent programs by calculation. Journal of Logical and Algebraic Methods in Programming. 100681. ISSN 2352-2216

https://doi.org/10.1016/j.jlamp.2021.100681

Automated Verification of Reactive and Concurrent Programs by Calculation

Simon Foster, Kangfeng Ye, Ana Cavalcanti, Jim Woodcock

Abstract

Reactive programs combine traditional sequential programming constructs with primitives to allow communication with other concurrent agents. They are ubiquitous in modern applications, ranging from components systems and web services, to cyber-physical systems and autonomous robots. In this paper, we present an algebraic verification strategy for concurrent reactive programs, with a large or infinite state space. We define novel operators to characterise interactions and state updates, and an associated equational theory. With this we can calculate a reactive program's denotational semantics, and thereby facilitate automated proof. Of note is our reasoning support for iterative programs with reactive invariants, based on Kleene algebra, and for parallel composition. We illustrate our strategy by verifying a reactive buffer. Our laws and strategy are mechanised in Isabelle/UTP, our implementation of Hoare and He's Unifying Theories of Programming (UTP) framework, to provide soundness guarantees and practical verification support.

1. Introduction

Reactive programming [23, 3] is a paradigm that enables effective description of software systems that exhibit both internal sequential behaviour and event-driven interaction with a concurrent party. Reactive programs are ubiquitous in safety-critical systems, and typically have a very large or infinite state space. Though model checking is an invaluable verification technique, it exhibits inherent limitations with state explosion and infinite-state systems that can be overcome by supplementing it with theorem proving.

Previously [14], we have shown how *reactive contracts* support an automated verification technique for reactive programs. Reactive contracts follow the design-by-contract paradigm [32], where programs are accompanied by pre- and postconditions. Reactive programs are often non-terminating and so we also capture intermediate behaviours, where the program has not terminated, but is quiescent and offers opportunities to interact. Our contracts are triples, $[P_1 \vdash P_2 \mid P_3]$, where P_1 is the precondition, P_3 the postcondition, and P_2 the pericondition. P_2 characterises the quiescent observations in terms of the interaction history, and the events enabled at that point. Broadly speaking, our contract theory has its roots in the CSP process algebra [26], and its failures-divergences semantic model [38, 8].

Reactive contracts describe communication and state updates, so P_1, P_2, and P_3 can refer to both a trace history of events and internal program variables. They are, therefore, called "reactive relations": like relations that model sequential programs, they can refer to variables before (x) and later (x') in execution, but also the interaction trace (tt), in both intermediate and final observations.

Verification using contracts employs refinement ($\sqsubseteq$), which requires that an implementation weakens the precondition, and strengthens both the peri- and postcondition when the precondition holds. We employ the "programs-as-predicates" approach [25], where the implementation (Q) is itself denoted as a composition of contracts. Thus, a verification problem, $[P_1 \vdash P_2 \mid P_3] \sqsubseteq Q$, can be solved by calculating a program $[Q_1 \vdash Q_2 \mid Q_3] = Q$, and then discharging three proof obligations: (1) $Q_1 \sqsubseteq P_1$; (2) $P_2 \sqsubseteq (Q_2 \wedge P_1)$; and

*Corresponding author
Email addresses: simon.foster@york.ac.uk (Simon Foster), kangfeng.ye@york.ac.uk (Kangfeng Ye), ana.cavalcanti@york.ac.uk (Ana Cavalcanti), jim.woodcock@york.ac.uk (Jim Woodcock)

(3) $P_3 \sqsubseteq (Q_3 \wedge P_1)$. These can be further decomposed, using relational calculus, to produce verification conditions. In [14] we employ this strategy in an Isabelle/HOL tactic.

In summary, in our approach verification of reactive programs reduces to reasoning about reactive relations. For programs of a significant size, these relations are complex, and so the resulting proof obligations are difficult to discharge using relational calculus. We need, first, abstract patterns so that the relations can be simplified. This necessitates bespoke constructs that allow us to concisely formulate the three parts of a contract: assumptions, quiescent observations, and terminated observations. Second, we need calculational laws to handle iterative programs, which are only partly handled in our previous work [14].

In this paper we present a novel calculus for description, composition, and simplification of reactive relations in the stateful failures-divergences model [38, 28, 35]. We characterise conditions, external interactions, and state updates. An equational theory allows us to reduce pre-, peri-, and postconditions to compositions of the new constructs using operators of Kleene algebra [30] (KA) and utilise KA proof techniques. Our theory is characterised in the Unifying Theories of Programming [28, 8] (UTP) framework. For that, we identify a class of UTP theories that induce KAs, and utilise it in the derivation of calculational laws for iteration. We use our UTP mechanisation, called Isabelle/UTP [13, 20], to implement an automated verification approach for infinite-state reactive programs with rich data structures based on our calculus.

Our framework can be applied to a wide spectrum of reactive programming languages with trace-based semantics, including real-time and hybrid dynamical systems [24, 48, 42]. A particular focus is languages descended from CSP [26, 38]. In this paper, our approach is applied to the *Circus* modelling language [46, 35] which combines state modelling using Z [43] and reactive primitives from CSP [26, 38]. An example application is verification of Simulink block diagrams, to which both *Circus* and hybrid CSP [24] have been succesfully applied [7, 49]. More recently, *Circus* and CSP have been used for verification of a formal state-machine based language for robotic controllers called RoboChart [33, 12].

The paper is structured as follows. §2 outlines preliminary material, including UTP, its mechanisation in Isabelle/UTP, and reactive programs. §3 identifies a class of UTP theories that induce KAs, and applies this class for calculation of iterative contracts. §4 specialises reactive relations with new operators to capture stateful failures-divergences, and derives their equational theory. This allows us to automatically calculate semantics for sequential reactive programs. §5 extends our equational theory with support for calculating external choices, for programs where the environment has control over a decision. We also develop healthiness conditions characterising productivity – a requirement for both algebraic laws of external choice and iteration. §6 extends the strategy with while loops and reactive invariants. §7 encodes parallel composition as a reactive design, and further extends the strategy with calculational laws for concurrent behaviours. With this, we can then calculate semantics for concurrency and communication between reactive processes. §8 demonstrates the resulting proof strategy in a small verification. §9 outlines related work and concludes.

All our theorems, definitions, and proofs have been mechanically verified in Isabelle/UTP, and are documented in a series of technical reports[1] [20, 11, 16, 17]. 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 our Isabelle/UTP GitHub repository[2].

This paper is an extension of [18]. It adds a body of additional theorems in §4 on more specialised healthiness conditions for stateful-failure reactive relations (Theorem 4.3), calculation of iterative reactive relations (Theorem 4.8-(7)), preconditions of reactive contracts (Theorem 4.11), and also extended supporting commentary. Moreover, a substantial new §7 extends the strategy for parallel composition. A number of additional supporting theorems and definitions are also included in the other sections.

2. Preliminaries

This section describes background material relevant for the definition of our new calculus.

[1] For historical reasons, we use the syntax $\boldsymbol{R}_s(P \vdash Q \diamond R)$ in our mechanisation for a contract $[P \vdash Q \mid R]$. The former builds on Hoare and He's original syntax for the theory of designs [28].

[2] Isabelle/UTP repository: https://github.com/isabelle-utp/utp-main. An archive containing all the files for this paper, and instructions on how to load them into Isabelle/HOL, can also be found at http://doi.org/10.5281/zenodo.3541080.

2.1. Unifying Theories of Programming

UTP [28, 8] uses the "programs-as-predicates" approach to encode denotational semantics and facilitate reasoning about programs. It uses the alphabetised relational calculus, which combines predicate calculus operators, such as disjunction ($\vee$), complement ($\neg$), and quantification ($\exists\, x \bullet P(x)$), with relation algebra [44], to denote programs as binary relations between initial variables (x) and their subsequent values (x'). Here, "alphabetised" means that every such relational predicate is accompanied by a set of variables to which the predicate can refer. For example, a program fragment, $x := 1\ ;\ x := x + y$, with two distinct variables, can be modelled by the relational predicate $x' = y{+}1 \wedge y' = y$, with the alphabet $\alpha = \{x, x', y, y'\}$.

In this presentation of the UTP, we first define the set of alphabetised expressions, $[\mathcal{V}, \alpha]\mathit{uexpr}$, which is parametric over $\mathcal{V}$ and α, types that represents the value type and state space. The latter corresponds to the alphabet, with a set of typed variable declarations. Expressions are isomorphic to functions $\alpha \to \mathcal{V}$, which return a value in $\mathcal{V}$ for a given state space. Alphabetised predicates are represented by Boolean expressions, $[\alpha]\mathit{upred} \triangleq [\mathit{bool}, \alpha]\mathit{uexpr}$. We denote the set of alphabetised relations by $[\alpha, \beta]\mathit{urel} \triangleq [\alpha \times \beta]\mathit{upred}$, a predicate over a product space, where α and β denote the sets of undashed and dashed variables[3], called the input and output alphabets. The set of homogeneous relations $[\alpha]\mathit{hrel} \triangleq [\alpha, \alpha]\mathit{urel}$ has identical input and output alphabets. We often notationally distinguish predicates over a unitary type and relations over a product type by use of boldface characters; for example, true is a predicate and $\boldsymbol{\mathit{true}}$ is a relation.

For any given α and β, $[\alpha, \beta]\mathit{urel}$ is partially ordered by refinement $\sqsubseteq$ (refined-by), denoting universally closed reverse implication, where $\boldsymbol{\mathit{false}}$ refines every relation. In this context, $S \sqsubseteq P$ means that P is more deterministic that S. For example, we have it that $(x' > 2) \sqsubseteq (x := 3)$, since the specification that x should finally have a value greater than 2 is satisfied by assigning 3 to x.

Every operator of a sequential programming language can be denoted using relations in UTP. Relational composition ($P\ ;\ Q$) denotes sequential composition, and has the type $[\alpha, \beta]\mathit{urel} \to [\beta, \gamma]\mathit{urel} \to [\alpha, \gamma]\mathit{urel}$, since the output alphabet of the first relation must match the input alphabet of the second. Sequential composition has identity $\mathit{II} \triangleq (v' = v)$, of type $[\alpha]\mathit{hrel}$, where v denotes the entire state.

We summarise the algebraic properties of a homogeneous UTP theory of relations in terms of Boolean quantales [34], a useful algebraic structure for characterising homogeneous relations.

Definition 2.1 (Boolean Quantales)**.** A Boolean quantale [34] is a structure $(S, \leq, 0, \cdot, 1)$, where $(S, \leq)$ is a complete Boolean lattice with least element 0; $(S, \cdot, 1)$ is a monoid with 0 as left and right annihilator; and the function $\cdot$ distributes over the lattice join from the left and right.

Theorem 2.2. *For any α, $([\alpha]\mathit{hrel}, \sqsupseteq, \boldsymbol{\mathit{false}}, ;, \mathit{II})$ is a Boolean quantale [34], so that:*

1. *$([\alpha]\mathit{hrel}, \sqsubseteq)$ is a complete lattice, with infimum $\bigvee$, supremum $\bigwedge$, greatest element $\boldsymbol{\mathit{false}}$, least element $\boldsymbol{\mathit{true}}$, and weakest (least) fixed-point operator $\mu\, F$;*
2. *$([\alpha]\mathit{hrel}, \vee, \boldsymbol{\mathit{false}}, \wedge, \boldsymbol{\mathit{true}}, \neg)$ is a Boolean algebra;*
3. *$([\alpha]\mathit{hrel}, ;, \mathit{II})$ is a monoid with $\boldsymbol{\mathit{false}}$ as left and right annihilator;*
4. *$;$ distributes over $\bigvee$ from the left and right.*

We emphasise that our complete lattice is inverted compared to several conventions [31, 34], which is normal for UTP [28, 8]. In particular, we often use $\bigsqcap_{i \in I} P(i)$ to denote an indexed disjunction over I, which intuitively refers to a nondeterministic choice, and likewise $P \sqcap Q$ to denotes $P \vee Q$. As we have mentioned, refinement reduces nondeterminism, which is illustrated by the following law.

$$\bigsqcap_{i \in I} P(i) \sqsubseteq \bigsqcap_{j \in J} P(j) \text{ when } J \subseteq I$$

In other words, refinement reduces the possible choices that a program is permitted to make. We note that the partial order $\leq$ of the Boolean quantale is $\sqsupseteq$, and so our lattice operators are inverted: for example, $\bigvee$ is the infimum with respect to $\sqsubseteq$, and $\mu\, F$ is the least fixed-point.

[3] Textbook presentations of UTP [28, 8] typically use $in\alpha P$ and $out\alpha P$ to denote the input and output alphabet. Here, we find it more convenient to invoke parametric sets, which is also consistent with our mechanisation.

Relations can be used to denote sequential programming constructs like assignment and iteration [28, 1]. From these denotations the algebraic laws of programming can be derived [27], along with operational and axiomatic presentations of the semantics [28]. Moreover, relations can be enriched to characterise more advanced computational paradigms — such as object orientation [41], real-time [42], hybrid computation [15], and concurrency [28] — using UTP theories that encode semantic domains.

UTP theories use distinguished observational variables to record observable quantities of the program or operating environment. By their very nature, such variables are not under the control of the programmer, and instead are governed by logical invariants called healthiness conditions. For example, we may introduce variables $time, time' : \mathbb{R}_{\geq 0}$ into α to record the time before and after a real-time program fragment executed. We can then define a delay construct, $wait(n) \triangleq time' = time + n \wedge v' = v$, where v is shorthand for any variable other than $time$, that advances time whilst leaving all other variables unchanged.

Normally time can only advance, and so a desirable healthiness condition is $time \leq time'$, a predicate that any relation modelling a healthy real-time program should respect. The delay construct $wait(n)$ is an example of a healthy relation, and $time = 1 \wedge time' = 0$ is an unhealthy one. We can also prove more general theorems for the other relational operators: for example, if P and Q are both healthy, then also clearly $P \,;\, Q$ is healthy, by transitivity of $\leq$. Similar closure laws can be proved for other operators, which allows us to characterise the signature, or syntax, of our UTP theory: the set of function symbols guaranteed to construct healthy programs when the arguments are healthy.

UTP thus inverts the typical denotational semantic approach of defining an inductive syntax tree, for example using an algebraic datatype, and then giving it a semantics by a recursive function. It has the significant advantages that we can (1) further constrain our semantic domain by adding extra healthiness conditions, and (2) extend the signature with additional syntax when necessary, whilst at the same time retaining all theorems proved with respect to the existing healthiness conditions and operators. Moreover, we avoid the need to perform induction over the syntax tree in our proofs.

A UTP theory can be formally characterised as the set of fixed-points of a function $\mathbf{H} : [\alpha]\mathsf{hrel} \to [\alpha]\mathsf{hrel}$, that models the healthiness conditions. For example, $\mathbf{HT}(P) \triangleq (P \wedge time \leq time')$ is an idempotent healthiness function whose fixed-points are those relations that satisfy $time \leq time'$. Any predicate on the observational variables can be encoded as a healthiness function in this way, and therefore we treat the terms healthiness condition and healthiness function as synonyms. If P is a fixed-point of $\mathbf{H}$, it is said to be $\mathbf{H}$-healthy, and the set of healthy relations is $[\![\mathbf{H}]\!]_{\mathsf{H}} \triangleq \{P \mid \mathbf{H}(P) = P\}$.

In UTP, it is desirable that $\mathbf{H}$ is idempotent ($\mathbf{H} \circ \mathbf{H} = \mathbf{H}$) and also monotonic ($X \sqsubseteq Y \Rightarrow \mathbf{H}(X) \sqsubseteq \mathbf{H}(Y)$). Idempotence ensures that, for any P, $\mathbf{H}(P)$ is indeed $\mathbf{H}$-healthy, and also means that $[\![\mathbf{H}]\!]_{\mathsf{H}}$ is actually the image of $\mathbf{H}$. Monotonicity additionally ensures, by the Knaster-Tarski theorem, that $[\![\mathbf{H}]\!]_{\mathsf{H}}$ forms a complete lattice under $\sqsubseteq$. Consequently, there exist strongest and weakest fixed-points operators, which allow us to reason about both nondeterministic and recursive elements of the UTP theory.

Often, we construct a UTP theory by composition of several healthiness functions, $\mathbf{H}_1 \circ \mathbf{H}_2 \cdots \circ \mathbf{H}_n$. In this case, we can demonstrate idempotence and monotonicity of $\mathbf{H}$ using the following important theorem:

Theorem 2.3. *We assume that $\mathbf{H} \triangleq \mathbf{H}_1 \circ \mathbf{H}_2 \cdots \circ \mathbf{H}_n$. Then, $\mathbf{H}$ is idempotent provided that (1) each $\mathbf{H}_i$, for $i \in 1..n$ is idempotent, and (2) each pair commutes: for any $i, j \in 1..n$, such that $i \neq j$, $\mathbf{H}_i \circ \mathbf{H}_j = \mathbf{H}_j \circ \mathbf{H}_i$. Moreover, $\mathbf{H}$ is monotonic, provided each $\mathbf{H}_i$ is monotonic.*

Consequently, we can reason about a composite healthiness condition in terms of its components. In this paper, we use such a UTP theory to characterise concurrent and reactive programs.

2.2. Isabelle/UTP

Theory engineering and verification using UTP is supported by Isabelle/UTP [13, 20], which provides a shallow embedding of the relational calculus on top of Isabelle/HOL, and various approaches to automated proof. The foundation of Isabelle/UTP is its model of observations, which utilises lenses [10, 13, 20] to model variables as algebraic structures. A lens is a pair of functions $\mathit{get} : \mathcal{S} \to \mathcal{V}$ and $\mathit{put} : \mathcal{S} \to \mathcal{V} \to \mathcal{S}$, which are used to query and update a view ($\mathcal{V}$) of a larger observation space ($\mathcal{S}$). We write $X : \mathcal{V} \Longrightarrow \mathcal{S}$ for a lens X viewing $\mathcal{V}$ in the source $\mathcal{S}$, and get_X and put_X for its functions. Typically, $\mathcal{S}$ is characterised

by an alphabet of variables (α), and consequently we can safely conflate the state space and alphabet. We characterise the behaviour of each lens using three axioms [10], which link together the functions.

Definition 2.4 (Lens Axioms)**.** A lens $X : \mathcal{V} \Longrightarrow \mathcal{S}$ satisfies, for any $s : \mathcal{S}$ and $v, v' : \mathcal{V}$, the equations

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

In this paper, we require that all lenses satisfy these three axioms. We note in passing that these axioms have close analogues in Back and von Wright's variable calculus [2], which predate lenses by several years. There, *get* is called val and *put* is called set, but they are governed by the same axioms. These axioms have several models including record types, total functions, and products [13, 20]. From them, we can characterise the laws of assignment and substitution without dependence on a particular state model. Moreover, we describe semantically when two lenses correspond to different variables, using lens independence [13].

Definition 2.5. We require that $X : \mathcal{V}_1 \Longrightarrow \mathcal{S}$ and $Y : \mathcal{V}_2 \Longrightarrow \mathcal{S}$, and then define:

$$X \bowtie Y \triangleq (\forall\, s : \mathcal{S}, u : \mathcal{V}_1, v : \mathcal{V}_2 \bullet \mathit{put}_X(\mathit{put}_Y\, s\, v)\, u = \mathit{put}_Y(\mathit{put}_X\, s\, u)\, v)$$

X and Y are independent provided that their *put* functions commute, meaning that they do not interfere with one another. Lenses can model, not just individual variables, but also sets thereof. Intuitively, a lens $X : \mathcal{V} \Longrightarrow \mathcal{S}$ abstractly characterises a $\mathcal{V}$-shaped subregion of a $\mathcal{S}$. The lens summation operator [13], $X \oplus Y$, allows us to compose two such regions, provided they are independent ($X \bowtie Y$). With it, we can model a set of variables $\{x, y, z\}$ through the summation, $x \oplus y \oplus z$. We also introduce two special lenses [13]:

- $\mathbf{0} : \{\emptyset\} \Longrightarrow \mathcal{S}$, which for any given $\mathcal{S}$, characterises an empty (point) region; and
- $\mathbf{1} : \mathcal{S} \Longrightarrow \mathcal{S}$, which characterises the entirety of $\mathcal{S}$.

We can also use lenses to construct a state space by combining the view of one state $s_2 : \mathcal{S}$ with the complement from another state $s_1 : \mathcal{S}$. This is useful for merging of parallel threads that act on disjoint parts of the state. We define a novel lens override operator to perform this state merge.

Definition 2.6. We fix $X : \mathcal{S} \Longrightarrow \mathcal{V}$ and $s_1, s_2 : \mathcal{S}$, and define $s_1 \lhd_X s_2 \triangleq \mathit{put}_X\, s_1\, (\mathit{get}_X\, s_2)$

Lens override ($s_1 \lhd_X s_2$) extracts the region described by X from s_2 and overwrites the corresponding region in s_1, leaving the complement unchanged. This operator obeys a number of useful algebraic laws.

Theorem 2.7 (Override Laws)**.**

$$s_1 \lhd_{\mathbf{0}} s_2 = s_1 \tag{1}$$
$$s_1 \lhd_{\mathbf{1}} s_2 = s_2 \tag{2}$$
$$s \lhd_X s = s \tag{3}$$
$$(s_1 \lhd_X s_2) \lhd_Y s_3 = (s_1 \lhd_Y s_3) \lhd_X s_3 \qquad \textit{provided } X \bowtie Y \tag{4}$$

Law (1) shows that overriding s_1 with s_2 using $\mathbf{0}$, the empty lens, effectively means that we use none of s_2, and (2) is the dual case with the $\mathbf{1}$ lens. Law (3) shows that overriding a source element is idempotent. Law (4) is a kind of commutativity law. In the term $s_1 \lhd_X s_2 \lhd_Y s_3$ we are constructing a composite source from the X region of s_2, the Y region of s_3, and the remainder from s_1, with the assumption that X and Y are independent. The law shows that we can, in this case, commute the order in which we apply s_2 and s_3.

We can also relate lenses using the sublens preorder [13], $X \preceq Y$, which requires that the view of x is contained within the view of y. For example, $X \preceq X \oplus Y$ – the order is analogous to a subset relation for variable sets: $\{x, y\} \preceq \{x, y, z\}$. Moreover, $\mathbf{0} \preceq X$ and $X \preceq \mathbf{1}$, as these are the smallest and largest lenses.

With lenses, we can also construct substitutions, which are modelled as functions $\sigma : \mathcal{S} \to \mathcal{S}$. They are used in Isabelle/UTP to unify variable substitutions, state updates, assignments, and evaluation contexts, also following the pattern given by Back and von Wright [2]. We can construct substitutions $(\!|x_1 \mapsto v_1, x_2 \mapsto$

$v_2, \cdots, x_n \mapsto v_n\rangle\!\rangle$, which assign an expression $v_i : [\mathcal{V}_i, \mathcal{S}]\mathit{uexpr}$ to each lens $x_i : \mathcal{V}_i \Longrightarrow \mathcal{S}$ with a matching view type. Each expression can refer to the previous values of the variables, and variables not mentioned retain their current value. A substitution $\sigma : \mathcal{S} \to \mathcal{S}$ can be applied to an expression $e : [\mathcal{V}, \mathcal{S}]\mathit{uexpr}$ using the operator $\sigma \dagger e \triangleq e \circ \sigma$, which precomposes the characteristic function of e with the substitution function. We can then define $e[v/x] \triangleq \langle\!\langle x \mapsto v \rangle\!\rangle \dagger e$ to obtain the classical substitution operator. It obeys similar laws to syntactic substitution, though it is a semantic operator [13].

This substitution constructor is syntactic sugar for a more general update operator $\sigma(x \mapsto v)$, which updates the value of lens x to expression v. Specifically, $\langle\!\langle x_1 \mapsto v_1, x_2 \mapsto v_2, \cdots \rangle\!\rangle = id(x_1 \mapsto v_1)(x_2 \mapsto v_2)\cdots$, where id is the identity substitution. Substitution update obeys several useful laws.

Theorem 2.8 (Substitution Laws)**.**

$$\sigma(x \mapsto x) = \sigma \tag{1}$$

$$\sigma(x \mapsto e, y \mapsto f) = \sigma(y \mapsto f, x \mapsto e) \quad \text{if } x \bowtie y \tag{2}$$

$$\sigma(x \mapsto e, y \mapsto f) = \sigma(y \mapsto f) \quad \text{if } x \preceq y \tag{3}$$

$$\sigma \dagger (f\ e_1 \cdots e_n) = f\ (\sigma \dagger e_1) \cdots (\sigma \dagger e_n) \tag{4}$$

$$\sigma(x \mapsto e) \dagger x = e \tag{5}$$

An update of a variable to itself has no effect (1). We can commute two updates provided the variables are independent (2). An update to y overrides one to x when x is a narrower lens than y, or is equivalent (3). Substitution application distributes through applied function symbols (4), and replaces variables with their assigned value (5). These laws provide the foundation for modelling state in a variety of works. In this paper, lenses are valuable in characterising concurrent state updates, as demonstrated in §7.

2.3. Reactive Programs

Whilst sequential programs determine the relationship between an initial and final state, reactive programs also pause during execution to interact with the environment. For example, the CSP [26, 8] and *Circus* [46, 35] languages can model networks of concurrent processes that communicate using shared channels. Reactive behaviour is described using primitives such as event prefix $a \rightarrow P$, which awaits event a and then enables P; conditional guard, $b \mathbin{\&} P$, which enables P when b is true; external choice $P \Box Q$, where the environment resolves the choice by communicating an initial event of P or Q; and iteration ***while*** b ***do*** P. Channels can carry data, and so events can take the form of an input $(c?x)$ or output $(c!v)$. *Circus* processes also have encapsulated state variables that can be assigned $(x := v)$.

We exemplify the *Circus* notation with the program for an unbounded buffer.

Example 2.9. In the *Buffer* process below, variable $bf : \mathrm{seq}\,\mathbb{N}$ is a sequence of natural numbers[4] that records the elements, and channels $inp(n : \mathbb{N})$ and $outp(n : \mathbb{N})$ represent inputs and outputs.

$$\mathit{Buffer} \triangleq bf := \langle\rangle \,;\, \left(\begin{array}{l} \textbf{\textit{while}}\ \mathit{true}\ \textbf{\textit{do}} \\ \quad \mathit{inp}?v \rightarrow bf := bf \frown \langle v \rangle \\ \quad \Box\ (\#bf > 0)\ \&\ \mathit{out}!(\mathit{head}(bf)) \rightarrow bf := \mathit{tail}(bf) \end{array} \right)$$

Here, $xs \frown ys$ denotes sequence concatenation [43], and $\langle x, y, z, \cdots \rangle$ denotes an enumerated sequence. Variable bf is set to the empty sequence $\langle\rangle$, and then a non-terminating loop describes the main behaviour. Its body repeatedly allows the environment to either provide a value v over inp, followed by which bf is extended, or else, if the buffer is non-empty, receive the value at the head, and then bf is contracted. □

Circus has previously been given both a denotational [35] and an operational semantics [47], which are linked in the UTP framework. Here, we build on these previous results and capture the axiomatic semantics for reactive programs using reactive contracts [14]. Reactive contracts can be used both to specify

[4] In Isabelle/UTP, we model sequences using the HOL parametric type $[A]\mathit{list}$, which represents inductive lists.

requirements for reactive programs, under certain assumptions, and also to assign denotational semantics to each operator of a reactive programming language. The denotational semantics symbolically encodes the possible transitions a reactive program can exhibit. We can therefore use a theorem prover to reason about a reactive program with a very large or infinite state space. As an example application, we have used them for verifying state-machine diagrams in the RoboChart language [12].

Reactive contracts are built with the following constructor, which is part of our UTP theory's signature:

$$[\, P_1(tt, st, r) \vdash P_2(tt, st, r, r') \mid P_3(tt, st, st', r, r') \,]$$

P_1 is called the precondition, P_2 is the pericondition, and P_3 is the postcondition. The notation $P_i(x, y, z)$ indicates that relation P_i may refer only to x, y, and z explicitly; any number of variables may be indicated. The variables are modelled as lenses, but for brevity we omit this technicality. Variable tt refers to the trace, which is modelled using a trace algebra [15], and $st, st' : \Sigma$ to the state, for state space Σ. Traces are equipped with operators for the empty trace $\langle\rangle$, concatenation $tt_1 \frown tt_2$, prefix $tt_1 \leq tt_2$, and difference $tt_1 - tt_2$, which removes a prefix tt_2 from tt_1.

P_{1-3} are reactive relations [14]: a specialised form of UTP relation with information about the trace history and state. They respectively encode, (1) the precondition in terms of the initial state and permissible traces; (2) possible intermediate interactions with respect to an initial state; and (3) final states following execution. Pericondition P_2 and postcondition P_3 are both within the "guarantee" part of the underlying design contract, and so can be strengthened by refinement; see [14] for details. P_2 does not refer to intermediate state variables since they are concealed when a program is quiescent. We sometimes abbreviate $[\, \mathbf{true}_r \vdash P_2 \mid P_3 \,]$, a contract with a true precondition, with the notation $[\, \vdash P_2 \mid P_3 \,]$. Our precondition corresponds to the "assume" part of a contract. Reactive contracts lie with the greater field of assume-guarantee conditions [4, 5, 40]; a detailed comparison can be found in [14].

In this paper, traces are modelled as finite sequences, $tt : \mathrm{seq}\ \mathit{Event}$, for some set of events given by Event, though other models are also admitted [15]. Events can be parametric, written $a.x$, where a is a channel and x is the data. Moreover, the relations can encode additional semantic data, such as refusals, using variables r, r'. Our theory, therefore, provides an extensible denotational semantic model for reactive and concurrent languages. To exemplify, we consider the semantics of the skip, event, and assignment actions from *Circus*, which require that we add variable $ref' : \mathbb{P}(\mathit{Event})$ to record refusals.

Definition 2.10 (Skip Action, Terminated Event Prefix, and Assignment)**.**

$$\begin{aligned}
\mathbf{Skip} &\triangleq [\, \mathbf{true}_r \vdash \mathbf{false} \mid tt = \langle\rangle \wedge st' = st \,] \\
\mathbf{Do}(a) &\triangleq \left[\, \mathbf{true}_r \vdash tt = \langle\rangle \wedge a \notin ref' \;\middle|\; tt = \langle a \rangle \wedge st' = st \,\right] \\
x := v &\triangleq [\, \mathbf{true}_r \vdash \mathbf{false} \mid st' = st(x \mapsto v) \wedge tt = \langle\rangle \,]
\end{aligned}$$

Each of these contracts specifies the possible behaviours that can be observed in the reactive program. **Skip** is an action that cannot diverge, and immediately terminates leaving the state unchanged. Its precondition is $\mathbf{true}_r$, the universal reactive relation (defined below), since it is always satisfied. The pericondition is **false** because there are no quiescent behaviours. In the postcondition, we define that no events occur ($tt = \langle\rangle$), and the state is left unchanged ($st' = st$). The event action ($\mathbf{Do}(a)$) also has a true precondition. In the pericondition, we specify that in an intermediate state no events have occurred, but a is not being refused – intuitively this means that the program is waiting to engage in the a event. In the postcondition, we specify that the trace is extended by a, since it has now happened, and the state is unchanged. With this we can define the *Circus* event prefix: $a \rightarrow P \triangleq \mathbf{Do}(a)\ ;\ P$. Assignment also has a true precondition, and a false pericondition since it terminates without interaction. The postcondition specifies the updates to the state, and leaves the trace unchanged.

As mentioned, reactive contracts can also be used as a specification mechanism. For example, we can define the following contract for deadlock-freedom.

Example 2.11 (Deadlock-freedom Contract)**.** $\mathbf{CDF} \triangleq \left[\, \mathbf{true}_r \vdash \exists\, e \bullet e \notin ref' \;\middle|\; \mathbf{true}_r \,\right]$ 

CDF requires that every intermediate observation must exhibit at least one enabled event e, that is, one event e is not being refused – that is what deadlock-freedom means. The pre- and postcondition do not specify any particular behaviours, since we are only concerned with quiescent observations. Any reactive program that refines **CDF** must always have an enabled transition. For example, it is the case that $\mathbf{CDF} \sqsubseteq \mathbf{Do}(a)$. This can be formally demonstrated using the contract refinement theorem below (Theorem 2.14). First though, we give an overview of the encoding of reactive contracts in UTP.

Following the UTP approach, the constructor $[\, P_1 \vdash P_2 \mid P_3 \,]$ is really syntactic sugar for a complex relation [14] that is defined using constructs from the UTP theories of reactive processes and designs[5]. Consequently, contracts can be composed using the UTP relational operators. Reactive relations and contracts are characterised by healthiness conditions **RR** and **NSRD**, respectively, which we have previously described [14], and reproduce in Table 1. They are all both idempotent and continuous [14]. The observational variables include ok and $wait$, which are used to distinguish normal from divergent behaviour, and quiescent from terminating behaviour, respectively. **NSRD** specialises the theory of reactive designs [8, 35] to *normal stateful reactive designs* [14]. This version of reactive designs imposes the requirement that $\mathbf{st}'$ cannot be referenced in the pericondition, as we assume that quiescent observations do not reveal the state.

Reactive relations characterise the inner elements of a reactive contract, namely the pre-, peri-, and postconditions. Using healthiness conditions called **R1** and **R2**, **RR** ensures that every observation describes a well-formed trace ($\mathbf{tt}$), and furthermore does not depend on ok or $wait$, as these are only required by the reactive contract infrastructure. Technically, $\mathbf{tt}$ is not a relational variable, but a special variable $\mathbf{tt} \triangleq tr' - tr$ where tr, tr', as usual in UTP, encode the trace relationally [28], under the assumption that **RR** is satisfied. Nevertheless, due to our previous results [20, 15], $\mathbf{tt}$ can be treated as a variable, and it is more intuitive to do so. We treat tr and tr' as semantic machinery that is concealed in $\mathbf{tt}$, which represents the actual trace.

Preconditions of a reactive contract are elements $[\![\mathbf{RC}]\!]_{\mathsf{H}}$, which specialises **RR** by requiring that only the initial state ($\mathbf{st}$) is referenced, and that the trace is prefix closed. The intuition here is that when a trace violates the precondition of a contract, then any extension of this trace must also violate it, similar to how the set of divergences in CSP is extension closed [6]. By duality, if a trace satisfies the precondition, then any prefix of the trace must also satisfy the precondition, and hence the precondition is prefix closed with respect to the trace. The basic reactive relational operators are defined below.

Definition 2.12 (Reactive Relational Operations)**.**

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

$$\mathbb{I}_r \triangleq (tr' = tr \wedge \mathbf{st}' = \mathbf{st} \wedge \mathbf{v}' = \mathbf{v}) \qquad \mathbb{I}_{\mathbf{R}} \triangleq ((\exists\, st \bullet \mathbb{I}) \lhd wait \rhd \mathbb{I}) \lhd ok \rhd (tr \leq tr')$$

The theory of reactive relations forms a Boolean algebra, but we have to redefine **true**, $\neg$, and $\Rightarrow$ as these are not reactive relations. The relational **true** is not **RR** healthy, since it permits any combination of tr and tr', and so we define $\mathbf{true}_r$ to be the least reactive relation. We also need a bespoke complement, $(\neg_r P)$, because $[\![\mathbf{RR}]\!]_{\mathsf{H}}$ is similarly not closed under $\neg$. So, after taking the negation, we need to apply **R1** to obtain a healthy relation. We also redefine implication for the same reasons $(P \Rightarrow_r Q)$. We do not need to redefine **false** because, unlike **true**, it is already **RR**-healthy, and the same follows for the other logical connectives. We then have proved the following theorem [14].

Theorem 2.13. $([\![\mathbf{RR}]\!]_{\mathsf{H}}, \vee, \mathbf{false}, \wedge, \mathbf{true}_r, \neg_r)$ *forms a Boolean algebra*

Both $[\![\mathbf{RR}]\!]_{\mathsf{H}}$ and $[\![\mathbf{NSRD}]\!]_{\mathsf{H}}$ are closed under sequential composition, and have units $\mathbb{I}_r$ and $\mathbb{I}_{\mathbf{R}}$, respectively. We note that $\mathbb{I}_{\mathbf{R}}$ and **Skip** are different operators, as the latter does not restrict **ref** in the pericondition [14]. Both UTP theories also form complete lattices under $\sqsubseteq$, with top elements **false** and $\mathbf{Miracle} = [\mathbf{true}_r \vdash \mathbf{false} \mid \mathbf{false}]$, respectively. $\mathbf{Chaos} = [\mathbf{false} \vdash \mathbf{false} \mid \mathbf{false}]$, the least determinisitic contract, is the bottom of the reactive contract lattice. Any action refines **Chaos**, and it therefore allows us to denote unspecified or unpredictable behaviour. We define the conditional operator $P \lhd b \rhd Q \triangleq ((b \wedge P) \vee (\neg b \wedge Q))$, where b is a condition on state variables, which can be used for both reactive relations and contracts.

[5] The definition of contracts is here omitted, for reasons of brevity, but it can be found in [14]

Healthiness Condition	Description
$\mathbf{R1}(P) \triangleq P \wedge tr \leq tr'$	The trace monotonically increases
$\mathbf{R2}(P) \triangleq P[\langle\rangle, tr' - tr/tr, tr'] \lhd tr \leq tr' \rhd P$	The trace extension is independent of the history
$\mathbf{R3}_h(P) \triangleq \mathbb{I}_R \lhd wait \rhd P$	When a predecessor is quiescent behave as $\mathbb{I}_R$
$\mathbf{RR}(P) \triangleq (\exists(ok, ok', wait, wait') \bullet \mathbf{R1}(\mathbf{R2}(P)))$	Reactive Relations; no references to ok and $wait$
$\mathbf{RC}(P) \triangleq \mathbf{R1}(\mathbf{RR}(P) \,;\, tr' \leq tr)$	Reactive Conditions: trace is also prefix closed
$\mathbf{SRD1}(P) \triangleq (ok \Rightarrow_r P)$	Observations are only possible without divergence
$\mathbf{SRD3}(P) \triangleq (P \,;\, \mathbb{I}_R)$	Reactive skip is a right unit for ;
$\mathbf{NSRD} \triangleq \mathbf{SRD3} \circ \mathbf{SRD1} \circ \mathbf{R3}_h \circ \mathbf{R2} \circ \mathbf{R1}$	Normal Stateful Reactive Designs

Table 1: Overview of Reactive Design Healthiness Conditions

Verification can be facilitated through refinement $[P_1 \vdash P_2 \mid P_3] \sqsubseteq Q$, where the required property is specified as an explicit contract triple, and the program Q is an **NSRD** relation. Contract refinement allows the precondition to be weakened, and the peri- and postcondition both to be strengthened [14].

Theorem 2.14 (Reactive Design Refinement)**.**
$[\, P_1 \vdash P_2 \mid P_3 \,] \sqsubseteq [\, Q_1 \vdash Q_2 \mid Q_3 \,]$ *if, and only if,* $P_1 \Rightarrow Q_1$, $P_2 \sqsubseteq (Q_2 \wedge P_1)$, *and* $P_3 \sqsubseteq (Q_3 \wedge P_1)$.

Thus, if the contract of the reactive program Q can be calculated to be $[Q_1 \vdash Q_2 \mid Q_3]$, then refinement follows by three proof obligations: (1) $P_1 \Rightarrow Q_1$; (2) $P_2 \sqsubseteq (Q_2 \wedge P_1)$; and (3) $P_3 \sqsubseteq (Q_3 \wedge P_1)$. In words, the precondition must be weakened, and both the peri- and postcondition must be strengthened, assuming the precondition P_1 holds. As usual, refinement can remove choices, making a contract more deterministic. A consequence is that a non-terminating contract, with postcondition **false**, can refine a terminating contract. Indeed we have that for any P, $P \sqsubseteq \mathbf{Miracle}$. We can avoid refinement by miraculous behaviour by adding feasibility healthiness conditions [28, 8], though this is not a concern for this paper.

Contracts can be composed using relational calculus. The following identities [14, 16] show how this entails composition of the underlying pre-, peri-, and postconditions for $\bigsqcap$ and ;, and also demonstrate closure of reactive contracts under these operators.

Theorem 2.15 (Reactive Contract Composition)**.**

$$\textstyle\bigsqcap_{i \in I} [P_1(i) \vdash P_2(i) \mid P_3(i)] = \left[\, \bigwedge_{i \in I} P_1(i) \vdash \bigvee_{i \in I} P_2(i) \mid \bigvee_{i \in I} P_3(i) \,\right] \tag{1}$$

$$[\, P_1 \vdash P_2 \mid P_3 \,] \lhd b \rhd [\, Q_1 \vdash Q_2 \mid Q_3 \,] = [\, P_1 \lhd b \rhd Q_1 \vdash P_2 \lhd b \rhd Q_2 \mid P_3 \lhd b \rhd Q_3 \,] \tag{2}$$

$$[P_1 \vdash P_2 \mid P_3] \,;\, [Q_1 \vdash Q_2 \mid Q_3] = [P_1 \wedge (P_3 \;\mathbf{wlp}_r\; Q_1) \vdash P_2 \vee (P_3 \,;\, Q_2) \mid P_3 \,;\, Q_3 \,] \tag{3}$$

$$[\, \vdash P_2 \mid P_3 \,] \,;\, [\, \vdash Q_2 \mid Q_3 \,] = [\, \vdash P_2 \vee (P_3 \,;\, Q_2) \mid P_3 \,;\, Q_3 \,] \tag{4}$$

Nondeterministic choice requires all preconditions, and asserts that one of the peri- and postcondition pairs hold. Conditional $(P \lhd b \rhd Q)$ distributes through a reactive contract. For sequential composition, the precondition assumes that P_1 holds, and that P_3 does not violate Q_1. The latter is formulated using a reactive weakest liberal precondition ($\mathbf{wlp}_r$). Intuitively, $P \;\mathbf{wlp}_r\; b$ is the weakest reactive condition such that when P terminates, it satisfies b. It obeys standard predicate transformer laws [9, 14] such as:

$$(\textstyle\bigvee_{i \in I} P(i)) \;\mathbf{wlp}_r\; R = \bigwedge_{i \in I} P(i) \;\mathbf{wlp}_r\; R \qquad (P \,;\, Q) \;\mathbf{wlp}_r\; R = P \;\mathbf{wlp}_r\; (Q \;\mathbf{wlp}_r\; R) \qquad P \;\mathbf{wlp}_r\; \mathbf{true}_r = \mathbf{true}_r$$

In the pericondition of Theorem 2.15-(3), it is specified that an intermediate observation if either of the first contract (P_2), or else it terminated (P_3) and then following we have an intermediate observation of

the second contract (Q_2). In the postcondition, the observation specified is for when the contracts have both terminated (P_3 ; Q_3). The final law, Theorem 2.15-(4), is a simpler case of the previous law. If both preconditions are true, then since P_2 $\mathbf{wlp}_r$ $\mathbf{true}_r$ reduces to $\mathbf{true}_r$, the overall precondition is also $\mathbf{true}_r$.

With these and related theorems [14], we can calculate contracts of reactive programs. Verification, then, can be performed by proving refinement between two reactive contracts, a strategy we have mechanised in the Isabelle/UTP tactics rdes-refine and rdes-eq [14]. The question remains, though, of how to reason about the underlying compositions of reactive relations for the pre-, peri-, and postconditions. As an example, we consider the action $(a \rightarrow \mathbf{Skip})$; $x := v$. To reason about its postcondition, we must simplify $(tt = \langle a \rangle \wedge st' = st)$; $(st' = st(x \mapsto v) \wedge tt = \langle\rangle)$. To simplify its precondition, we also need to consider reactive weakest preconditions. Without such simplifications, reactive relations can grow very quickly and hamper proof. Of particular importance is the handling of iterative and parallel reactive relations. We address these issues in this paper.

3. Linking UTP and Kleene Algebra

In this section, we characterise properties of a UTP theory sufficient to identify a Kleene Algebra [30], and use this to obtain theorems for iterative contracts. The results in this section apply, not only to stateful-failure reactive designs, but the larger class of reactive designs ($\mathbf{NSRD}$) as well. Consequently, the theorems can be applied in the context of other trace models [15].

Kleene Algebras (KA) characterise sequential and iterative behaviour in nondeterministic programs using a signature $(K, +, 0, \cdot, 1, ^*)$, where $+$ is a choice operator with unit 0, and $\cdot$ a composition operator, with unit 1. Kleene closure P^* denotes iteration of P using $\cdot$ zero or more times.

We consider the class of weak Kleene algebras [22], which build on weak dioids, as these are the most useful class of Kleene algebra to characterise reactive programs.

Definition 3.1. A weak dioid is an algebraic structure $(K, +, 0, \cdot, 1)$ such that $(K, +, 0)$ is an idempotent and commutative monoid; $(K, \cdot, 1)$ is a monoid; the composition operator $\cdot$ left- and right-distributes over $+$; and 0 is a left annihilator for $\cdot$.

The 0 operator represents miraculous behaviour. It is a left annihilator of composition, but not a right annihilator as this often does not hold for programs. K is partially ordered by $x \leq y \triangleq (x + y = y)$, which is defined in terms of $+$, and has least element 0. A weak KA extends this with the behaviour of the star.

Definition 3.2. A weak Kleene algebra is a structure $(K, +, 0, \cdot, 1, ^*)$ such that

1. $(K, +, 0, \cdot, 1)$ is a weak dioid
2. $1 + x \cdot x^* \leq x^*$
3. $z + x \cdot y \leq y \Rightarrow x^* \cdot z \leq y$
4. $z + y \cdot x \leq y \Rightarrow z \cdot x^* \leq y$

Various enrichments and specialisations of these axioms exist; for a complete survey see [30]. For our purposes, these axioms alone suffice. From this base, a number of useful identities can be derived:

Theorem 3.3. $1^* = 0^* = 1 \quad x^{**} = x^* \quad x^* = 1 + x \cdot x^* \quad (x + y)^* = (x^* \cdot y^*)^* \quad x \cdot x^* = x^* \cdot x$

Kleene Algebra with Tests [31] (KAT) extends the algebra with conditions, and has been successfully applied in program verification [1, 21]. A test is a kind of assumption that entails miraculous behaviour if a condition is violated, and is otherwise ineffectual. The set of tests T are those elements $a, b \in K$ below the identity: $a \leq 1$, over which a Boolean algebra is defined. Tests enjoy a number of additional properties.

Theorem 3.4. $a \cdot 0 = 0 \quad a \cdot b = b \cdot a \quad a^* = 1$

UTP relations form a KA $(Rel, \sqcap, ;, \mathbb{I}, ^*)$, where $P^* \triangleq (\nu X \bullet \mathbb{I} \sqcap P \ ; X)$. This definition is equivalent to $P^* = (\sqcap_{i \in \mathbb{N}} P^i)$ [19] where P^n iterates sequential composition n times. The proof proceeds by application of antisymmetry, the star induction law of Definition 3.2, and the complete lattice theorems.

Typically, UTP theories, like $[\![\mathbf{NSRD}]\!]_H$, share the operators for choice ($\sqcap$) and composition ($;$), only redefining them when absolutely necessary. Formally, given a UTP theory defined by a healthiness condition

$\mathbf{H}$, the set of healthy relations $[\![\mathbf{H}]\!]_{\mathsf{H}}$ is closed under $\sqcap$ and ;. This has the major advantage that a large body of laws is directly applicable from the relational calculus. The ubiquity of $\sqcap$, in particular, can be characterised through the subset of continuous UTP theories, where $\mathbf{H}$ distributes through arbitrary non-empty infima. We formally define this class of healthiness condition below.

Definition 3.5 (Continuous Healthiness Condition)**.**

$$\mathbf{H}\left(\sqcap_{i\in I}\ P(i)\right) = \sqcap_{i\in I}\ \mathbf{H}(P(i)) \text{ provided } I \neq \emptyset$$

An infinite nondeterministic choice is necessary to support Kleene star iteration. Monotonicity of $\mathbf{H}$ follows from continuity, and so such theories induce a complete lattice. Moreover, if $\mathbf{H}$ is defined by composition $\mathbf{H}_1 \circ \mathbf{H}_2 \cdots \circ \mathbf{H}_n$, as in Theorem 2.3, then we can show it is continuous by showing each $\mathbf{H}_i$ is continuous. Continuous UTP theories include designs [28, 22], CSP, and *Circus* [35]. A consequence of continuity is that the relational weakest fixed-point operator $\mu\, X \bullet F(X)$ constructs healthy relations when $F : \mathit{Rel} \to [\![\mathbf{H}]\!]_{\mathsf{H}}$.

Though these theories share infima and weakest fixed points, they do not, in general, share $\top$ and $\bot$ elements, which is why the infima are non-empty Definition 3.5. Rather, we have a top element $\top_{\mathbf{H}} \triangleq \mathbf{H}(\mathbf{false})$ and a bottom element $\bot_{\mathbf{H}} \triangleq \mathbf{H}(\mathbf{true})$ [14]. The theories also do not share the relational identity $\mathbb{I}$, but typically define a bespoke identity $\mathbb{I}_{\mathbf{H}}$, which means that $[\![\mathbf{H}]\!]_{\mathsf{H}}$ is not closed under the relational Kleene star. However, $[\![\mathbf{H}]\!]_{\mathsf{H}}$ is closed under Kleene plus, $P^+ \triangleq P\ ;\ P^*$, since it is equivalent to $(\sqcap_{i\in\mathbb{N}}\ P^{i+1})$, which iterates P one or more times. Thus, we can obtain a theory Kleene star with $P^* \triangleq \mathbb{I}_{\mathbf{H}} \sqcap P^+$, under which $\mathbf{H}$ is indeed closed. We, therefore, define the following criteria for a UTP theory.

Definition 3.6. A Kleene UTP theory $(\mathbf{H}, \mathbb{I}_{\mathbf{H}})$ satisfies the following conditions: (1) $\mathbf{H}$ is idempotent and continuous; (2) $\mathbf{H}$ is closed under sequential composition; (3) identity $\mathbb{I}_{\mathbf{H}}$ is $\mathbf{H}$-healthy; (4) it is a left- and right-unit, $\mathbb{I}_{\mathbf{H}}\ ;\ P = P\ ;\ \mathbb{I}_{\mathbf{H}} = P$, when P is $\mathbf{H}$-healthy; and (5) $\top_{\mathbf{H}}\ ;\ P = \top_{\mathbf{H}}$, when P is $\mathbf{H}$-healthy.

From these properties, we can prove the following theorem.

Theorem 3.7. *If* $(\mathbf{H}, \mathbb{I}_{\mathbf{H}})$ *is a Kleene UTP theory, then* $([\![\mathbf{H}]\!]_{\mathsf{H}}, \sqcap, \top_{\mathbf{H}}, ;, \mathbb{I}_{\mathbf{H}}, ^*)$ *forms a weak Kleene algebra.*

Proof. We prove this in Isabelle/UTP by lifting of laws from the Isabelle/HOL KA hierarchy [1, 21]. For details see [11]. □

The identities of Theorem 3.3 hold in a Kleene UTP theory, which allow us to reason about iterative programs. In particular, we can show that $([\![\mathbf{NSRD}]\!]_{\mathsf{H}}, \sqcap, \mathbf{Miracle}, ;, \mathbb{I}_{\mathbf{R}}, ^*)$ and $([\![\mathbf{RR}]\!]_{\mathsf{H}}, \sqcap, \mathbf{false}, ;, \mathbb{I}_r, ^*)$ both form weak KAs. Moreover, we can now also show how to calculate iterative contracts [16].

Theorem 3.8 (Reactive Contract Iteration)**.**

$$[\,P_1 \vdash P_2 \mid P_3\,]^* = [\,P_3^*\ \mathbf{wlp}_r\ P_1 \vdash P_3^*\ ;\ P_2 \mid P_3^*\,]$$
$$[\,P_1 \vdash P_2 \mid P_3\,]^+ = \left[\,P_3^*\ \mathbf{wlp}_r\ P_1 \vdash P_3^*\ ;\ P_2 \;\middle|\; P_3^+\,\right]$$

We note that the outer and inner star are different operators. The outer star is formed from the identity $\mathbb{I}_{\mathbf{R}}$, and the inner star from $\mathbb{I}_r$. The precondition states that P_3 must not violate P_1 after any number of iterations. The pericondition has P_3 iterated followed by P_2 holding, since the final observation is intermediate. The postcondition simply iterates P_3. We also provide a similar law for the Kleene plus operator. It distributes in the same way, except that both the precondition and the pericondition use the star because they must hold before the first iteration; only the postcondition uses the plus.

In this section we have established the basis for calculating and reasoning about iterative reactive contracts. In the next section we specialise our UTP theory to stateful-failure reactive designs, and developed the underlying equational theory. We return to the subject of iteration in Section 6.

4. Reactive Relations of Stateful Failures-Divergences

In this section, we specialise our contract theory to incorporate failure traces, which are used in CSP, *Circus*, and related languages [48]. We define atomic operators to describe the underlying reactive relations, and the associated equational theory to expand and simplify compositions arising from Theorems 2.15 and 3.8, and thus support automated reasoning. We consider external choice separately (§5).

The failures-divergences model [38] was defined to give a denotational semantics to CSP. It models a process with a pair of sets: $F \subseteq \mathbb{P}(\text{seq } \mathit{Event} \times \mathbb{P}\ \mathit{Event})$ and $D \subseteq \mathbb{P}(\text{seq } \mathit{Event})$, which are, respectively, the set of failures and divergences. A failure is a trace of events plus a set of events can be refused at the end of the interaction. A divergence is a trace of events that leads to divergent behaviour, that is, unpredictable behaviour like that of $\mathbf{Chaos}$. A distinguished event $\checkmark \in \mathit{Event}$ is used as the final element of a trace to indicate that this is a terminating observation. The UTP gives a relational account of the failures-divergences model [28], which was expanded upon by Woodcock and Cavalcanti [8], and by Oliveira [36] to account for state variables in *Circus* [35]. It is this latter model that we here call the stateful failures-divergences model.

Healthiness condition $\mathbf{NCSP} \triangleq \mathbf{NSRD} \circ \mathbf{CSP3} \circ \mathbf{CSP4}$ characterises the stateful failures-divergences model [8, 35]. Healthiness conditions $\mathbf{CSP3}$ and $\mathbf{CSP4}$ are defined below.

Definition 4.1 (Stateful-Failure Healthiness Conditions)**.**

$\mathbf{CSP3}(P) \triangleq (\mathbf{Skip} \,;\, P)$	There are no references to ref.
$\mathbf{CSP4}(P) \triangleq (P \,;\, \mathbf{Skip})$	The postcondition may not refer to ref'.

$\mathbf{CSP3}$ and $\mathbf{CSP4}$ ensure the refusal sets are well-formed [8, 28]: ref' can be mentioned only in the pericondition, since refusals are only observed in quiescent observations. $\mathbf{NCSP}$, like $\mathbf{NSRD}$, is continuous and has $\mathbf{Skip}$, defined below, as a left and right unit. Thus, $([\![\mathbf{NCSP}]\!]_{\mathrm{H}}, \sqcap, \mathbf{Miracle}, ;, \mathbf{Skip}, ^*)$ forms a Kleene algebra. An $\mathbf{NCSP}$ contract has the following specialised form [17].

$$\left[\, P_1(tt, st) \vdash P_2(tt, st, ref') \;\middle|\; P_3(tt, st, st') \,\right]$$

The underlying reactive relations capture a portion of the stateful failures-divergences. P_1 is the precondition, which captures the initial states and traces that do not induce divergence. It corresponds to the the complement of D, the set of divergences [38, 8]. P_2 is the pericondition, which captures the stateful failures of a program: the set of events not being refused (ref') having performed trace tt, starting in state st. It corresponds to the the failure traces in F that are not terminating. P_3 captures the terminated behaviours, where a final state is observed but no refusals. It, of course, corresponds to the traces in F that have $\checkmark$ as the final element. We now characterise these reactive relations using healthiness conditions.

Definition 4.2. Stateful-failure Reactive Relations, Finalisers, and Conditions are characterised as fixed-points of the healthiness conditions $\mathbf{CRR}$, $\mathbf{CRF}$, and $\mathbf{CRC}$ defined below.

$$\mathbf{CRR}(P) \triangleq \exists\, \mathit{ref} \bullet \mathbf{RR}(P) \qquad \mathbf{CRF}(P) \triangleq \exists(\mathit{ref}, \mathit{ref}') \bullet \mathbf{RR}(P) \qquad \mathbf{CRC}(P) \triangleq \exists\, \mathit{ref} \bullet \mathbf{RC}(P)$$

These are straightforward extensions of the healthiness conditions for reactive relations ($\mathbf{RR}$) and conditions ($\mathbf{RC}$) that we previously defined [14] and are presented in Table 1. In addition to requiring that the relations describe a well-formed trace, $\mathbf{CRR}$, $\mathbf{CRF}$, and $\mathbf{CRC}$ require that there is no reference to ref, because there is never a dependence on the refusal set of a predecessor. Reactive finalisers ($\mathbf{CRF}$) additionally forbid reference to ref': they are used to characterise postconditions in a stateful-failure reactive contracts. Every $\mathbf{CRF}$-healthy relation is also $\mathbf{CRR}$-healthy; further containments are shown in Figure 1. We can formally characterise $\mathbf{NCSP}$ contracts with the following theorem.

Theorem 4.3. $[\, P_1 \vdash P_2 \mid P_3 \,]$ *is* $\mathbf{NCSP}$*-healthy if the following conditions are satisfied:*

1. P_1 *is* $\mathbf{CRC}$*-healthy;*
2. P_2 *is* $\mathbf{CRR}$*-healthy;*

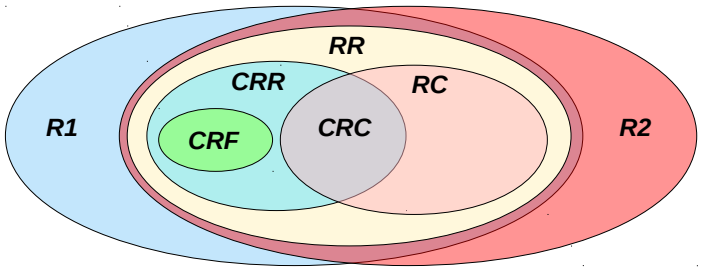

Figure 1: Reactive Relational Healthiness Conditions Venn diagram

3. P_2 *does not refer to* st'*; and*
4. P_3 *is* **CRF***-healthy.*

Due to the restrictions on *ref*, $\mathbb{I}_r$ is not **CRR**-healthy, and so we define the identity relation below.

Definition 4.4. $\mathbb{I}_c \triangleq (st' = st \land tr' = tr)$

This identity specifies only that the trace and state remain unchanged, whilst *ref* is unspecified, and so it is **CRF**-healthy. $\mathbb{I}_c$ is a left unit for **CRR**, **CRF**, and **CRC**-healthy relations, but it is a right unit only for **CRF**. This is because in $P \,;\, \mathbb{I}_c$, if P refers to ref' then this information is lost, whilst **CRF** relations do not refer to ref'. We can construct a Kleene algebra for **CRF**-healthy relations.

Theorem 4.5. $([\![\textbf{CRF}]\!]_H, \sqcap, \textbf{false}, ;, \mathbb{I}_c, ^*)$ *forms a weak Kleene algebra.*

Using the Kleene star operators for **NCSP** and **CRF**, we can also revalidate Theorem 3.8 in this context. This is necessary because the star in Theorem 3.8 is defined in terms of $\mathbb{I}_r$ and not $\mathbb{I}_c$. Nevertheless the two star operators are strongly related, and so we need only to slightly adapt the proof.

Having defined our theories of stateful-failure reactive relations, we now proceed to define operators for constructing pre-, peri-, and postconditions. These operators allow us to describe a distinct pattern for the form of reactive programs. We describe this pattern using the following constructs.

Definition 4.6 (Reactive Relational Operators)**.**

$$\begin{array}{lr} \text{Assumption:} & \mathcal{I}[s(st), t(st)] \triangleq \textbf{CRC}(s(st) \Rightarrow_r \neg_r (t(st) \leq tt)) \\ \text{Quiescence:} & \mathcal{E}[s(st), t(st), E(st)] \triangleq \textbf{CRR}(s(st) \land tt = t(st) \land (\forall\, e \in E(st) \bullet e \notin ref')) \\ \text{Finalisation:} & \Phi[s(st), \sigma, t(st)] \triangleq \textbf{CRR}(s(st) \land st' = \sigma(st) \land tt = t(st)) \end{array}$$

We utilise expressions s, t, and E that refer only to the variables indicated. Namely, $s : \mathbb{B}$ is a condition on st, $t : \mathrm{seq}\, Event$ is a trace expression that describes an event sequence in terms of st, and $E : \mathbb{P}\, Event$ describes a set of events. The use of trace expressions allows us to handle symbolic traces that contain free state variables, and characterise a potentially infinite number of traces with a finite presentation. With this, we can calculate semantics for reactive programs with an infinite number of states.

$\mathcal{I}[s(st), t(st)]$ is a **CRC**-healthy reactive condition that is used to specify assumptions on the state and trace in preconditions. It states that, if the state initially satisfies condition s, then t is not a prefix of the overall trace. For example, the assumption $\mathcal{I}[x > 2, \langle a, b \rangle]$ means that if the state variable x is initially greater than 2, then we disallow the trace $\langle a, b \rangle$, and any extension thereof. The intuition here is that t is a trace that introduces divergence, and so any extension of t violates the precondition. Put another way, any trace that is a either strict prefix or orthogonal to t satisfies the precondition when s holds. Effectively, t sets a strict upper bound on the traces permitted by the precondition.

$\mathcal{E}[s(st), t(st), E(st)]$ is used in periconditions to specify quiescent observations, and corresponds to a set of symbolic failure traces. It specifies that the state variables initially satisfy s, the interaction described by t has occurred, and finally we reach a quiescent phase where none of the events in E are being refused.

It has the form of an acceptance trace [38], as this provides a more comprehensible presentation, but the semantics is encoded as a collection of failure traces. Specifically, $(\forall\, e \in E(\mathit{st}) \bullet e \notin \mathit{ref}')$ defines all possible refusals that satisfy symbolic acceptance set E. We sometimes write $\mathcal{E}[t(\mathit{st}), E(\mathit{st})]$ when s is true.

$\Phi[s(\mathit{st}), \sigma, t(\mathit{st})]$ is used in postconditions to specify final terminated observations. It specifies that the initial state satisfies s, the state update σ is applied to update st, and the symbolic interaction t has occurred. Since $\Phi[s, \sigma, t]$ does not refer to ref', it is $\mathbf{CRF}$-healthy. We sometimes write $\Phi[\sigma, t(\mathit{st})]$ in the case that s is true. Moreover, we also introduce the abbreviation $[s]_c \triangleq \Phi[s, id, \langle\rangle]$ that denotes a reactive relational test (or assumption) on the state of the property s.

These operators are all deterministic, in the sense that they describe a single interaction and state-update history. There is no need for explicit nondeterminism here, as this is achieved using $\bigvee$. These operators allow us to concisely specify the basic operators of our theory as given below.

Definition 4.7 (Basic Reactive Operators)**.**

$$\langle\sigma\rangle_c \triangleq [\, \mathbf{true}_r \vdash \mathbf{false} \mid \Phi[\mathit{true}, \sigma, \langle\rangle] \,] \tag{1}$$

$$\mathbf{Skip} = \langle id\rangle_c \tag{2}$$

$$\mathbf{Do}(a) = [\, \mathbf{true}_r \vdash \mathcal{E}[\mathit{true}, \langle\rangle, \{a\}] \mid \Phi[\mathit{true}, id, \langle a\rangle] \,] \tag{3}$$

$$\mathbf{Stop} \triangleq [\, \mathbf{true}_r \vdash \mathcal{E}[\mathit{true}, \langle\rangle, \emptyset] \mid \mathbf{false} \,] \tag{4}$$

The definitions of $\mathbf{Skip}$ and $\mathbf{Do}(a)$ are expressed as theorems that we have proved using Definition 2.10. However, for the remainder of this paper we treat these identities as definitions. Generalised assignment $\langle\sigma\rangle_c$ is again inspired by [2]. It has a $\mathbf{true}_r$ precondition and a $\mathbf{false}$ pericondition: it has no intermediate observations. The postcondition states that for any initial state (true), the state is updated using σ, and no events are produced ($\langle\rangle$). A singleton assignment $x := v$ can be expressed using $\langle x \mapsto v\rangle_c$. We can use it to show that $\mathbf{Skip} = \langle id\rangle_c$, where $id : \Sigma \to \Sigma$ is the identity function that leaves all variables unchanged.

$\mathbf{Do}(a)$ encodes an event action. Its pericondition states that no event has occurred, and a is accepted. Its postcondition extends the trace by a, leaving the state unchanged. We can denote *Circus* event prefix $a \rightarrow P$ as $\mathbf{Do}(a)\ ;\ P$. Finally, $\mathbf{Stop}$ represents a deadlock: its pericondition states the trace is unchanged and no events are being accepted. The postcondition is false as there is no way to terminate. A *Circus* guard $g \mathbin{\&} P$ can be denoted as $(P \lhd g \rhd \mathbf{Stop})$, which behaves as P when g is true, and otherwise deadlocks.

To calculate contractual semantics, we need laws to reduce pre-, peri-, and postconditions. These need to cater for compositions of quiescent and final observations using operators like internal choice ($\sqcap$), sequential composition ($;$), and external choice ($\Box$, see §5). So, we prove [17] the following laws for $\mathcal{E}$ and Φ.

Theorem 4.8 (Reactive Relational Compositions)**.**

$$\Phi[\mathit{true}, id, \langle\rangle] = \mathbb{I}_c \tag{1}$$

$$\Phi[s_1, \sigma_1, t_1]\ ;\ \Phi[s_2, \sigma_2, t_2] = \Phi[s_1 \wedge \sigma_1 \dagger s_2, \sigma_2 \circ \sigma_1, t_1 \frown \sigma_1 \dagger t_2] \tag{2}$$

$$\Phi[s_1, \sigma_1, t_1]\ ;\ \mathcal{E}[s_2, t_2, E] = \mathcal{E}[s_1 \wedge \sigma_1 \dagger s_2, t_1 \frown \sigma_1 \dagger t_2, \sigma_1 \dagger E] \tag{3}$$

$$\Phi[s_1, \sigma_1, t_1] \lhd c \rhd \Phi[s_2, \sigma_2, t_2] = \Phi[s_1 \lhd c \rhd s_2, \sigma_1 \lhd c \rhd \sigma_2, t_1 \lhd c \rhd t_2] \tag{4}$$

$$\mathcal{E}[s_1, t_1, E_1] \lhd c \rhd \mathcal{E}[s_2, t_2, E_2] = \mathcal{E}[s_1 \lhd c \rhd s_2, t_1 \lhd c \rhd t_2, E_1 \lhd c \rhd E_2] \tag{5}$$

$$\left(\bigwedge_{i \in I} \mathcal{E}[s(i), t, E(i)]\right) = \mathcal{E}\left[\bigwedge_{i \in I} s(i), t, \bigcup_{i \in I} E(i)\right] \tag{6}$$

$$\left(\bigvee_{i \in I} \mathcal{E}[s(i), t, E(i)]\right) = \mathcal{E}\left[\bigvee_{i \in I} s(i), t, \bigcap_{i \in I} E(i)\right] \tag{7}$$

$$\Phi[s, \sigma, t]^* = \bigsqcap_{n \in \mathbb{N}} \Phi\left[\bigwedge_{i \le n} (\sigma^i \dagger s), \sigma^n, \prod_{j < n} (\sigma^j \dagger t)\right] \tag{8}$$

Law (1) gives the meaning of Φ with a trivial precondition, state update, and empty trace: it is simply the reactive identity. Law (2) states that the composition of two terminated observations results in the conjunction of the state conditions, composition of the state updates, and concatenation of the traces. It is necessary to apply the initial state update σ_1 as a substitution to both the second state condition (s_2) and the trace expression (t_2). Law (3) is similar, but accounts for the enabled events rather than state updates. Laws (2) and (3) are required because of Theorem 2.15-(3), which sequentially composes a pericondition with a postcondition, and a postcondition with a postcondition.

Laws (4) and (5) show how conditional distributes through the operators. Law (6) shows that a conjunction of intermediate observations with a common trace corresponds to the conjunction of the state conditions, and the union of the enabled events. It is needed for external choice, which conjoins the periconditions (see §5). Law (7) shows the dual case of (6): when taking a choice of periconditions, we have the disjunction of all the state conditions, and intersection of all enabled events.

Finally, (8) gives the meaning of an iterated final observation. The nondeterministic choice over $n \in \mathbb{N}$ denotes the number of iterations. Inside, the Φ operator distributes iteration though the condition, state updates, and trace. Here, f^n is iterated function composition ($f \circ f \circ f \cdots$), and $\prod$ is iterated concatenation:

$$\prod_{i<n} xs(i) \triangleq xs(0) \frown xs(1) \frown \cdots \frown xs(n-1)$$

The condition of an iterated final observation requires that s holds whenever σ is applied as a substitution i times, where $i \leq n$. The state update applies σ a total of n times in sequence. The trace expression concatenates t a total of n times, and each instance has the state update applied $j < n$ times.

We can now use these laws, along with Theorem 2.15, to calculate the semantics of processes, and to prove equality and refinement conjectures, as we illustrate below.

Example 4.9. We show that $(x{:=}1 \,;\, \mathbf{Do}(a.x) \,;\, x := x{+}2) = (\mathbf{Do}(a.1) \,;\, x{:=}3)$. By applying Definition 4.7 and Theorems 2.15-(3), 4.8, 4.11, both sides reduce to $[\,\vdash \mathcal{E}[\mathit{true}, \langle\rangle, \{a.1\}] \mid \Phi[\mathit{true}, \{x \mapsto 3\}, \langle a.1\rangle]\,]$, which has a single quiescent state, waiting for event $a.1$, and a single final state, where $a.1$ has occurred and state variable x has been updated to 3. We calculate the left-hand side below.

$$
\begin{aligned}
&(x{:=}1 \,;\, \mathbf{Do}(a.x) \,;\, x := x+2) \\
&= \left(\begin{array}{l} [\,\vdash \mathbf{false} \mid \Phi[\mathit{true}, (\!|x \mapsto 1|\!), \langle\rangle]\,] \,; \\ [\,\vdash \mathcal{E}[\mathit{true}, \langle\rangle, \{a.x\}] \mid \Phi[\mathit{true}, \mathit{id}, \langle a.x\rangle]\,] \,; \\ [\,\vdash \mathbf{false} \mid \Phi[\mathit{true}, (\!|x \mapsto x+1|\!), \langle\rangle]\,] \end{array} \right) && [\text{Def. 4.7}] \\
&= \left(\begin{array}{l} \left[\begin{array}{l|l} \vdash & \mathbf{false} \vee \\ & \Phi[\mathit{true}, (\!|x \mapsto 1|\!), \langle\rangle] \,;\, \mathcal{E}[\mathit{true}, \langle\rangle, \{a.x\}] \end{array} \;\middle|\; \begin{array}{l} \Phi[\mathit{true}, (\!|x \mapsto 1|\!), \langle\rangle] \,; \\ \Phi[\mathit{true}, \mathit{id}, \langle a.x\rangle] \end{array} \right] \,; \\ [\,\vdash \mathbf{false} \mid \Phi[\mathit{true}, (\!|x \mapsto x+1|\!), \langle\rangle]\,] \end{array} \right) && [\text{Thm. 2.15}] \\
&= \left([\,\vdash \mathcal{E}[\mathit{true}[1/x], \langle\rangle[1/x], \{a.x\}[1/x]] \mid \Phi[\mathit{true}[1/x], (\!|x \mapsto 1|\!), \langle a.x\rangle[1/x]]\,] \,; \right. \\
&\quad \left. [\,\vdash \mathbf{false} \mid \Phi[\mathit{true}, (\!|x \mapsto x+1|\!), \langle\rangle]\,] \right) && [\text{Thm. 4.8}] \\
&= \left(\begin{array}{l} [\,\vdash \mathcal{E}[\mathit{true}, \langle\rangle, \{a.1\}] \mid \Phi[\mathit{true}, (\!|x \mapsto 1|\!), \langle a.1\rangle]\,] \,; \\ [\,\vdash \mathbf{false} \mid \Phi[\mathit{true}, (\!|x \mapsto x+1|\!), \langle\rangle]\,] \end{array} \right) \\
&= \left[\begin{array}{l|l} \vdash & \mathcal{E}[\mathit{true}, \langle\rangle, \{a.1\}] \vee \\ & \Phi[\mathit{true}, (\!|x \mapsto 1|\!), \langle a.1\rangle] \,;\, \mathbf{false} \end{array} \;\middle|\; \begin{array}{l} \Phi[\mathit{true}, (\!|x \mapsto 1|\!), \langle a.1\rangle] \,; \\ \Phi[\mathit{true}, (\!|x \mapsto x+1|\!), \langle\rangle] \end{array} \right] && [\text{Thm. 2.15}] \\
&= [\,\vdash \mathcal{E}[\mathit{true}, \langle\rangle, \{a.1\}] \mid \Phi[\mathit{true}[1/x], (\!|x \mapsto x+1|\!) \circ (\!|x \mapsto 1|\!), \langle a.1\rangle]\,] && [\text{Thm. 4.8}] \\
&= [\,\vdash \mathcal{E}[\mathit{true}, \langle\rangle, \{a.1\}] \mid \Phi[\mathit{true}, \{x \mapsto 3\}, \langle a.1\rangle]\,]
\end{aligned}
$$

In the first step, we expand out the definitions of the three sequential actions using Definition 4.7. In the second step, we employ Theorem 2.15 to calculate the sequential composition of the first two contracts. In the third step, we use Theorem 4.8 to calculate the resulting composite peri- and postconditions, which in

particular pushes the initial substitution into both the quiescent and terminated observations of the second contract. In the fourth step, we apply the resulting substitutions to complete composition of the first two contracts. In the remaining steps, we apply the same theorems again to compose with the third contract. □

This proof can be automated using a single invocation of the rdes-eq tactic [14] in Isabelle/UTP, which implements our calcuational proof strategy[6]. We can also use our calculation theorems, with the help of rdes-eq, to prove a number of general laws, which would otherwise require a complex manual proof [17].

Theorem 4.10 (Stateful Failures-Divergences Laws)**.**

$$\langle\sigma\rangle_{\mathsf{c}} \mathbin{;} [\, P_1 \vdash P_2 \mid P_3 \,] = [\, \sigma \dagger P_1 \vdash \sigma \dagger P_2 \mid \sigma \dagger P_3 \,] \quad (1)$$

$$\langle\sigma\rangle_{\mathsf{c}} \mathbin{;} \mathbf{Do}(e) = \mathbf{Do}(\sigma \dagger e) \mathbin{;} \langle\sigma\rangle_{\mathsf{c}} \quad (2)$$

$$x := v \mathbin{;} e \to P = e[v/x] \to x := v \mathbin{;} P \quad (3)$$

$$a \to (P \sqcap Q) = (a \to P) \sqcap (a \to Q) \quad (4)$$

$$\langle\sigma\rangle_{\mathsf{c}} \mathbin{;} \langle\rho\rangle_{\mathsf{c}} = \langle\rho \circ \sigma\rangle_{\mathsf{c}} \quad (5)$$

$$\mathbf{Stop} \mathbin{;} P = \mathbf{Stop} \quad (6)$$

Law (1) shows how a leading assignment distributes substitutions through a contract. Laws (2) and (5) are consequences of Law (1). Law (2) shows that an assignment can be pushed through an event by applying the substitution to the event expression. Law (3) is a further consequence of Law 1 that shows the case for a singleton assignment and a prefixed action. Law (4) shows that a prefix event distributes from the left through nondeterministic choice. Law (5) shows that composing two assignments yields a single assignment where the two substitution functions are composed. Effectively, this law shows the correspondence between functional and relational composition for deterministic relations represented by assignments. Finally, Law (6) shows that the deadlock action, ***Stop***, is a left annihilator.

So far, the reactive contracts we have considered have all contained trivial preconditions. However, divergence is a useful modelling technique that allows us to model unspecified or unpredictable behaviour, when certain assumptions are violated. We consider, for example, the simple action $a \to \mathbf{Skip} \,\Box\, b \to \mathbf{Chaos}$. If event a occurs, then it terminates, and if b occurs it diverges. The behaviour following the occurrence of a is predictable (termination), but the behaviour following the occurrence of b is unpredictable.

In order to calculate contracts for actions of this form, we need to consider the weakest precondition operator $\mathbf{wlp}_r$. So far, we have only considered simple formulae of the form $P\ \mathbf{wlp}_r\ \mathbf{true}_r = \mathbf{true}_r$; we now supply theorems for more sophisticated preconditions. Theorem 2.15-(3) requires that, in a sequential composition $P \mathbin{;} Q$, we need to show that the postcondition of contract P satisfies the precondition of contract Q. We, consider for example the following partial calculation of $b \to \mathbf{Chaos}$.

$$\begin{aligned} b \to \mathbf{Chaos} &= \mathbf{Do}(b) \mathbin{;} \mathbf{Chaos} \\ &= [\, \vdash \mathcal{E}[\mathit{true}, \langle\rangle, \{b\}] \mid \Phi[\mathit{true}, \mathit{id}, \langle b\rangle] \,] \mathbin{;} [\, \mathbf{false} \vdash \mathbf{false} \mid \mathbf{false} \,] \\ &= [\, \Phi[\mathit{true}, \mathit{id}, \langle b\rangle]\ \mathbf{wlp}_r\ \mathbf{false} \vdash \mathcal{E}[\mathit{true}, \langle\rangle, \{b\}] \vee \mathbf{false} \mid \mathbf{false} \,] \\ &= [\, \Phi[\mathit{true}, \mathit{id}, \langle b\rangle]\ \mathbf{wlp}_r\ \mathbf{false} \vdash \mathcal{E}[\mathit{true}, \langle\rangle, \{b\}] \mid \mathbf{false} \,] \end{aligned}$$

The postcondition is ***false***, so this action has no final state. It can be quiescent, waiting for b to occur. We cannot, however, calculate the precondition yet; it states that the trace $\langle b\rangle$ should never occur.

In general, the precondition of a reactive contract uses the weakest precondition of a previously applied postcondition. Theorem 4.8 explains how to eliminate most composition operators in a contract's postcondition, but not disjunction ($\vee$). Postconditions are, therefore, typically expressed as disjunctions of the Φ operator. So, our weakest precondition calculus needs to handle disjunctions of Φ terms.

[6] Several examples of this can be found in our respository, using the link to the right.

Theorem 4.11 (Reactive Preconditions)**.**

$$\Phi[s, \sigma, t] \ \textbf{\textit{wlp}}_r \ \textbf{\textit{false}} = \mathcal{I}[s, t] \tag{1}$$
$$\Phi[s_1, \sigma, t_1] \ \textbf{\textit{wlp}}_r \ \mathcal{I}[s_2, t_2] = \mathcal{I}[s_1 \wedge \sigma \dagger s_2, t_1 \frown (\sigma \dagger t_2)] \tag{2}$$
$$\mathcal{I}[\mathit{false}, t] = \textbf{\textit{true}}_r \tag{3}$$
$$\mathcal{I}[\mathit{true}, \langle\rangle] = \textbf{\textit{false}} \tag{4}$$
$$\mathcal{I}[s_1, t] \wedge \mathcal{I}[s_2, t] = \mathcal{I}[s_1 \vee s_2, t] \tag{5}$$
$$\mathcal{I}[s_1, t] \vee \mathcal{I}[s_2, t] = \mathcal{I}[s_1 \wedge s_2, t] \tag{6}$$

We recall that $\mathcal{I}[s, t]$ means that, if s is satisfied in the current state, then the action can only perform traces that do not have t as a prefix, or else divergence will result. Law (1) calculates the weakest precondition under which $\Phi[s, \sigma, t]$ achieves ***false***, which is impossible. Consequently, we must require that the trace t never occurs, when the state initially satisfies s. With this law, we can complete the contract calculation of $b \rightarrow$ ***Chaos*** to obtain

$$[\,\mathcal{I}[\mathit{true}, \langle b \rangle] \vdash \mathcal{E}[\mathit{true}, \langle\rangle, \{b\}] \mid \textbf{\textit{false}}\,]$$

whose precondition assumes that event b does not occur initially. Law (2) considers the final observation specified $\mathcal{I}[s_2, t_2]$. If we start in a state that satisifes s_1 and s_2 with state update σ applied, then an upper bound on the trace is $t_1 \frown (\sigma \dagger t_2)$, which also inserts the state update.

The remaining laws are for different compositions for $\mathcal{I}$. Law (3) shows that if an assumption's condition is *false*, then it reduces to the reactive precondition $\textbf{\textit{true}}_r$. Conversely, law (4) shows that if the condition is *true*, but the trace is $\langle\rangle$, then this is ***false***, since all traces are disallowed. The remaining two laws show the effect of conjunction and disjunction on assumptions sharing a trace expression.

This completes the calculational approach for the core sequential programming operators. In the next section, we extend our proof approach to support external choice [26, 38].

5. External Choice and Productivity

In this section we consider reasoning about external choice [26, 38], and characterise the class of productive contracts [14], which are also essential in verifying recursive and iterative reactive programs.

An external choice $P \,\Box\, Q$ resolves whenever either P or Q engages in an event or terminates. Thus, its semantics requires that we filter observations with a non-empty trace. We introduce healthiness condition $\textbf{\textit{R4}}(P) \triangleq (P \wedge \mathit{tt} > \langle\rangle)$, whose fixed points strictly increase the trace, and its dual $\textbf{\textit{R5}}(P) \triangleq (P \wedge \mathit{tt} = \langle\rangle)$ where the trace is unchanged. We use these to define indexed external choice.

Definition 5.1 (Indexed External Choice)**.**

$$\Box\, i \in I \bullet [\,P_1(i) \vdash P_2(i) \mid P_3(i)\,] \triangleq$$
$$[\,\textstyle\bigwedge_{i \in I} P_1(i) \vdash (\bigwedge_{i \in I} \textbf{\textit{R5}}(P_2(i))) \vee (\bigvee_{i \in I} \textbf{\textit{R4}}(P_2(i))) \mid \bigvee_{i \in I} P_3(i)\,]$$

This generalises the binary definition [28, 35], and recasts our definition in [14] for calculation. Like nondeterministic choice, the precondition requires that all constituent preconditions are satisfied. In the pericondition, ***R4*** and ***R5*** filter all observations. We take the conjunction of all ***R5*** behaviours: no event has occurred, and all branches are offering an event. We take the disjunction of all ***R4*** behaviours: an event occurred, and the choice is resolved. In the postcondition the choice is resolved, either by synchronisation or termination, and so we take the disjunction of all constituent postconditions. Since unbounded choice is supported, we can denote indexed input prefix for any size of input domain A.

$$a?x : A \rightarrow P(x) \ \triangleq \ \Box\, x \in A \bullet a.x \rightarrow P(x)$$

We next show how ***R4*** and ***R5*** filter the various reactive relational operators.

Theorem 5.2 (Trace Filtering)**.**

$$\textbf{\textit{R4}}\left(\bigvee_{i\in I} P(i)\right) = \bigvee_{i\in I} \textbf{\textit{R4}}(P(i)) \qquad \textbf{\textit{R5}}\left(\bigvee_{i\in I} P(i)\right) = \bigvee_{i\in I} \textbf{\textit{R5}}(P(i))$$

$$\textbf{\textit{R4}}(\Phi[s,\sigma,\langle\rangle]) = \textbf{\textit{false}} \qquad \textbf{\textit{R5}}(\mathcal{E}[s,\langle\rangle,E]) = \mathcal{E}[s,\langle\rangle,E]$$

$$\textbf{\textit{R4}}(\Phi[s,\sigma,\langle a,...\rangle]) = \Phi[s,\sigma,\langle a,...\rangle] \qquad \textbf{\textit{R5}}(\mathcal{E}[s,\langle a,...\rangle,E]) = \textbf{\textit{false}}$$

Both operators distribute through $\bigvee$. Relations that produce an empty trace yield ***false*** under ***R4*** and are unchanged under ***R5***. Relations that produce a non-empty trace yield ***false*** for ***R5***, and are unchanged under ***R4***. We can now filter the behaviours that do and do not resolve the choice, as exemplified below.

Example 5.3. We consider the calculation of the action $a \to b \to \textbf{\textit{Skip}} \;\Box\; c \to \textbf{\textit{Skip}}$. The left branch has two quiescent observations, one waiting for a, and one for b having performed a: its pericondition is $\mathcal{E}[true,\langle\rangle,\{a\}] \vee \mathcal{E}[true,\langle a\rangle,\{b\}]$. Application of ***R5*** to this yields the first disjunct, since the trace has not increased, and application of ***R4*** yields the second disjunct. For the right branch there is one quiescent observation, $\mathcal{E}[true,\langle\rangle,\{c\}]$, which contributes an empty trace and is ***R5*** only. The overall pericondition is

$$(\mathcal{E}[true,\langle\rangle,\{a\}] \wedge \mathcal{E}[true,\langle\rangle,\{c\}]) \vee \mathcal{E}[true,\langle a\rangle,\{b\}]$$

which is simply $\mathcal{E}[true,\langle\rangle,\{a,c\}] \vee \mathcal{E}[true,\langle a\rangle,\{b\}]$. □

By calculation, we can now prove that $(\llbracket \textbf{\textit{NCSP}} \rrbracket_{\mathsf{H}}, \Box, \textbf{\textit{Stop}})$ forms a commutative and idempotent monoid, and ***Chaos***, the divergent program, is its annihilator. Sequential composition also distributes from the left and right through external choice, but only when the choice branches are productive [14].

Definition 5.4. A contract $[\,P_1 \vdash P_2 \mid P_3\,]$ is productive when P_3 is ***R4***-healthy.

A productive contract is one that, whenever it terminates, strictly increases the trace. For example $a \to \textbf{\textit{Skip}}$ is productive, but ***Skip*** is not. Constructs that do not terminate, like ***Chaos***, are also productive. The imposition of ***R4*** ensures that only final observations that increase the trace, or are ***false***, are admitted.

We define healthiness condition ***PCSP***, which extends ***NCSP*** with productivity. We also define ***ICSP***, which formalises instantaneous contracts where the postcondition is ***R5***-healthy and the pericondition is ***false***. Both healthiness conditions are defined below.

Definition 5.5 (Productive and Instantaneous Healthiness Conditions)**.**

$$\textbf{\textit{Productive}}(P) \triangleq P \otimes [\,\textbf{\textit{true}}_r \vdash \textbf{\textit{true}} \mid tr < tr'\,]$$

$$\textbf{\textit{ISRD1}}(P) \triangleq P \otimes [\,\textbf{\textit{true}}_r \vdash \textbf{\textit{false}} \mid tr' = tr\,]$$

$$\textbf{\textit{PCSP}} \triangleq \textbf{\textit{Productive}} \circ \textbf{\textit{NCSP}}$$

$$\textbf{\textit{ICSP}} \triangleq \textbf{\textit{ISRD1}} \circ \textbf{\textit{NCSP}}$$

Here, the $\otimes$ operator combines two contracts by conjoining the pre-, peri-, and postconditions, that is:

$$[\,P_1 \vdash P_2 \mid P_3\,] \otimes [\,Q_1 \vdash Q_2 \mid Q_3\,] = [\,P_1 \wedge Q_1 \vdash P_2 \wedge Q_2 \mid P_3 \wedge Q_3\,]$$

Healthiness condition ***Productive*** leaves the pre- and periconditions unchanged, but conjoins the postcondition with $tr < tr'$ – the trace must strictly increase. ***ISRD1*** similarly leaves the precondition unchanged, but coerces the pericondition to ***false*** to remove quiescent observations. The postcondition is conjoined with $tr' = tr$ to disallow events from occuring. We then define ***PCSP*** and ***ICSP*** by composing the former two functions with ***NCSP***. These healthiness conditions obey the following equations for reactive contracts.

Theorem 5.6 (***PCSP*** and ***ICSP*** contracts)**.**

$$\textbf{\textit{PCSP}}([\,P_1 \vdash P_2 \mid P_3\,]) = [\,P_1 \vdash P_2 \mid \textbf{\textit{R4}}(P_3)\,]$$

$$\textbf{\textit{ICSP}}([\,P_1 \vdash P_2 \mid P_3\,]) = [\,P_1 \vdash \textbf{\textit{false}} \mid \textbf{\textit{R5}}(P_3)\,])$$

Application of **PCSP** to a reactive contract is equivalent to applying **R4** to its postcondition. Application of **ICSP** to a reactive contract makes the pericondition **false**, and applies **R5** to its postcondition, meaning it can contribute no events. Both **Skip** and $x := v$ are **ICSP**-healthy as they do not contribute to the trace and have no intermediate observations. This allows us to prove the following laws.

Theorem 5.7 (External Choice Distributivity)**.**

$$(\Box\, i \in I \bullet P(i)) \,;\, Q \;=\; \Box\, i \in I \bullet (P(i) \,;\, Q) \qquad [\textit{if, } \forall\, i \in I, P(i) \textit{ is } \textbf{PCSP} \textit{ healthy}]$$
$$P \,;\, (\Box\, i \in I \bullet Q(i)) \;=\; \Box\, i \in I \bullet (P \,;\, Q(i)) \qquad [\textit{if } P \textit{ is } \textbf{ICSP} \textit{ healthy}]$$

The first law follows because every $P(i)$, being productive, must resolve the choice before terminating, and thus it is not possible to reach Q before this occurs. It generalises the standard guarded choice distribution law for CSP [28, page 211]. The second law follows for the converse reason: since P cannot resolve the choice with any of its behaviour, it is safe to execute it first. Productivity also forms an important criterion for guarded recursion that we use in §6 to calculate fixed points.

PCSP is closed under several operators.

Theorem 5.8 (Productive Constructions)**.**

- **Miracle**, **Chaos**, **Stop**, *and* **Do**(a) *are all* **PCSP** *healthy;*
- $b \,\&\, P$ *is* **PCSP** *if* P *is* **PCSP***;*
- $P \,;\, Q$ *is* **PCSP** *if either* P *or* Q *is* **PCSP***;*
- $\sqcap\, i \in I \bullet P(i)$ *is* **PCSP** *if, for all* $i \in I$*,* $P(i)$ *is* **PCSP***;*
- $\Box\, i \in I \bullet P(i)$ *is* **PCSP** *if, for all* $i \in I$*,* $P(i)$ *is* **PCSP***.*

With these results, calculation of external choice is now supported, and a notion of productivity, with relevant laws, is defined. In the next section we use the latter for calculation of while-loops.

6. While Loops and Reactive Invariants

Iterative programs can be constructed using the reactive while loop.

$$b \circledast P \;\triangleq\; (\mu\, X \bullet P \,;\, X \lhd b \rhd \textbf{Skip}).$$

We use the weakest fixed-point so that an infinite loop with no observable activity corresponds to the divergent action **Chaos**, rather than **Miracle**. For example, $(\mathit{true} \circledast x{:=}x{+}1) = \textbf{Chaos}$. The *true* condition is not a problem because, unlike its imperative counterpart, the reactive while loop pauses for interaction with its environment, and therefore infinite executions are observable and potentially useful.

In order to reason about iteration, we need additional calculational laws. A fixed-point $(\mu\, X \bullet F(X))$ is guarded provided at least one event is contributed to the trace by F prior to it reaching X. For instance, $\mu\, X \bullet a \rightarrow X$ is guarded, but $\mu\, X \bullet y := 1 \,;\, X$ is not. Hoare and He's theorem [28, theorem 8.1.13, page 206] states that if F is guarded, then there is a unique fixed-point and hence $(\mu\, X \bullet F(X)) = (\nu X \bullet F(X))$. So, provided F is continuous, we can invoke Kleene's fixed-point theorem to calculate νF. Our previous result [14] shows that if P is productive, then $\lambda\, X \bullet P \,;\, X$ is guarded, and so we can calculate its fixed-point. We now generalise this for the function above.

Theorem 6.1. *If* P *is productive, then* $(\mu\, X \bullet P \,;\, X \lhd b \rhd \textbf{Skip})$ *is guarded.*

Proof. In addition to our previous theorem [14], we use the following properties:

- If X is not mentioned in P then $\lambda\, X \bullet P$ is guarded;
- If F and G are both guarded, then $\lambda\, X \bullet F(X) \lhd b \rhd G(X)$ is guarded. □

This allows us to convert the fixed-point into an iterative form. In particular, we can prove the following theorem that expresses it in terms of the Kleene star.

Theorem 6.2. *If P is* ***PCSP*** *healthy then* $b \circledast P = ([b]_{\mathsf{C}} \,;\, P)^* \,;\, [\neg b]_{\mathsf{C}}$.

Here, $[b]_{\mathsf{C}} \triangleq \mathbf{Skip} \lhd b \rhd \mathbf{Miracle}$ denotes a reactive program state test (cf. $[b]_c$, which is for reactive relations). This theorem is similar to the usual imperative definition in Kleene Algebra with Tests [31, 1, 21]. It relies on productivity of P, though the condition b can be used to guard P and therefore prune away any unproductive behaviours that violate b. In Theorem 6.2, P is executed multiple times when b is true initially, but each run concludes when b is false. However, due to the embedding of reactive behaviour, there is more going on than meets the eye; the next theorem shows how to calculate an iterative contract.

Theorem 6.3. *If R is* ***R4*** *healthy then*

$$b \circledast [P_1 \vdash P_2 \mid P_3] = [\,([b]_c \,;\, P_3)^* \;\mathbf{wlp}_r\, (b \Rightarrow P_1) \vdash ([b]_c \,;\, P_3)^* \,;\, [b]_c \,;\, P_2 \mid ([b]_c \,;\, P_3)^* \,;\, [\neg b]_c\,]$$

The precondition requires that any number of P_3 iterations, where b is initially true, satisfies P_1. This ensures that the contract does not violate its own precondition from one iteration to the next. The pericondition states that intermediate observations have P_3 executing several times, with b true, and following this b remains true and the contract is quiescent (P_2). The postcondition is similar, but after several iterations, b becomes false and the loop terminates, which is the standard relational form of a while loop.

Theorem 6.3 can be used to prove a refinement introduction law for the reactive while loop. This employs "reactive invariant" relations, which describe how both the trace and state variables are permitted to evolve.

Theorem 6.4. $[I_1 \vdash I_2 \mid I_3] \sqsubseteq b \circledast [Q_1 \vdash Q_2 \mid Q_3]$ *provided that:*

1. *Q_3 is* ***R4****-healthy, so that the reactive contract is productive;*
2. *the assumption is weakened* $(([b]_c \,;\, Q_3)^* \;\mathbf{wlp}_r\, (b \Rightarrow Q_1) \sqsubseteq I_1)$*;*
3. *when b holds, Q_2 establishes the I_2 pericondition invariant* $(I_2 \sqsubseteq ([b]_c \,;\, Q_2))$ *and, Q_3 maintains it* $(I_2 \sqsubseteq [b]_c \,;\, Q_3 \,;\, I_2)$*;*
4. *postcondition invariant I_3 is established when b is false* $(I_3 \sqsubseteq [\neg b]_c)$ *and Q_3 establishes it when b is true* $(I_3 \sqsubseteq [b]_c \,;\, Q_3 \,;\, I_3)$*.*

Proof. By application of refinement introduction, with Theorems 3.2-(3) and 6.3. □

Theorem 6.4 shows the conditions under which an iterated contract satisfies an invariant contract $[I_1 \vdash I_2 \mid I_3]$. Relations I_2 and I_3 are reactive invariants that must hold in quiescent and final observations. Both can refer to st and tt, I_2 can additionally refer to ref', and I_3 to st'. There is no need to supply a variant since productivity guarantees the existence of a descending approximation chain for the iteration [14]. Combined with the results from §4 and §5, this result forms the basis for a proof strategy for iterative reactive programs.

7. Parallel Composition

In this section we extend our calculational approach to one of the most challenging operators: parallel composition. We build on the parallel-by-merge scheme [28], $P \parallel_M Q$, where the semantics is expressed in terms of a merge predicate M that describes how the observations of each parallel program, P and Q, should be merged [20]. We create a specialised law for parallel composition of stateful-failure reactive designs, that merges the pre-, peri-, and postconditions, and show how the $\mathcal{I}$, Φ, and $\mathcal{E}$ operators are merged. We use the strategy on a number of examples, and prove characteristic algebraic theorems for parallel composition.

The contents of the first two subsections, §7.1 and §7.2, contain some restatements of theorems we have previously proved [14], which are included for the purpose of self-containment and explanation. §7.3 onwards, which specialises to stateful-failure reactive designs, is entirely novel.

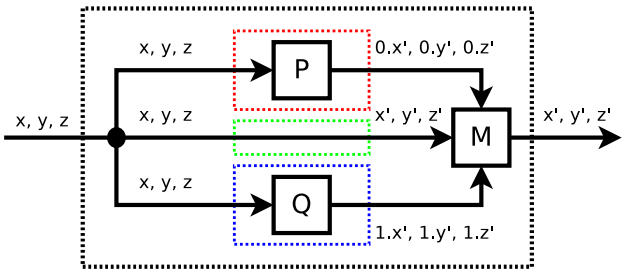

Figure 2: Parallel-by-merge dataflow

7.1. Parallel-by-Merge

We recall the parallel-by-merge operator [14]. It employs the $\lceil P \rceil_n$ construct, which renames all dashed variables of P by adding an index n, so that they can be distinguished from other indexed variables[7].

Definition 7.1 (Parallel-by-Merge)**.** $P \parallel_M Q \triangleq (\lceil P \rceil_0 \wedge \lceil Q \rceil_1 \wedge \mathsf{v}' = \mathsf{v}) \mathbin{;} M$

This operator effectively 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 input v, and merges them into a single output. Here, v is a special variable that denotes the entirety of the state space. The dataflow of this operator is depicted in Figure 2, which illustrates the definition for an example with three variables, x, y, and z.

The outputs of P and Q are distinguished by a numeral prefix. If P and Q act on an observation space $\mathcal{S}$, then M is a heterogeneous relation of type $[\mathcal{S} \times \mathcal{S} \times \mathcal{S}, \mathcal{S}]\mathit{urel}$, which relates three input copies of $\mathcal{S}$ with a single output $\mathcal{S}$. The merge predicate therefore refers to variables from P, using the $0.x$ notation, variables from Q, using $1.x$, initial variables, as usual written as x, and final variables, as x'.

A substantial advantage of using parallel-by-merge is that several theorems can be proven for the generic operator. Below, we highlight some of the most important theorems.

Theorem 7.2 (Parallel-by-Merge Laws)**.**

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

$$\mathbf{false} \parallel_M P = \mathbf{false} \qquad P \parallel_M \mathbf{false} = \mathbf{false}$$

$$P_1 \sqsubseteq P_2 \wedge Q_1 \sqsubseteq Q_2 \Rightarrow (P_1 \parallel_M Q_1) \sqsubseteq (P_1 \parallel_M Q_1)$$

Parallel-by-merge distributes through nondeterministic choice ($\prod$) from both the left and right, regardless of the merge predicate M. Since $\prod$ corresponds to $\exists$ and also $\vee$, we can similarly distribute through an existential quantification and a disjunction. The miraculous relation $\mathbf{false}$ is both a left and right annihilator for parallel composition, which is also monotonic with respect to refinement in both arguments.

Parallel-by-merge may or may not be commutative, depending on the merge predicate. A helpful scheme can be used for proving that parallel composition is commutative, which reduces this to a property of the merge predicate. We adopt a similar approach to [28], but give an account that is more algebraic in nature. We first define the following auxiliary operator.

Definition 7.3 (Merge Swap)**.** $\mathbf{sw} \triangleq \mathsf{v}' = \mathsf{v} \wedge 0.\mathsf{v}' = 1.\mathsf{v} \wedge 1.\mathsf{v}' = 0.\mathsf{v}$

The relation $\mathbf{sw}$ swaps the outputs from the left- and right-hand sides, whilst keeping the initial values (v) the same. Using $\mathbf{sw}$, we can prove the following property of parallel-by-merge.

[7] These are called "separating simulations" in [28, page 172], and are denoted using special relations called $U0$ and $U1$.

Theorem 7.4 (Parallel-by-Merge Swap). $P \parallel_{\mathbf{sw};M} Q = Q \parallel_M P$

This theorem shows that precomposing a merge predicate with $\mathbf{sw}$ effectively commutes the arguments P and Q. A corollary of this [20], given below, shows how this can be used to demonstrate commutativity.

Theorem 7.5. $P \parallel_M Q = Q \parallel_M P$ *provided that* $\mathbf{sw} ; M = M$

This theorem shows how proof of commutativity can be reduced to a property of the merge predicate. Specifically, if swapping the order of the inputs to the merge predicate has no effect then it is a symmetric merge, and consequently parallel composition is commutative.

7.2. Parallel Reactive Designs

In previous work [14], we have used parallel-by-merge to prove a general theorem for composing reactive designs. As for the sequential operators, this develops operators that respectively merge the pre-, peri-, and postconditions of the corresponding reactive contract. The theorem below, reproduced from [14], shows how we may calculate a parallel reactive contract using these operators.

Theorem 7.6 (Reactive Design Parallel Composition).

$$[\, P_1 \vdash P_2 \mid P_3 \,] \parallel^M_{\mathbf{R}} [\, Q_1 \vdash Q_2 \mid Q_3 \,] =$$
$$\left[\begin{array}{l|l|l} \begin{array}{l}(P_1 \Rightarrow_r P_2)\ \mathbf{wr}_M\ Q_1 \wedge \\ (P_1 \Rightarrow_r P_3)\ \mathbf{wr}_M\ Q_1 \wedge \\ (Q_1 \Rightarrow_r Q_2)\ \mathbf{wr}_M\ P_1 \wedge \\ (Q_1 \Rightarrow_r Q_3)\ \mathbf{wr}_M\ P_1\end{array} & \begin{array}{l}P_2 \parallel^M_{\mathbf{E}} Q_2 \vee \\ P_3 \parallel^M_{\mathbf{E}} Q_2 \vee \\ P_2 \parallel^M_{\mathbf{E}} Q_3\end{array} & P_3 \parallel_M Q_3 \end{array}\right]$$

This complex law describes how the pre-, peri-, and postconditions are merged by the parametric reactive design parallel composition operator $\parallel^M_{\mathbf{R}}$. Here, M is an "inner merge predicate" [14], which needs to deal only with observational variables like tt, st, and ref; the variables ok and $wait$ having already been merged by $\parallel_{\mathbf{R}}$, which constructs the "outer merge predicate", to which parallel-by-merge is applied.

The precondition of the composite contract in Theorem 7.6 captures the possible divergent behaviours that both P and Q permit. There are four conjuncts in the precondition, as we require that neither the peri- nor the postcondition can permit divergent behaviour disallowed by its opposing precondition.

The predicate $A\ \mathbf{wr}_M\ B$, standing for "weakest rely", is a reactive condition that describes the weakest context in which reactive relation A does not violate the reactive condition B. It is analogous to the reactive weakest precondition operator, $\mathbf{wlp}_r$ (outlined in §2.3), but is defined with respect to parallel composition rather than sequential composition. Specifically, whereas $\mathbf{wlp}_r$ gives the weakest condition in a sequential context, $\mathbf{wr}_M$ gives the weakest condition in a parallel context. It obeys several related laws shown below.

Theorem 7.7 (Weakest Rely Laws).

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

The laws show that (1) a miraculous 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. These results are similar to those for $\mathbf{wlp}_r$

The pericondition in Theorem 7.6 is a disjunction of three terms that calculate possible quiescent merged behaviours. Parallel composition is quiescent when at least one of P and Q is quiescent, and so the three conjuncts characterise quiescence in both, in Q only, and in P only, respectively. The $\parallel^M_{\mathbf{E}}$ operator is an intermediate merge operator, which restricts access to state (see [14, §6.6]). Finally, the overall contract can only terminate when both P and Q do, and so the postcondition simply merges their respective postconditions.

Using these laws, we can show that $\mathbf{Miracle}$ is always a annihilator for parallel composition, regardless of the inner merge predicate [14]. The proof exemplifies the calculational approach for parallel composition

Theorem 7.8. $\textbf{\textit{Miracle}} \parallel^{M}_{R} P = \textbf{\textit{Miracle}}$

Proof.

$$
\begin{aligned}
\textbf{\textit{Miracle}} \parallel^{M}_{R} P &= [\,\textbf{\textit{true}}_r \vdash \textbf{\textit{false}} \mid \textbf{\textit{false}}\,] \parallel^{M}_{R} [\,P_1 \vdash P_2 \mid P_3\,] \\
&= \left[\begin{array}{l} (\textbf{\textit{true}}_r \Rightarrow_r \textbf{\textit{false}})\ \textbf{\textit{wr}}_M\ P_1 \wedge \\ (\textbf{\textit{true}}_r \Rightarrow_r \textbf{\textit{false}})\ \textbf{\textit{wr}}_M\ P_1 \wedge \\ (P_1 \Rightarrow_r P_2)\ \textbf{\textit{wr}}_M\ \textbf{\textit{true}}_r \wedge \\ (P_1 \Rightarrow_r P_3)\ \textbf{\textit{wr}}_M\ \textbf{\textit{true}}_r \end{array} \middle| \begin{array}{l} \textbf{\textit{false}} \parallel^{M}_{E} P_2 \vee \\ \textbf{\textit{false}} \parallel^{M}_{E} P_2 \vee \\ \textbf{\textit{false}} \parallel^{M}_{E} P_3 \end{array} \middle| \ \textbf{\textit{false}} \parallel_{M} P_3 \right] && [7.6] \\
&= \left[\begin{array}{l} \textbf{\textit{false}}\ \textbf{\textit{wr}}_M\ P_1 \wedge \textbf{\textit{false}}\ \textbf{\textit{wr}}_M\ P_1 \wedge \\ \textbf{\textit{true}}_r \wedge \textbf{\textit{true}}_r \end{array} \middle| \ \textbf{\textit{false}} \vee \textbf{\textit{false}} \vee \textbf{\textit{false}} \ \middle| \ \textbf{\textit{false}} \right] && [7.2, 7.7] \\
&= [\ \textbf{\textit{true}}_r \wedge \textbf{\textit{true}}_r \ \vdash\ \textbf{\textit{false}}\ \mid \textbf{\textit{false}}\,] && [7.7] \\
&= [\,\textbf{\textit{true}}_r \vdash \textbf{\textit{false}} \mid \textbf{\textit{false}}\,] \\
&= \textbf{\textit{Miracle}} && \square
\end{aligned}
$$

In this case, the pericondition and the postcondition both reduce to ***false***, since by Theorem 7.2 the merge of any relation with ***false*** reduces to ***false***. The four clauses in the precondition all reduce to $\textbf{\textit{true}}_r$ by the weakest rely laws of Theorem 7.7. Thus the entire relation reduces to the ***Miracle*** contract. We will next specialise this calculational approach to stateful-failure reactive designs.

7.3. Parallel Stateful-Failure Reactive Designs

Our parallel composition operator is adopted from *Circus* [35] and has the general form

$$P \,[\![ns_1 | cs | ns_2]\!]\, Q$$

for actions P and Q, event set $cs \subseteq \mathit{Event}$, and name-sets ns_1 and ns_2. P and Q both act on the same state space $\mathcal{S}$, and have the same event alphabet (Event). Like for parallel composition in CSP, P and Q must synchronise on events contained in cs, but independently engage in events outside cs. Since P and Q also have states, we must describe how to merge their final states. We do not permit sharing, and so require partitioning of the state into two independent regions, characterised by two disjoint variable name sets ns_1 and ns_2. The final state is then the composition of the two regions. We model these name sets using independent lenses [20] from Isabelle/UTP, that is, $ns_1 : V_1 \Longrightarrow \mathcal{S}$, and $ns_2 : V_2 \Longrightarrow \mathcal{S}$, for some V_1 and V_2 (see §2.2).

As usual [35], we define a few abbreviations for the operator.

Definition 7.9 (Parallel Composition Abbreviations)**.**

$$P \,[\![cs]\!]\, Q \triangleq P \,[\![\mathbf{0} | cs | \mathbf{0}]\!]\, Q$$
$$P \,|||\, Q \triangleq P \,[\![\emptyset]\!]\, Q$$

The operator $P \,[\![cs]\!]\, Q$ synchronises on cs, but ignores the final state of both P and Q. It is therefore broadly equivalent to CSP parallel composition when applied to stateless actions. It uses the special $\mathbf{0}$ lens for the name sets, which characterises an empty region of the state space. The interleaving operator $P \,|||\, Q$ synchronises on none of the events, and requires independent activity for P and Q. We denote the general operator using the reactive design parallel-by-merge operator, as shown below.

Definition 7.10 (Parallel Composition)**.** Let $ns_1 : \mathcal{V}_1 \Longrightarrow \mathcal{S}$ and $ns_2 : \mathcal{V}_2 \Longrightarrow \mathcal{S}$ be lenses that characterise disjoint regions, $\mathcal{V}_1$ and $\mathcal{V}_2$, of the state space $\mathcal{S}$ (that is, $ns_1 \bowtie ns_2$), and let cs be a set of events. Parallel composition is then defined as follows:

$$P \,[\![ns_1 | cs | ns_2]\!]\, Q \triangleq P \parallel^{N_C}_{R} Q$$

$$\text{where } N_C \begin{bmatrix} ns_1 \\ cs \\ ns_2 \end{bmatrix} \triangleq \left(\begin{array}{l} tt \in 0.tt \parallel_{cs} 1.tt \\ \wedge\ 0.tt \upharpoonright cs = 1.tt \upharpoonright cs \\ \wedge\ \textit{ref}' \subseteq ((0.\textit{ref} \cup 1.\textit{ref}) \cap cs) \cup ((0.\textit{ref} \cap 1.\textit{ref}) \setminus cs) \\ \wedge\ st' = st \lhd_{ns_1} 0.st \lhd_{ns_2} 1.st \end{array}\right)$$

Here, N_c is an inner merge predicate [14] with arguments ns_1, cs, and ns_2, which we omit when they can be determined from the context. It defines how the traces, states, and refusal sets from P and Q are merged. It is adapted from the original *Circus* merge predicate [36, 35], which also defines the function $t_1 \parallel_{cs} t_2$ that specifies the set of traces obtained by merging traces t_1 and t_2, synchronising on the events in cs. For completeness, we define this recursive function below, adapting slightly the original definition[8] [36].

Definition 7.11 (Trace Merge Function)**.** We define $\parallel_{cs} : \operatorname{seq} E \to \operatorname{seq} E \to \mathbb{P}(\operatorname{seq} E)$ to be the least function that satisfies the following equations:

$$
\begin{aligned}
\langle\rangle \parallel_{cs} \langle\rangle &= \{\langle\rangle\} \\
(e \mathbin{\#} t) \parallel_{cs} \langle\rangle &= (\{\langle\rangle\} \lhd e \in cs \rhd (\{\langle e\rangle\} \frown (t \parallel_{cs} \langle\rangle))) \\
\langle\rangle \parallel_{cs} (e \mathbin{\#} t) &= (\{\langle\rangle\} \lhd e \in cs \rhd (\{\langle e\rangle\} \frown (\langle\rangle \parallel_{cs} t))) \\
(e \mathbin{\#} t_1) \parallel_{cs} (e \mathbin{\#} t_2) &= (\langle e\rangle \frown (t_1 \parallel_{cs} t_2)) \lhd e \in cs \rhd (\{\langle e\rangle\} \frown (t_1 \parallel_{cs} (e \mathbin{\#} t_2) \cup (e \mathbin{\#} t_1) \parallel_{cs} t_2)) \\
(e_1 \mathbin{\#} t_1) \parallel_{cs} (e_2 \mathbin{\#} t_2) &= \left(\begin{array}{l} (\{\langle\rangle\} \lhd e_2 \in cs \rhd (\{\langle e_2\rangle\} \frown ((e_1 \mathbin{\#} t_1) \parallel_{cs} t_2))) \\ \quad \lhd e_1 \in cs \rhd \\ \left(\begin{array}{l} (\{\langle e_1\rangle\} \frown (t_1 \parallel_{cs} (e_2 \mathbin{\#} t_2))) \\ \quad \lhd e_2 \in cs \rhd \\ ((\{\langle e_1\rangle\} \frown (t_1 \parallel_{cs} (e_2 \mathbin{\#} t_2))) \cup (\{\langle e_2\rangle\} \frown ((e_1 \mathbin{\#} t_1) \parallel_{cs} t_2))) \end{array} \right) \end{array} \right) \quad e_1 \neq e_2
\end{aligned}
$$

where $ts_1 \frown ts_2 \triangleq \{t_1 \frown t_2 \mid t_1 \in ts_1 \land t_2 \in ts_2\}$

The trace merge function $t_1 \parallel_{cs} t_2$ produces the set of maximal possible merges from every pair of traces; that is the traces that include the maximum possible number of events from both t_1 and t_2, ordered to reflect synchronisation on th events in cs [36]. If an event is encountered in cs, then both traces must agree to allow this event simultaneously for behaviour to progress. For any other events, all possible interleavings are recorded.

The merge predicate in Definition 7.10 has four conjuncts. The first conjunct states that any permissible trace tt arises from merging the constituent traces $0.tt$ and $1.tt$. The second ensures that the same synchronisations on events from cs occur in both $0.tt$ and $1.tt$ in the same order. Filter function $t \upharpoonright cs$ returns the elements of sequence t that are contained in cs, whilst retaining the order and number of occurrences. A consequence of the second conjunct is that the resulting trace contains, in a suitable order, all the events from both constituents. The third conjunct requires that the overall refusal is either a subset of the set of synchronised events independently refused ($(0.ref \cup 1.ref) \cap cs$), or the non-synchronised events refused by both ($(0.ref \cap 1.ref) \setminus cs$). The fourth conjunct constructs the final state by merging the ns_1 region of P's state, the ns_2 region of Q's state, and the remaining region from the initial state. It uses the lens override operator $s_1 \lhd_{ns} s_2$ from Definition 2.6.

The parallel operator of Definition 7.10 is not, in general, commutative due to its asymmetric partitioning of the state space. However, we can prove a useful theorem of the inner merge predicate.

Theorem 7.12 (Swap Inner Merge)**.** *If* $ns_1 \bowtie ns_2$ *then* $\mathbf{sw} ; N_c \begin{bmatrix} ns_1 \\ cs \\ ns_2 \end{bmatrix} = N_c \begin{bmatrix} ns_2 \\ cs \\ ns_1 \end{bmatrix}$

If we precompose N_c with the swap relation ($\mathbf{sw}$), then this amounts to switching the name sets. The proof of this depends on the commutativity of $\parallel_{cs}$, a property that is proved in [35], and on Theorem 2.7 to switch the name set lenses. A corollary of Theorem 7.12 is a quasi-commutativity theorem for our parallel composition operator.

Theorem 7.13. *If* $ns_1 \bowtie ns_2$ *then* $P \llbracket ns_1 | cs | ns_2 \rrbracket Q = Q \llbracket ns_2 | cs | ns_1 \rrbracket P$

Thus, we can commute a parallel composition by also commuting the respective name sets.

[8] Specifically, the definition of $\parallel_{cs}$ may be found in Oliveira's thesis [36], Appendix B on page 183. It is based on the trace merge operator defined by Roscoe in Section 2.4, page 70, of [38].

In order to calculate a stateful-failure reactive design for parallel composition, we specialise Theorem 7.6. This requires that we have specialised versions of the merge operators for peri- and postconditions, and also a specialised weakest rely condition operator. These are defined below.

Definition 7.14 (Intermediate Merge, Final Merge, and Weakest Rely Condition)**.**

$$\begin{aligned} P \,[\![cs]\!]_{\mathsf{I}}\, Q &\triangleq P \parallel_{(\exists\, st' \bullet N_{\mathsf{I}})} Q \\ P \,[\![ns_1 | cs | ns_2]\!]_{\mathsf{F}}\, Q &\triangleq P \parallel_{(\exists\, ref' \bullet N_{\mathsf{C}})} Q \\ P \;\mathbf{wr}[cs]\; Q &\triangleq P \;\mathbf{wr}_{N_{\mathsf{I}}}\; Q \end{aligned} \qquad \text{where } N_{\mathsf{I}} \triangleq N_{\mathsf{C}} \begin{bmatrix} \mathbf{0} \\ cs \\ \mathbf{0} \end{bmatrix}$$

The intermediate merge, $[\![cs]\!]_{\mathsf{I}}$ defines how two quiescent observations are merged. It is parametrised only in cs and not ns_1 or ns_2, as state is concealed in quiescent observations. Its definition applies the merge predicate $(\exists\, st' \bullet N_{\mathsf{I}})$, which abstracts from the final state and is defined in terms of N_{I}. The latter applies N_{C}, but uses the $\mathbf{0}$ lens for ns_1 and ns_2, and therefore ignores the final state of P and Q.

The final state merge $[\![ns_1 | cs | ns_2]\!]_{\mathsf{F}}$ defines how terminated observations are merged. It is defined similarly, but abstracts from ref', since there is no refusal information in a final observation, and uses N_{C} directly to merge the states. Finally, the weakest rely condition $\mathbf{wr}[cs]$ is simply the general reactive design version, using N_{I} as the merge predicate as final states are also not relevant in preconditions.

We now demonstrate the healthiness of these new operators.

Theorem 7.15 (Merge Closure Properties)**.**

1. *If P and Q are* **CRR***-healthy then $P \,[\![cs]\!]_{\mathsf{I}}\, Q$ is* **CRR***-healthy.*
2. *$P \,[\![cs]\!]_{\mathsf{I}}\, Q$ does not refer to st'.*
3. *If P and Q are* **CRF***-healthy then $P \,[\![ns_1 | cs | ns_2]\!]_{\mathsf{F}}\, Q$ is* **CRF***-healthy.*
4. *If P is* **CRR***-healthy and Q is* **CRC***-healthy, then P $\mathbf{wr}[cs]$ Q is* **CRC***-healthy.*

The intermediate merge constructs a reactive relation that does not refer to the final state. The final merge constructs a reactive finaliser, since it does not refer to ref'. Weakest rely constructs a reactive condition. Following a similar approach to Theorem 7.13, we can also demonstrate commutativity properties.

Theorem 7.16 (Inner Merge Commutativity)**.**

$$\begin{aligned} P \,[\![cs]\!]_{\mathsf{I}}\, Q &= Q \,[\![cs]\!]_{\mathsf{I}}\, P \\ P \,[\![ns_1 | cs | ns_2]\!]_{\mathsf{F}}\, Q &= Q \,[\![ns_2 | cs | ns_1]\!]_{\mathsf{F}}\, P \end{aligned} \qquad ns_1 \bowtie ns_2$$

Using these new operators, we can finally prove the specialised calculation law for parallel composition.

Theorem 7.17 (Parallel Calculation)**.**

$$\begin{aligned} &[\, P_1 \vdash P_2 \mid P_3 \,] \,[\![ns_1 | cs | ns_2]\!]\, [\, Q_1 \vdash Q_2 \mid Q_3 \,] = \\ &\left[\begin{array}{l|l|l} \begin{array}{l} (P_1 \Rightarrow_r P_2)\ \mathbf{wr}[cs]\ Q_1 \wedge \\ (P_1 \Rightarrow_r P_3)\ \mathbf{wr}[cs]\ Q_1 \wedge \\ (Q_1 \Rightarrow_r Q_2)\ \mathbf{wr}[cs]\ P_1 \wedge \\ (Q_1 \Rightarrow_r Q_3)\ \mathbf{wr}[cs]\ P_1 \end{array} & \begin{array}{l} P_2 \,[\![cs]\!]_{\mathsf{I}}\, Q_2 \ \vee \\ P_3 \,[\![cs]\!]_{\mathsf{I}}\, Q_2 \ \vee \\ P_2 \,[\![cs]\!]_{\mathsf{I}}\, Q_3 \end{array} & P_3 \,[\![ns_1 | cs | ns_2]\!]_{\mathsf{F}}\, Q_3 \end{array}\right] \end{aligned}$$

This is similar to Theorem 7.6, but uses the specialised merge and weakest rely operators. This theorem shows that calculation of reactive contracts can be reduced to merging the peri- and postconditions. A corollary, for the simpler case when the preconditions are both $\mathbf{true}_r$, is given below.

Theorem 7.18 (Simplified Parallel Calculation)**.**

$$[\, \vdash P_2 \mid P_3 \,] \,[\![ns_1 | cs | ns_2]\!]\, [\, \vdash Q_2 \mid Q_3 \,] = [\, \vdash P_2 \,[\![cs]\!]_{\mathsf{I}}\, Q_2 \vee P_3 \,[\![cs]\!]_{\mathsf{I}}\, Q_2 \vee P_2 \,[\![cs]\!]_{\mathsf{I}}\, Q_3 \mid P_3 \,[\![ns_1 | cs | ns_2]\!]_{\mathsf{F}}\, Q_3 \,]$$

This follows by application of Theorem 7.7 because P $\mathbf{wr}[cs]$ $\mathbf{true}_r = \mathbf{true}_r$. We can also show, with the help of Theorems 4.3 and 7.15, that **NCSP** is closed under parallel composition.

Theorem 7.19. *If P and Q are **NCSP**-healthy, and $ns_1 \bowtie ns_2$, then $P \llbracket ns_1 | cs | ns_2 \rrbracket Q$ is **NCSP**-healthy.*

For sequential processes, we have already shown that the peri- and postconditions of reactive programs can be specified using disjunctions of the $\mathcal{E}$ and Φ operators. Consequently, to extend our calculational method to parallel composition, we need to prove how these operators should be merged. The following theorems show how reactive relations describing final and intermediate observations are merged.

Theorem 7.20 (Merging Finalisers)**.**

$$\Phi[s_1, \sigma_1, t_1] \ \llbracket ns_1 | cs | ns_2 \rrbracket_{\mathsf{F}} \ \Phi[s_2, \sigma_2, t_2] = (\exists\, t \bullet \Phi[s_1 \wedge s_2 \wedge t \in t_1 \parallel_{cs} t_2 \wedge t_1 \upharpoonright cs = t_2 \upharpoonright cs, \sigma_1 \, [ns_1 | ns_2] \, \sigma_2, t])$$

$$\Phi[s_1, \sigma_1, \langle\rangle] \ \llbracket ns_1 | cs | ns_2 \rrbracket_{\mathsf{F}} \ \Phi[s_2, \sigma_2, \langle\rangle] = \Phi[s_1 \wedge s_2, \sigma_1 \, [ns_1 | ns_2] \, \sigma_2, \langle\rangle]$$

The first equation shows how to merge two finalisers. We require the existence of the trace t, which is one of the possible merges of t_1 and t_2, and require that both preconditions s_1 and s_2 hold initially. The overall trace of the finaliser is then t. The second equation is a corollary for when both traces are empty, and the event merge is trivial. In either case, the final state update is constructed using the operator $[ns_1 | ns_2]$, which uses lens override to merge the two disjoint state updates. It obeys the following laws.

Theorem 7.21. *Given independent lenses, $ns_1 \bowtie ns_2$, the following identities hold:*

$$id \, [ns_1 | ns_2] \ id = id \tag{1}$$

$$\sigma \, [ns_1 | ns_2] \ \rho = \rho \, [ns_2 | ns_1] \ \sigma \tag{2}$$

$$(\sigma(x \mapsto v)) \, [ns_1 | ns_2] \ \rho = (\sigma \, [ns_1 | ns_2] \ \rho)(x \mapsto v) \qquad x \preceq ns_1 \tag{3}$$

$$(\sigma(x \mapsto v)) \, [ns_1 | ns_2] \ \rho = \sigma \, [ns_1 | ns_2] \ \rho \qquad x \bowtie ns_1 \tag{4}$$

Merging two identity (vacuous) assignments yields an identity assignment (1). The operator is quasi-commutative, when the name sets are also swapped (2). When one of the assignments is constructed with a state update, if the variable being assigned is part of the corresponding name set ($x \preceq ns_1$), then the update is applied to the top-level assignment (3). Effectively, this means that the assignment is retained when the parallel composition terminates. Conversely, if the assignment is to a variable outside of the name set ($x \bowtie ns_1$), then its effect is lost (4). Using these laws we calculate the contracts for some examples.

Example 7.22. We assume the existence of lenses x and y, with $x \preceq ns_1$, $y \preceq ns_2$, and $ns_1 \bowtie ns_2$, and calculate the meaning of parallel assignment to these variables.

$$\begin{aligned}
&(x := u) \ \llbracket ns_1 | cs | ns_2 \rrbracket (y := v) \\
&= [\, \vdash \mathbf{false} \mid \Phi[(\!| x \mapsto u |\!), \langle\rangle] \,] \ \llbracket ns_1 | cs | ns_2 \rrbracket \, [\, \vdash \mathbf{false} \mid \Phi[(\!| y \mapsto v |\!), \langle\rangle] \,] && [4.7] \\
&= \left[\, \left| \begin{array}{l} \mathbf{false} \ \llbracket cs \rrbracket_{\mathsf{I}} \ \mathbf{false} \\ \vdash \vee \ \Phi[(\!| x \mapsto u |\!), \langle\rangle] \ \llbracket cs \rrbracket_{\mathsf{I}} \ \mathbf{false} \\ \quad \vee \ \mathbf{false} \ \llbracket cs \rrbracket_{\mathsf{I}} \ \Phi[(\!| y \mapsto v |\!), \langle\rangle] \end{array} \right| \Phi[(\!| x \mapsto u |\!), \langle\rangle] \ \llbracket ns_1 | cs | ns_2 \rrbracket_{\mathsf{F}} \ \Phi[(\!| y \mapsto v |\!), \langle\rangle] \,\right] && [7.18] \\
&= [\, \vdash \mathbf{false} \mid \Phi[(\!| x \mapsto u |\!) \ [ns_1 | ns_2] \ (\!| y \mapsto v |\!), \langle\rangle] \,] && [7.2, 7.20] \\
&= [\, \vdash \mathbf{false} \mid \Phi[(\!| x \mapsto u, y \mapsto v |\!), \langle\rangle] \,] && [7.21] \\
&= \langle x \mapsto u, y \mapsto v \rangle_{\mathsf{C}} && [4.7]
\end{aligned}$$

This does not rule out having $u \triangleq y$ and $v \triangleq x$, since both processes have access to the entirety of the initial state. We first calculate the contract for the two assignments. Since the preconditions are trivial, we apply Theorem 7.18 to compute the composition contract. Since both periconditions are **false**, by application of Theorem 7.2 and relational calculus, the overall pericondition is also **false**. Thus, we can simply apply Theorem 7.20 to compute the merge of the two finalisers, and then Theorem 7.21 to merge the

two assignments. The final form is, by Definition 4.7, equivalent to a single assignment. In Isabelle/UTP, this proof of this equality is fully automated by the rdes-eq tactic [14].

Using a similar calculation, and using Definition 7.9, we can also show that

$$(x := u) \mid\mid\mid (y := v) = \textbf{\textit{Skip}}$$

The name sets are both $\mathbf{0}$ and consequently, since $x \bowtie \mathbf{0}$ and $y \bowtie \mathbf{0}$, both assignments are lost. □

The independence constraints on the process state spaces and loss of assignments may, at first sight, seem unsatisfactory as this prevents shared variables. However, variables here are only for the sequential case. Shared variables in languages like CSP and *Circus* should be modelled using channel communication, for separation of concerns. This approach has been demonstrated in several previous works [39], including JCSP [45] and the RoboChart state-machine language [33].

We next show how quiescent observations are merged using the $[\![cs]\!]_{\mathsf{I}}$.

Theorem 7.23 (Merging Quiescent Observations)**.**

$$\mathcal{E}[s_1, t_1, E_1] \; [\![cs]\!]_{\mathsf{I}} \; \mathcal{E}[s_2, t_2, E_2] = \left(\exists\, t \bullet \mathcal{E}\left[\begin{pmatrix} s_1 \wedge s_2 \wedge t \in t_1 \parallel_{cs} t_2 \wedge \\ t_1 \upharpoonright cs = t_2 \upharpoonright cs \end{pmatrix}, t, \begin{pmatrix} (E_1 \cap E_2 \cap cs) \cup \\ ((E_1 \cup E_2) \setminus cs) \end{pmatrix}\right]\right)$$

$$\mathcal{E}[s_1, t_1, E_1] \; [\![cs]\!]_{\mathsf{I}} \; \Phi[s_2, \sigma_2, t_2] = \left(\exists\, t \bullet \mathcal{E}\left[\begin{pmatrix} s_1 \wedge s_2 \wedge t \in t_1 \parallel_{cs} t_2 \wedge \\ t_1 \upharpoonright cs = t_2 \upharpoonright cs \end{pmatrix}, t, E_1 \setminus cs\right]\right)$$

The equations in Theorem 7.23 are similar to those in Theorem 7.20, but involve at least one quiescent observation. We omit the symmetric case, since we know that $[\![cs]\!]_{\mathsf{I}}$ is commutative. As for the finaliser, we need to merge the traces t_1 and t_2. There is no state update merge, as this is a quiescent observation. We also need to consider merging the refusal sets together, which here are given by a set of accepted events. If merging two quiescent observations, we accept (1) the events in the synchronisation set cs enabled by both P and Q ($E_1 \cap E_2 \cap cs$); and (2) the events not in cs that enabled available in either P or Q ($(E_1 \cup E_2) \setminus cs$). If a quiescent observation is merged with a final observation, then the accepted events are simply those not in cs – $E_1 \setminus cs$.

In order to illustrate the use of these theorems, we provide the following example.

Example 7.24. We calculate the meaning of $a \rightarrow b \rightarrow$ ***Skip*** $[\![\{b\}]\!]$ $b \rightarrow c \rightarrow$ ***Skip***.

$$= \left[\; \middle|\; \begin{matrix} \mathcal{E}[\langle\rangle, \{a\}] \vee \\ \mathcal{E}[\langle a\rangle, \{b\}] \end{matrix} \;\middle|\; \Phi[id, \langle a, b\rangle] \right] \left\| \{b\} \right\| \left[\; \middle|\; \begin{matrix} \mathcal{E}[\langle\rangle, \{b\}] \vee \\ \mathcal{E}[\langle b\rangle, \{c\}] \end{matrix} \;\middle|\; \Phi[id, \langle b, c\rangle] \right] \tag{1}$$

$$= \left[\; \middle|\; \begin{matrix} \mathcal{E}[\langle\rangle, \{a\}] \; [\![\{b\}]\!]_{\mathsf{I}} \; \mathcal{E}[\langle\rangle, \{b\}] \vee \\ \mathcal{E}[\langle\rangle, \{a\}] \; [\![\{b\}]\!]_{\mathsf{I}} \; \mathcal{E}[\langle b\rangle, \{c\}] \vee \\ \mathcal{E}[\langle\rangle, \{a\}] \; [\![\{b\}]\!]_{\mathsf{I}} \; \Phi[id, \langle b, c\rangle] \vee \\ \mathcal{E}[\langle a\rangle, \{b\}] \; [\![\{b\}]\!]_{\mathsf{I}} \; \mathcal{E}[\langle\rangle, \{b\}] \vee \\ \mathcal{E}[\langle a\rangle, \{b\}] \; [\![\{b\}]\!]_{\mathsf{I}} \; \mathcal{E}[\langle b\rangle, \{c\}] \vee \\ \cdots \end{matrix} \;\middle|\; \Phi[id, \langle a, b\rangle] \; [\![\mathbf{0} | \{b\} | \mathbf{0}]\!]_{\mathsf{F}} \; \Phi[id, \langle b, c\rangle] \right] \tag{2}$$

$$= \left[\; \middle|\; \begin{matrix} \left(\exists\, t \bullet \mathcal{E}\left[\begin{pmatrix} t \in \langle\rangle \parallel_{\{b\}} \langle\rangle \wedge \\ \langle\rangle \upharpoonright cs = \langle\rangle \upharpoonright cs \end{pmatrix}, t, \begin{matrix} \{a\} \cap \{b\} \cap \{b\} \cup \\ (\{a\} \cup \{b\}) \setminus \{b\} \end{matrix}\right]\right) \\ \vee \cdots \end{matrix} \;\middle|\; \exists\, t \bullet \Phi\left[\begin{pmatrix} t \in \langle a, b\rangle \parallel_{\{b\}} \langle b, c\rangle \wedge \\ \langle a, b\rangle \upharpoonright \{b\} = \langle b, c\rangle \upharpoonright \{b\} \end{pmatrix}, id, t\right] \right] \tag{3}$$

$$= [\; \vdash \mathcal{E}[\langle\rangle, \{a\}] \vee \mathcal{E}[\langle a\rangle, \{b\}] \vee \mathcal{E}[\langle a, b\rangle, \{c\}] \mid \Phi[id, \langle a, b, c\rangle]\;] \tag{4}$$

$$= a \rightarrow b \rightarrow c \rightarrow \textbf{\textit{Skip}} \tag{5}$$

Step (1) calculates the sequential contracts for the left- and right-hand sides of the parallel composition using the rules already outlined in §4. Step (2) expands out all the possible merges for the peri- and postcondition,

by application of Theorem 7.17, and also Theorem 7.2 to distribute through the various disjunctions. There are a total of nine observations (of which we show five) in the pericondition, because we need to merge every disjunct of the pericondition, plus the postcondition, with every corresponding disjunct. The majority of these are inadmissible and thus reduce to ***false***; for example

$$\mathcal{E}[\langle\rangle, \{a\}] \; [\![\{b\}]\!]_{\shortparallel} \; \mathcal{E}[\langle b\rangle, \{c\}] = \textbf{\textit{false}}$$

since the b event cannot occur independently. Step (3) uses Theorem 7.20 to demonstrate explicitly how to merge the first of the nine periconditions, and also the postcondition. For both the peri- and postcondition, we need to find a t that merges to two traces ($\langle\rangle$), and respects the synchronisation order. For the pericondition, there is only one such trace, $\langle\rangle$, and so this is the one selected. Moreover, it is necessary to calculate the events being accepted by appropriately selecting synchronised and non-synchronised events. For the postcondition, there is again only one trace, $\langle a, b, c\rangle$. Step (4) calculates all the admissible periconditions, of which there are three, and the postcondition. The three possible quiescent observations are (1) nothing has happened, and a is accepted; (2) a has occurred, and b is accepted; and (3) a and b have occurred, and c is accepted. The postcondition performs no state updates (id), and have the total sequence of events. This reactive contract is equivalent to the action $a \rightarrow b \rightarrow c \rightarrow$ ***Skip***, as shown in step (5).

Though this calculation seems very complicated, the benefit of our theorems and mechanisation is that it can be performed automatically in Isabelle/UTP. The rdes-eq tactic can also discover the contract form given in step (4) of the proof, though not the final form given in step (5). □

In the example given above, the preconditions are always trivial. For non-trivial preconditions, we need laws analogous to those for the sequential case in Theorem 4.11, but for the weakest rely calculus.

Theorem 7.25 (Parallel Preconditions)**.**

$$\Phi[s_1, \sigma_1, t_1] \; \textbf{\textit{wr}}[cs] \; \mathcal{I}[s_2, t_2] = (\forall\, tt_0, tt_1 \bullet \mathcal{I}[s_1 \wedge s_2 \wedge tt_1 \in (t_2 \frown tt_0) \parallel_{cs} t_1 \wedge (t_2 \frown tt_0) \upharpoonright cs = t_1 \upharpoonright cs, tt_1])$$
$$\mathcal{E}[s_1, t_1, E] \; \textbf{\textit{wr}}[cs] \; \mathcal{I}[s_2, t_2] = (\forall\, tt_0, tt_1 \bullet \mathcal{I}[s_1 \wedge s_2 \wedge tt_1 \in (t_2 \frown tt_0) \parallel_{cs} t_1 \wedge (t_2 \frown tt_0) \upharpoonright cs = t_1 \upharpoonright cs, tt_1])$$

As usual, we need to conjoin both conditions s_1 and s_2. However, determining permissible traces is rather more involved. We recall that in $\mathcal{I}[s_2, t_2]$, t_2 is a strict upper bound on the permissible traces. These two laws give the conditions under which a concurrent quiescent or final observation does not allow divergence; both have the same form. Divergence occurs when t_1, a trace contributed by one action, permits t_2, a divergent trace, to be exhausted when the two are merged. This situation occurs for any trace tt_1 such that there is an arbitrary extension tt_0, where (1) tt_1 is one of the traces obtained by merging t_2 extended with tt_0, with t_1, and (2) the order of synchronisation of t_2 with its extension is the same as that of t_1. The trace tt_1 must therefore be a trace that exhausts all the events in t_2, whilst respecting both the synchronisation set and t_1. Consequently, it is a strict upper bound on the permissible behaviours.

We exemplify these laws with the calculation below.

Example 7.26.

$$
\begin{array}{ll}
& a \to \textbf{\textit{Chaos}} \,[\![\{a\}]\!]\, a \to \textbf{\textit{Skip}} \\
= & [\, \mathcal{I}[\mathit{true}, \langle a \rangle] \vdash \mathcal{E}[\mathit{true}, \langle\rangle, \{a\}] \mid \textbf{\textit{false}} \,]\, [\![\{a\}]\!]\, [\, \vdash \mathcal{E}[\langle\rangle, \{a\}] \mid \Phi[id, \langle a \rangle] \,] & [4.7] \\
= & \left[\begin{array}{l} \mathcal{E}[\langle\rangle, \{a\}] \ \textbf{\textit{wr}}[\{a\}]\ \mathcal{I}[\mathit{true}, \langle a \rangle] \\ \wedge\ \Phi[id, \langle a \rangle] \ \textbf{\textit{wr}}[\{a\}]\ \mathcal{I}[\mathit{true}, \langle a \rangle] \end{array} \middle| \begin{array}{l} \mathcal{E}[\langle\rangle, \{a\}] \ [\![\{a\}]\!]_{\shortparallel}\ \mathcal{E}[\langle\rangle, \{a\}] \\ \vee\ \mathcal{E}[\mathit{true}, \langle\rangle, \{a\}] \ [\![\{a\}]\!]_{\shortparallel}\ \Phi[id, \langle a \rangle] \end{array} \middle| \textbf{\textit{false}} \right] & [7.18] \\
= & \left[\begin{array}{l} \mathcal{E}[\langle\rangle, \{a\}] \ \textbf{\textit{wr}}[\{a\}]\ \mathcal{I}[\mathit{true}, \langle a \rangle] \\ \wedge\ \Phi[id, \langle a \rangle] \ \textbf{\textit{wr}}[\{a\}]\ \mathcal{I}[\mathit{true}, \langle a \rangle] \end{array} \middle| \begin{array}{l} \mathcal{E}[\langle\rangle, \{a\}] \\ \vee\ \textbf{\textit{false}} \end{array} \middle| \textbf{\textit{false}} \right] & [7.23] \\
= & \left[\begin{array}{l} \Big(\forall (tt_0, tt_1) \bullet \mathcal{I}\Big[tt_1 \in (\langle a \rangle \frown tt_0) \parallel_{\{a\}} \langle\rangle \wedge (\langle a \rangle \frown tt_0) \upharpoonright \{a\} = \langle\rangle \upharpoonright cs, tt_1\Big]\Big) \wedge \\ \Big(\forall (tt_0, tt_1) \bullet \mathcal{I}\Big[tt_1 \in (\langle a \rangle \frown tt_0) \parallel_{\{a\}} \langle a \rangle \wedge (\langle a \rangle \frown tt_0) \upharpoonright \{a\} = \langle a \rangle \upharpoonright \{a\}, tt_1\Big]\Big) \end{array} \middle| \mathcal{E}[\langle\rangle, \{a\}] \middle| \textbf{\textit{false}} \right] & [7.25] \\
= & \left[\begin{array}{l} \textbf{\textit{true}}_r \wedge \\ (\forall\, tt_1 \bullet \mathcal{I}\Big[tt_1 \in (\langle a \rangle \frown \langle\rangle) \parallel_{\{a\}} \langle a \rangle \wedge (\langle a \rangle \frown \langle\rangle) \upharpoonright \{a\} = \langle a \rangle \upharpoonright \{a\}, tt_1\Big]) \end{array} \middle| \mathcal{E}[\langle\rangle, \{a\}] \middle| \textbf{\textit{false}} \right] & \\
= & \left[\ (\forall\, tt_1 \bullet \mathcal{I}\Big[tt_1 \in \langle a \rangle \parallel_{\{a\}} \langle a \rangle, tt_1\Big]) \ \middle| \ \mathcal{E}[\langle\rangle, \{a\}] \ \middle| \ \textbf{\textit{false}} \right] & \\
= & [\ (\forall\, tt_1 \bullet \mathcal{I}[tt_1 = \langle a \rangle, tt_1]) \ \mid \mathcal{E}[\langle\rangle, \{a\}] \mid \textbf{\textit{false}} \,] & \\
= & [\ \mathcal{I}[\mathit{true}, \langle a \rangle] \ \mid \mathcal{E}[\langle\rangle, \{a\}] \mid \textbf{\textit{false}} \,] & \\
= & a \to \textbf{\textit{Chaos}} & [4.7]
\end{array}
$$

We calculate the contract for the parallel composition as usual, but in this case it is necessary to calculate two weakest rely formulae: (1) $\mathcal{E}[\langle\rangle, \{a\}]\ \textbf{\textit{wr}}[\{a\}]\ \mathcal{I}[\mathit{true}, \langle a \rangle]$, and (2) $\Phi[id, \langle a \rangle]\ \textbf{\textit{wr}}[\{a\}]\ \mathcal{I}[\mathit{true}, \langle a \rangle]$. We expand them both out using Theorem 7.25. For (1), we observe that the resulting formula has the equation $(\langle a \rangle \frown tt_0) \upharpoonright \{a\} = \langle\rangle \upharpoonright \{a\}$. This is impossible to satisfy, since the left-hand trace contains a, but the right-hand side does not. Consequently, this term, and therefore the whole condition, reduces to *false*, and so the resulting formula is $(\forall (tt_0, tt_1) \bullet \mathcal{I}[\mathit{false}, tt_1])$, which, by Theorem 4.11, is simply $\textbf{\textit{true}}_r$. The intuition here is that the empty trace does not permit violation of the corresponding precondition. For (2), we notice that there is exactly one possible valuation of tt_0 that satisfies the resulting formula, which is $\langle\rangle$. In this case, the precondition can be violated, and tt_1 also has one possible value, which is $\langle a \rangle$. The resulting formula is simply $\mathcal{I}[\mathit{true}, \langle a \rangle]$, and the overall behaviour is equivalent to $a \to \textbf{\textit{Chaos}}$. □

7.5. Algebraic Properties

We now explore the algebraic properties of parallel composition. We show that, under certain conditions characterised by healthiness conditions, ***Skip*** is a unit for parallel composition, and ***Chaos*** is an annihilator.

In previous work [36], support for this law is provided by an additional healthiness condition, which imposes downward closure of the refusals. This property is imposed in the standard CSP model [38] by healthiness condition ***F2***. However, not all expressible healthy reactive relations satisfy this property. For example the relation $\mathit{ref}' = \{a, b\}$, which is ***CRC***-healthy, identifies a single refusal and thus forbids the refusal sets $\{a\}$, $\{b\}$, and $\emptyset$, which we would normally expect to be admissible due to subset closure. We therefore define the following additional healthiness condition for quiescent observations.

Definition 7.27 (Refusal Downward Closure)**.** A reactive relation is downward closed with respect to refusals if it is a fixed-point of healthiness condition ***CDC***, defined below.

$$\textbf{\textit{CDC}}(P) \triangleq (\exists\, \mathit{ref}_0 \bullet P[\mathit{ref}_0/\mathit{ref}'] \wedge \mathit{ref}' \subseteq \mathit{ref}_0)$$

A reactive relation P has downward closed refusals if, when we replace ref' with an arbitrary subset, we obtain an observation that is still within P. It is easy to prove that ***CDC*** is idempotent, which follows due to transitivity of $\subseteq$, and also monotonic. We can also show that ***CDC*** is closed under existing operators.

Theorem 7.28 (***CDC*** Closure Properties)**.**

- ***CDC*** *is closed under the following operators:* $\mathbf{true}_r$, ***false***, $\vee$, *and* $\wedge$;
- *If* Q *is* ***CDC****-healthy, then* $(P\,;\,Q)$ *is* ***CDC****-healthy;*
- *If* $\forall\, i \in I \bullet P(i)$ *is* ***CDC*** *then* $\bigwedge_{i\in I} P(i)$ *is* ***CDC*** *and* $\bigvee_{i\in I} P(i)$ *is* ***CDC****;*
- *For any* s, t, *and* E, $\mathcal{E}[s,t,E]$ *is* ***CDC****-healthy.*

$\mathcal{E}[s,t,E]$ is ***CDC***-healthy because, as seen in Definition 4.6, we construct the set of refusals which do not include any event in E, a formulation that is downward closed. Consequently, we know that all the forms of pericondition considered so far, which are disjunctions of $\mathcal{E}[s,t,E]$ terms, are ***CDC***-healthy. Next, we recast Oliveira's healthiness condition for downward closure, called ***C2*** [35], to our setting.

Definition 7.29. $\mathbf{C2}(P) \triangleq P \,[\![\mathbf{1}|\emptyset|\mathbf{0}]\!]\, \mathbf{Skip}$

C2 states that ***Skip***, defined in Definition 4.7, is a right unit for the composition operator $[\![\mathbf{1}|\emptyset|\mathbf{0}]\!]$, which takes the entirety of its final state from the left action, and employs an empty synchronisation set. We now link ***C2*** to ***CDC***. The proof depends on two properties of final state merge and weakest rely predicates.

Theorem 7.30 (Merge and Weakest Rely of Identity)**.**

$$P \,[\![\mathbf{1}|\emptyset|\mathbf{0}]\!]_{\mathsf{F}}\, \Phi[id, \langle\rangle] = P \qquad \textit{if } P \textit{ is } \mathbf{CRF}\textit{-healthy}$$
$$\Phi[id, \langle\rangle]\ \mathbf{wr}[\emptyset]\ P = P \qquad \textit{if } P \textit{ is } \mathbf{CRC}\textit{-healthy}$$

The first property states that merging an arbitrary finaliser P with an identity finaliser, with P contributing all the final state, and an empty synchronisation set, is simply P. The second property, similarly, states that the weakest rely condition that an identity finaliser reaches reactive condition P, with an empty synchronisation set, is simply P. We can now prove the following important theorem for ***C2***, employing our calculational strategy, which reveals its intuitive meaning.

Theorem 7.31. *If* $[\,P_1 \vdash P_2 \mid P_3\,]$ *is* ***NCSP*** *then* $\mathbf{C2}([\,P_1 \vdash P_2 \mid P_3\,]) = [\,P_1 \vdash \mathbf{CDC}(P_2) \mid P_3\,]$

Proof.

$$\begin{aligned}
\mathbf{C2}([\,P_1 \vdash P_2 \mid P_3\,]) &= [\,P_1 \vdash P_2 \mid P_3\,]\,[\![\mathbf{1}|cs|\mathbf{0}]\!]\,[\,\vdash \mathbf{false} \mid \Phi[id, \langle\rangle]\,] && [7.29, 4.7]\\
&= [\,\Phi[id, \langle\rangle]\ \mathbf{wr}[\emptyset]\ P_1 \vdash P_2 \,[\![cs]\!]_{\mathsf{I}}\, \Phi[id, \langle\rangle] \mid P_3 \,[\![\mathbf{1}|\emptyset|\mathbf{0}]\!]_{\mathsf{F}}\, \Phi[id, \langle\rangle]\,] && [7.17]\\
&= [\,P_1 \vdash P_2 \,[\![cs]\!]_{\mathsf{I}}\, \Phi[id, \langle\rangle] \mid P_3\,] && [7.30]\\
&= \left[\, P_1 \vdash (\exists\, ref_0 \bullet P_2[ref_0/ref'] \wedge ref' \subseteq ref_0) \;\middle|\; P_3\,\right]\\
&= [\,P_1 \vdash \mathbf{CDC}(P_2) \mid P_3\,] && \square
\end{aligned}$$

This theorem tells us that an ***NCSP***-healthy reactive contract (see Theorem 4.3) is ***C2*** when its pericondition is ***CDC***. The proof calculates contract for the parallel composition with ***Skip***, and then shows that both the precondition and postcondition are unaltered, using Theorem 7.30. Finally, we show that the pericondition formula $P_2 \,[\![cs]\!]_{\mathsf{I}}\, \Phi[id, \langle\rangle]$ is equivalent to $\mathbf{CDC}(P)$, by application of relational calculus. From this theorem, and previous definitions, we can now prove the following closure theorems for ***C2***.

Theorem 7.32 (***C2*** closure properties)**.**

- ***Miracle***, ***Chaos***, ***Skip***, ***Stop***, $\mathbf{Do}(a)$, *and* $\langle\sigma\rangle_{\mathsf{c}}$ *are all* ***C2****;*
- *If* P *and* Q *are both* ***NCSP*** *and* ***C2****, then* $P\,;\,Q$, $P \lhd b \rhd Q$, *and* $P \,\Box\, Q$ *are all* ***C2****;*
- *If* $\forall\, i \in I \bullet P(i)$ *is* ***C2*** *then* $\bigsqcap_{i\in I} P(i)$ *is* ***C2****;*
- *If* P *is* ***PCSP*** *and* ***C2*** *then* $b \circledast P$ *is* ***C2****;*
- *If* $ns_1 \bowtie ns_2$, *and* P *and* Q *are both* ***NCSP*** *and* ***C2****, then* $P \,[\![ns_1|cs|ns_2]\!]\, Q$ *is* ***C2****.*

We can now prove two algebraic theorems for **C2** reactive programs.

Theorem 7.33. *If P is* **NCSP** *and* **C2** *then* $P \llbracket \mathbf{1} | cs | \mathbf{0} \rrbracket \textbf{Skip} = P$

Theorem 7.34. *If P is* **NCSP** *and* **C2**, *and* $\Sigma = \{\emptyset\}$, *then* $P \lll \textbf{Skip} = P$

Theorem 7.33 is essentially a restatement of **C2**, that is, **Skip** is a right-unit when P controls the entire state-space. However, we can now use Theorem 7.32 to satisfy its provisos, and thus apply it to programs whose pericondition is **CDC**. Theorem 7.34 is similar, but has the additional proviso that the state space Σ is unitary, and the state therefore contains no information. This being the case, P is a process [35], to use *Circus* terminology, rather than an action, since it has no visible state updates.

Next, we consider annihilators for parallel composition. We calculate the meaning of **Chaos** $\llbracket ns_1 | cs | ns_2 \rrbracket P$, for **NCSP** healthy reactive contract P, using our proof strategy:

Example 7.35. **Chaos** parallel composition

$$\begin{aligned} \textbf{Chaos} \llbracket ns_1 | cs | ns_2 \rrbracket P &= [\, \textbf{false} \vdash \textbf{false} \mid \textbf{false} \,] \llbracket ns_1 | cs | ns_2 \rrbracket [\, P_1 \vdash P_2 \mid P_3 \,] \\ &= [\, P_2 \ \textbf{wr}[cs] \ \textbf{false} \wedge P_3 \ \textbf{wr}[cs] \ \textbf{false} \wedge \textbf{true}_r \ \textbf{wr}[cs] \ P_1 \vdash \textbf{false} \mid \textbf{false} \,] \end{aligned}$$

Due to the definition of **Chaos**, by Theorem 7.17 the peri- and postcondition reduce to **false**. Consequently, to show that **Chaos** is an annihilator, it is necessary simply to show that the precondition reduces to **false** in order to complete the reduction to **Chaos**. We already know by Theorem 7.8 that at least **Miracle** does not satisfy this requirement, since it is itself an annihilator for any reactive design, including **Chaos**. Consequently, we need to consider constraints under which one of the precondition conjuncts reduce to **false**.

The third conjunct, $\textbf{true}_r \ \textbf{wr}[cs] \ P_1$, in general reduces to **false** only when P_1 is itself **false**, and therefore $P = \textbf{Chaos}$, which is a trivial and therefore uninteresting case. The first two cases are more interesting, and correspond to the presence of feasible behaviour by either the peri- or the postcondition. Here, we investigate the circumstances under which $P_2 \ \textbf{wr}[cs] \ \textbf{false} = \textbf{false}$. For this, we need an additional healthiness condition that ensures that there is at least one observation in the pericondition.

Definition 7.36 (Accepting Actions)**.**

$$\begin{aligned} \textbf{Accept} &\triangleq [\, \textbf{true}_r \vdash \mathcal{E}[\langle\rangle, \mathit{Event}] \mid \textbf{false} \,] \\ \textbf{CACC}(P) &\triangleq (P \vee \textbf{Accept}) \end{aligned}$$

Accept is the action that does not terminate, but has a single quiescent observation where nothing has occurred ($\langle\rangle$), and every event in *Event* is accepted. It has a similar form to **Stop**, except that the latter accepts no events. **Accept** accepts every event, and yet no event can ever be added to the trace. Like **Miracle**, it is excluded by several of the standard CSP healthiness conditions [38]; in particular it violates the requirement that every enabled event must also appear in the trace. However, like **Miracle**, it also possesses interesting theoretical properties. We emphasise that $\mathcal{E}[\langle\rangle, \mathit{Event}]$ is both **CRR** and **CDC** healthy.

The healthiness condition **CACC** takes the disjunction of P with **Accept**, which effectively states that P is refined by **Accept**, and thus sets an upper bound on P [28]. Using Theorem 2.15, we prove the following calculation for application of **CACC** to a reactive contract:

Theorem 7.37. $\textbf{CACC}([\, P_1 \vdash P_2 \mid P_3 \,]) = [\, P_1 \vdash \mathcal{E}[\langle\rangle, \mathit{Event}] \vee P_2 \mid P_3 \,]$

CACC thus requires that the pericondition refines $\mathcal{E}[\langle\rangle, \mathit{Event}]$. This means that the pericondition must admit an observation where nothing has occurred ($\langle\rangle$), and also that any subset of *Event* is accepted (including $\emptyset$). This intuition is demonstrated by the following useful theorem.

Theorem 7.38. $\mathcal{E}[s_1, t, E_1] \sqsubseteq \mathcal{E}[s_2, t, E_2] \Leftrightarrow (s_1 \Rightarrow s_2 \wedge E_1 \subseteq E_2)$

Refinement of one quiescent observation by another, sharing the same trace, requires that the state condition is strengthened, and that set of enabled events becomes larger. This may seem counter-intuitive, but it is because we encode refusals in ref', and therefore the most constrained refusal observation is $\mathit{ref}' = \emptyset$, which corresponds to every event being enabled. The majority of productive operators presented so far satisfy this constraint, and therefore we can prove the following closure properties for **CACC**.

Theorem 7.39 (***CACC*** Closure)**.** *Let P and Q be **NCSP**-healthy relations, then:*

- ***Chaos***, ***Stop***, *and* ***Do***(e) *are* ***CACC***-*healthy;*
- *If P is* ***CACC*** *then $P \,;\, Q$ is* ***CACC****;*
- *If P and Q are both* ***CACC*** *then $P \Box Q$ is* ***CACC****;*
- *If P and Q are both* ***CACC*** *then $P \sqcap Q$ is* ***CACC****.*

Miracle is not ***CACC*** because its pericondition is ***false***, and therefore does not have an empty interaction. More importantly, however, $\langle\sigma\rangle_c$ and ***Skip*** are also not ***CACC***, because they too have a ***false*** pericondition. However, for most processes that0 include at least one interaction, and do not invoke infeasible actions like ***Miracle***, ***CACC*** is satisfied. In particular, we note that closure of ***CACC*** under sequential composition only requires that the first argument is ***CACC***. Therefore, since by Theorems 4.10 and 5.7 we can usually push leading assignments forward, most actions are ***CACC***-healthy. Alternatively, we could define a pseudo unit, ***NoOp*** $\triangleq [\, \mathit{true}_r \vdash \mathcal{E}[\langle\rangle, \mathit{Event}] \mid \Phi[\mathit{id}, \langle\rangle]\,]$, but this has the undesirable characteristic of not refusing any event whilst also not engaging in any event, which violates the standard CSP healthiness conditions [38].

With this healthiness condition, we can finally prove the following theorem:

Theorem 7.40. *If $ns_1 \bowtie ns_2$, and P is* ***NCSP*** *and* ***CACC****, then* ***Chaos*** $[\![ns_1|cs|ns_2]\!]\ P =$ ***Chaos****.*

Proof. Given that $P = [\, P_1 \vdash P_2 \mid P_3\,]$, and noting the calculation in Example 7.35, it suffices to show that P_2 ***wr***$[cs]$ ***false*** reduces to ***false***.

$$
\begin{aligned}
P_2\ \textbf{\textit{wr}}[cs]\ \textbf{\textit{false}} &= (\mathcal{E}[\langle\rangle, \mathit{Event}] \vee P_2)\ \textbf{\textit{wr}}[cs]\ \textbf{\textit{false}} && [7.37]\\
&= (\mathcal{E}[\langle\rangle, \mathit{Event}]\ \textbf{\textit{wr}}[cs]\ \textbf{\textit{false}}) \wedge (P_2\ \textbf{\textit{wr}}[cs]\ \textbf{\textit{false}})\\
&= (\mathcal{E}[\langle\rangle, \mathit{Event}]\ \textbf{\textit{wr}}[cs]\ \mathcal{I}[\mathit{true}, \langle\rangle]) \wedge (P_2\ \textbf{\textit{wr}}[cs]\ \textbf{\textit{false}})\\
&= \left(\forall(tt_0, tt_1) \bullet \mathcal{I}\left[\begin{pmatrix} tt_1 \in (\langle\rangle \frown tt_0) \parallel_{cs} \langle\rangle \wedge \\ (\langle\rangle \frown tt_0) \restriction cs = \langle\rangle \restriction cs \end{pmatrix}, tt_1\right]\right) \wedge (P_2\ \textbf{\textit{wr}}[cs]\ \textbf{\textit{false}})\\
&= \left(\forall(tt_0, tt_1) \bullet \mathcal{I}\left[\begin{pmatrix} tt_1 \in tt_0 \parallel_{cs} \langle\rangle \\ \wedge\ tt_0 \restriction cs = \langle\rangle \end{pmatrix}, tt_1\right]\right) \wedge (P_2\ \textbf{\textit{wr}}[cs]\ \textbf{\textit{false}})\\
&= \begin{array}{l} \mathcal{I}[\langle\rangle \in \langle\rangle \parallel_{cs} \langle\rangle \wedge \langle\rangle \restriction cs = \langle\rangle, \langle\rangle] \\ \quad \wedge \left(\forall(tt_0, tt_1) \bullet \mathcal{I}\left[\begin{pmatrix} tt_1 \in tt_0 \parallel_{cs} \langle\rangle \wedge tt_0 \restriction cs = \langle\rangle \end{pmatrix}, tt_1\right]\right) \wedge (P_2\ \textbf{\textit{wr}}[cs]\ \textbf{\textit{false}}) \end{array}\\
&= \mathcal{I}[\mathit{true}, \langle\rangle] \wedge (\forall(tt_0, tt_1) \bullet \mathcal{I}[\cdots, tt_1]) \wedge (P_2\ \textbf{\textit{wr}}[cs]\ \textbf{\textit{false}})\\
&= \textbf{\textit{false}} \qquad \Box
\end{aligned}
$$

The crucial part of the proof is that the complex $\mathcal{I}$ formula must hold for any given tt_0 and tt_1, and so we can pick $\langle\rangle$ for both of them, and add this as an extra conjunct. Since $\langle\rangle$ merged with $\langle\rangle$ yields $\{\langle\rangle\}$, the resulting formula reduces to $\mathcal{E}[\mathit{true}, \langle\rangle]$, which is simply ***false***. This calculation would not be possible if we could not exhibit $\langle\rangle$ as a possible trace in the pericondition, which is the purpose of ***CACC***.

In this section, we have shown how the calculational strategy can be extended to handle parallel composition, and proved proved some important theorems that follow. In the next section we demonstrate the proof strategy on a small example.

8. Verification Strategy for Reactive Programs

Our results give rise to an automated verification strategy for reactive programs, whereby we (1) calculate the contract of a reactive program, (2) use our equational theory to simplify the underlying reactive relations, (3) identify invariants for reactive while loops, and (4) finally prove refinements using relational calculus. Although the relations can be complex, our equational theory from §4 and §5, aided by the Isabelle/HOL simplifier, can be used to rapidly reduce them to more compact forms amenable to automated proof. In this

section we illustrate this strategy using the buffer in Example 2.9. We prove two properties: (1) deadlock freedom, and (2) the order of values produced is the same as those consumed.

Deadlock freedom can be demonstrated with the help of the following specification contract [14].

Definition 8.1 (Deadlock-freedom Contract). $\textbf{\textit{CDF}} \triangleq [\vdash \exists\, s, t, E, e \bullet \mathcal{E}[s, t, \{e\} \cup E] \mid \textbf{\textit{true}}_r]$

Since only quiescent observations can deadlock, ***CDF*** constrains only the pericondition, which characterises observations where at least one event e is being accepted: there is no deadlock. For example, we can show that $a \to b \to \textbf{\textit{Skip}} \,[\![\{b\}]\!]\, b \to c \to \textbf{\textit{Skip}}$ is deadlock-free with the help of Example 7.24.

Example 8.2 (Deadlock-Freedom Calculation).

$$
\begin{aligned}
&\textbf{\textit{CDF}} \sqsubseteq a \to b \to \textbf{\textit{Skip}} \,[\![\{b\}]\!]\, b \to c \to \textbf{\textit{Skip}} \\
\Leftrightarrow\ & [\vdash \exists\, s, t, E, e \bullet \mathcal{E}[s, t, \{e\} \cup E] \mid \textbf{\textit{true}}_r] \sqsubseteq [\vdash \mathcal{E}[\langle\rangle, \{a\}] \lor \mathcal{E}[\langle a\rangle, \{b\}] \lor \mathcal{E}[\langle a, b\rangle, \{c\}] \mid \Phi[id, \langle a, b, c\rangle]] \\
\Leftrightarrow\ & (\exists\, s, t, E, e \bullet \mathcal{E}[s, t, \{e\} \cup E]) \sqsubseteq (\mathcal{E}[\langle\rangle, \{a\}] \lor \mathcal{E}[\langle a\rangle, \{b\}] \lor \mathcal{E}[\langle a, b\rangle, \{c\}]) \land \textbf{\textit{true}}_r \sqsubseteq \Phi[id, \langle a, b, c\rangle] \\
\Leftrightarrow\ & (\exists\, s, t, E, e \bullet \mathcal{E}[s, t, \{e\} \cup E]) \sqsubseteq \mathcal{E}[\langle\rangle, \{a\}] \land (\exists\, s, t, E, e \bullet \mathcal{E}[s, t, \{e\} \cup E]) \sqsubseteq \mathcal{E}[\langle a\rangle, \{b\}] \land \cdots \\
\Leftrightarrow\ & \textbf{\textit{true}}
\end{aligned}
$$

The intuition is that every $\mathcal{E}[\cdot, \cdot, \cdot]$ term in the process's pericondition corresponds to a possible transition. Consequently, we need to show that no transition exists without an enabled event. This is the case for all three disjuncts — they enable $\{a\}$, $\{b\}$ and $\{c\}$, respectively — and so the process is deadlock-free. □

To prove that the buffer is deadlock-free, we first calculate the contract of the main loop in the *Buffer* process in Example 2.9, and then use this to calculate the overall contract for the iterative behaviour.

Theorem 8.3 (Loop Body). *The body of the loop is* $[\textbf{\textit{true}}_r \vdash B_2 \mid B_3]$ *where*

$$
B_2 = \mathcal{E}\big[true, \langle\rangle, \textstyle\bigcup_{v \in \mathbb{N}} \{inp.v\} \cup (\{out.head(bf)\} \lhd 0 < \#bf \rhd \emptyset)\big]
$$

$$
B_3 = \begin{pmatrix} (\bigvee_{v \in \mathbb{N}} \Phi[true, \{bf \mapsto bf \frown \langle v\rangle\}, \langle inp.v\rangle]) \lor \\ \Phi[0 < \#bf, \{bf \mapsto tail(bf)\}, \langle out.head(bf)\rangle] \end{pmatrix}
$$

The $\textbf{\textit{true}}_r$ precondition implies no divergence. The pericondition states that every input event is enabled, and the output event is enabled if the buffer is non-empty. The postcondition contains two possible final observations: (1) an input event occurred and the buffer variable was extended; or (2) provided the buffer was non-empty initially, then the buffer's head is output and bf is contracted.

Proof. To exemplify, we calculate the left-hand side of the choice, employing Theorems 2.15, 4.7, 4.8, and 5.2. The entire calculation is automated in Isabelle/UTP.

$$
\begin{aligned}
& inp?v \to bf := bf \frown \langle v\rangle \\
&= \Box\, v \in \mathbb{N} \bullet \textbf{\textit{Do}}(inp.v)\ ;\ bf := bf \frown \langle v\rangle && \text{[Defs]} \\
&= \Box\, v \in \mathbb{N} \bullet \begin{pmatrix} [\textbf{\textit{true}}_r \vdash \mathcal{E}[true, \langle\rangle, \{inp.v\}] \mid \Phi[true, id, \langle inp.v\rangle]]\ ; \\ [\textbf{\textit{true}}_r \vdash \textbf{\textit{false}} \mid \Phi[true, (\!|bf \mapsto bf \frown \langle v\rangle|\!), \langle\rangle]] \end{pmatrix} && [4.7] \\
&= \Box\, v \in \mathbb{N} \bullet \left[\textbf{\textit{true}}_r \vdash \begin{array}{l|l} \mathcal{E}[true, \langle\rangle, \{inp.v\}] & \Phi[true, id, \langle inp.v\rangle]\ ; \\ \lor\ \textbf{\textit{false}} & \Phi[true, (\!|bf \mapsto bf \frown \langle v\rangle|\!), \langle\rangle] \end{array}\right] && [2.15, 4.11] \\
&= \Box\, v \in \mathbb{N} \bullet [\textbf{\textit{true}}_r \vdash \mathcal{E}[true, \langle\rangle, \{inp.v\}] \mid \Phi[true, (\!|bf \mapsto bf \frown \langle v\rangle|\!), \langle inp.v\rangle]] && [4.8] \\
&= \left[\textbf{\textit{true}}_r \vdash \mathcal{E}\left[true, \langle\rangle, \bigcup_{v \in \mathbb{N}} \{inp.v\}\right] \,\middle|\, \bigvee_{v \in \mathbb{N}} \Phi[true, (\!|bf \mapsto bf \frown \langle v\rangle|\!), \langle inp.v\rangle]\right] && [5.1, 5.2]
\end{aligned}
$$

Though this calculation seems complicated, in practice it is fully automated and a user need not be concerned with these minute calculational details, but can rather focus on finding suitable reactive invariants. □

Then, by Theorem 6.3 we can calculate the overall behaviour of the buffer.

$$Buffer = [\, \mathbf{true}_r \vdash \Phi[true, \{bf \mapsto \langle\rangle\}, \langle\rangle] \,;\, B_3^* \,;\, B_2 \mid \mathbf{false} \,]$$

This is a non-terminating contract where every quiescent behaviour begins with an empty buffer, performs some sequence of buffer inputs and outputs accompanied by state updates (B_3^*), and is finally offering the relevant input and output events (B_2). We can now employ Theorem 6.4 to verify the buffer. First, we tackle deadlock freedom, which can be proved using the following refinement.

Theorem 8.4 (Deadlock Freedom). $\mathbf{CDF} \sqsubseteq Buffer$

This theorem can be discharged automatically in 1.8s on an Intel i7-4790 desktop machine. This proof approach has also been applied in demonstrating that formalised state machine models, in the RoboChart language [33], are deadlock-free [12] with a similar level of automation. We next tackle the second property.

Theorem 8.5 (Buffer Order Property). *The sequence of items output is a prefix of those that were previously input. This can be formally expressed as*

$$[\, \mathbf{true}_r \vdash outps(tt) \leq inps(tt) \mid \mathbf{true}_r \,] \sqsubseteq Buffer$$

where $inps(t), outps(t) : \mathrm{seq}\ \mathbb{N}$ *extract the sequence of input and output elements from the trace* t*, respectively. The postcondition is left unconstrained as Buffer does not terminate.*

Proof. First, we identify the reactive invariant $I \triangleq outps(tt) \leq bf \frown inps(tt)$, and show that $[\mathbf{true}_r \vdash I \mid \mathbf{true}_r] \sqsubseteq \mathbf{true}_r \circledast [\mathbf{true}_r \vdash B_2 \mid B_3]$. By Theorem 6.4 it suffices to show case (2), that is $I \sqsubseteq B_2$ and $I \sqsubseteq B_3 \,;\, I$, as the other two cases are vacuous. These two properties can be discharged by relational calculus. Second, we prove that $[\mathbf{true}_r \vdash outps(tt) \leq inps(tt) \mid \mathbf{true}_r] \sqsubseteq bf := \langle\rangle \,;\, [\mathbf{true}_r \vdash I \mid \mathbf{true}_r]$. This holds, by Theorem 4.10-(1), since $I[\langle\rangle / bf] = outps(tt) \leq inps(tt)$. Thus, the overall theorem holds by monotonicity of ; and transitivity of $\sqsubseteq$. The proof is semi-automatic — since we have to manually apply induction with Theorem 6.4 — with the individual proof steps taking 2.2s in total.

□

9. Conclusion

We have demonstrated an effective verification strategy for concurrent and reactive programs employing reactive relations and Kleene algebra. We have provided three novel operators for expressing pre-, peri-, and postconditions in stateful-failure reactive contacts, and shown how they can be used to support automated verification through calculation. We have defined a number of novel UTP healthiness conditions for both reactive relations and reactive contracts, that capture important properties needed by the verification strategy and algebraic laws. Our theory supports most of the operators of the *Circus* language, including all the sequential operators from [35], and also parallel composition. Our theorems and verification tool can be found in our theory repository[9], together with companion proofs for the theorems presented here.

Related work includes the works of Struth *et al.* on verification of imperative programs [1, 21] using Kleene algebra for verification-condition generation, which our work heavily draws upon to deal with iteration. Automated proof support for the failures-divergences model was previously provided by the CSP-Prover tool [29], which can be used to verify infinite-state systems in CSP with Isabelle. Our work is different both in its contractual semantics, and also in our explicit handling of state, which allows us to express variable assignments. However, we believe that several of the proof tactics defined for CSP-Prover [29] could be applicable in our work for a restricted subset of reactive programs that model CSP processes.

Our work lies within the "design-by-contract" field [32], and is related to the assume-guarantee reasoning frameworks [4, 5, 40]; a detailed comparison can be found in [14]. The refinement calculus of reactive

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

systems [37] is a language based on property transformers containing trace information. Like our work, they support verification of reactive systems that are nondeterministic, non-input-receptive, and infinite state. The main differences are our handling of state variables, the basis in relational calculus, and our failures-divergences semantics. Nevertheless, our contract framework [14] can be linked to those results, and we plan to derive an assume-guarantee calculus to support verification of multi-party concurrent systems.

In future work, we will further optimise proof support for parallel composition through mechanisation of Oliveira's refinement and step laws [36], which allow efficient proof for concurrency patterns like bulk synchronous parallelism. We will also tackle the remaining operators of the *Circus* language [46], including hiding and renaming. Moreover, we hope to identify a normal form for stateful-failure reactive designs using our specialised operators, which we speculate may have the following approximate form:

$$\left[\bigwedge_{i \in I} \mathcal{I}[b_1(i), t_1(i)] \;\middle|\; \bigvee_{i \in J} \mathcal{E}[b_2(i), t_2(i), E(i)] \;\middle|\; \bigvee_{i \in K} \Phi[b_3(i), \sigma(i), t_3(i)] \right]$$

This contains a conjunction of trace assumptions, a disjunction of quiescent observation, and a disjunction of finalisers. It may well be the case that additional healthiness conditions will be required for this. We therefore will also explore additional properties that the healthiness conditions ***C2*** and ***CACC*** support. We will endeavour to establish formal links, using Galois connections, to existing semantic models like the original failure-divergences model of CSP and its healthiness conditions [38, 8]. This could provide a way of harnessing CSP-Prover proof tactics [29], and therefore expand our verification capabilities.

We also aim to apply our strategy to more substantial examples, and are currently using it to build a prototype tactic for verifying robotic controllers using a statechart-style notion with a mechanised denotational semantics [33, 12]. To support this, we will develop a *Circus*-based intermediate verification language with annotations, such as loop invariants, to provide greater automation. Further in this direction, our semantics and techniques will be also be extended to cater for real-time, probabilistic, and hybrid computational behaviours [15], which is possible due to the parametric nature of our reactive contract theory.

Acknowledgments

This research is funded by the CyPhyAssure project[10], EPSRC grant EP/S001190/1, the RoboCalc project[11], EPSRC grant EP/M025756/1, and the Royal Academy of Engineering.

[1] A. Armstrong, V. Gomes, and G. Struth. Building program construction and verification tools from algebraic principles. *Formal Aspects of Computing*, 28(2), 2015.
[2] R.-J. Back and J. Wright. *Refinement Calculus: A Systematic Introduction*. Springer, 1998.
[3] E. Bainomugisha, A. L. Carreton, T. V. Cutsem, S. Mostinckx, and W. De Meuter. A survey on reactive programming. *ACM Computing Surveys*, 45(4), August 2013.
[4] A. Benveniste, B. Caillaud, A. Ferrari, L. Mangeruca, R. Passerone, and C. Sofronis. Multiple viewpoint contract-based specification and design. In *6th Intl. Symp. on Formal Methods for Components and Objects (FMCO)*, volume 5382 of *LNCS*, pages 200–225. Springer, 2007.
[5] L. Benvenuti, A. Ferrari, L. Mangeruca, E. Mazzi, R. Passerone, and C. Sofronis. A contract-based formalism for the specification of heterogeneous systems. In *Proc. Forum on Specification, Verification, and Design Languages (FDL)*, pages 142–147, 2008.
[6] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(3):560–599, 1984.
[7] A. Cavalcanti, P. Clayton, and C. O'Halloran. From control law diagrams to ada via Circus. *Formal Aspects of Computing*, 23(4):465–512, 2011.
[8] A. Cavalcanti and J. Woodcock. A tutorial introduction to CSP in unifying theories of programming. In *PSSE*, volume 3167 of *LNCS*, pages 220–268. Springer, 2006.
[9] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.
[10] J. Foster. *Bidirectional programming languages*. PhD thesis, University of Pennsylvania, 2009.
[11] S. Foster. Kleene algebra in Unifying Theories of Programming. Technical report, University of York, 2018. http://eprints.whiterose.ac.uk/129359/.

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

[12] S. Foster, J. Baxter, A. Cavalcanti, A. Miyazawa, and J. Woodcock. Automating verification of state machines with reactive designs and Isabelle/UTP. In *Proc. 15th. Intl. Conf. on Formal Aspects of Component Software*, volume 11222 of *LNCS*. Springer, October 2018.
[13] S. Foster, J. Baxter, A. Cavalcanti, J. Woodcock, and F. Zeyda. Unifying semantic foundations for automated verification tools in Isabelle/UTP. *Science of Computer Programming*, 197, October 2020.
[14] S. Foster, A. Cavalcanti, S. Canham, J. Woodcock, and F. Zeyda. Unifying theories of reactive design contracts. *Theoretical Computer Science*, 802:105–140, January 2020.
[15] S. Foster, A. Cavalcanti, J. Woodcock, and F. Zeyda. Unifying theories of time with generalised reactive processes. *Information Processing Letters*, 135:47–52, 2018.
[16] S. Foster et al. Reactive designs in Isabelle/UTP. Technical report, University of York, 2018. http://eprints.whiterose.ac.uk/129386/.
[17] S. Foster et al. Stateful-failure reactive designs in Isabelle/UTP. Technical report, University of York, 2018. http://eprints.whiterose.ac.uk/129768/.
[18] S. Foster, K. Ye, A. Cavalcanti, and J. Woodcock. Calculational verification of reactive programs with reactive relations and Kleene algebra. In *Proc. 17th Intl. Conf. on Relational and Algebraic Methods in Computer Science (RAMICS)*, volume 11194 of *LNCS*. Springer, October 2018.
[19] S. Foster, F. Zeyda, Y. Nemouchi, P. Ribeiro, and B. Wolff. Isabelle/UTP: Mechanised Theory Engineering for Unifying Theories of Programming. *Archive of Formal Proofs*, 2019. https://www.isa-afp.org/entries/UTP.html.
[20] S. Foster, F. Zeyda, and J. Woodcock. Unifying heterogeneous state-spaces with lenses. In *ICTAC*, LNCS 9965. Springer, 2016.
[21] V. B. F Gomes and G. Struth. Modal Kleene algebra applied to program correctness. In *Formal Methods*, volume 9995 of *LNCS*, pages 310–325. Springer, 2016.
[22] W. Guttman and B. Möller. Normal design algebra. *Journal of Logic and Algebraic Programming*, 79(2):144–173, February 2010.
[23] D. Harel and A. Pneuli. On the development of reactive systems. In *Logics and Models of Concurrent Systems*, volume 13 of *NATO ASI*. Springer, 1985.
[24] J. He. From CSP to hybrid systems. In A. W. Roscoe, editor, *A classical mind: essays in honour of C. A. R. Hoare*, pages 171–189. Prentice Hall, 1994.
[25] E. C. R. Hehner. *A Practical Theory of Programming*. Springer, 1993.
[26] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
[27] C. A. R. Hoare, I. Hayes, J. He, C. Morgan, A. Roscoe, J. Sanders, I. Sørensen, J. Spivey, and B. Sufrin. The laws of programming. *Communications of the ACM*, 30(8):672–687, August 1987.
[28] C. A. R. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall, 1998.
[29] Y. Isobe and M. Roggenbach. CSP-Prover: a proof tool for the verification of scalable concurrent systems. *Journal of Computer Software, Japan Society for Software Science and Technology*, 25(4):85–92, 2008.
[30] D. Kozen. On Kleene algebras and closed semirings. In *MFCS*, volume 452 of *LNCS*, pages 26–47. Springer, 1990.
[31] D. Kozen. Kleene algebra with tests. *ACM Transactions on Programming Languages and Systems*, 19(3):427–443, May 1997.
[32] B. Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, 1992.
[33] A. Miyazawa, P. Ribeiro, W. Li, A. Cavalcanti, J. Timmis, and J. Woodcock. RoboChart: modelling and verification of the functional behaviour of robotic applications. *Software and Systems Modelling*, January 2019.
[34] B. Möller, P. Höfner, and G. Struth. Quantales and temporal logics. In *AMAST*, volume 4019 of *LNCS*, pages 263–277. Springer, 2006.
[35] M. Oliveira, A. Cavalcanti, and J. Woodcock. A UTP semantics for Circus. *Formal Aspects of Computing*, 21:3–32, 2009.
[36] M. V. M. Oliveira. *Formal Derivation of State-Rich Reactive Programs using Circus*. PhD thesis, Department of Computer Science - University of York, UK, 2006. YCST-2006-02.
[37] V. Preoteasa, I. Dragomir, and S. Tripakis. Refinement calculus of reactive systems. In *Intl. Conf. on Embedded Systems (EMSOFT)*. IEEE, October 2014.
[38] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 2005.
[39] A. W. Roscoe and D. Hopkins. SVA, a tool for analysing shared-variable programs. In *AVoCS*, pages 177–183, 2007.
[40] A. Sangiovanni-Vincentelli, W. Damm, and R. Passerone. Taming dr. frankenstein: Contract-based design for cyber-physical systems. *European Journal of Control*, 3:217–238, 2012.
[41] T. Santos, A. Cavalcanti, and A. Sampaio. Object-Orientation in the UTP. In *UTP 2006*, volume 4010 of *LNCS*, pages 20–38. Springer, 2006.
[42] A. Sherif, A. Cavalcanti, J. He, and A. Sampaio. A process algebraic framework for specification and validation of real-time systems. *Formal Aspects of Computing*, 22(2):153–191, 2010.
[43] M. Spivey. *The Z-Notation - A Reference Manual*. Prentice Hall, Englewood Cliffs, N. J., 1989.
[44] A. Tarski. On the calculus of relations. *J. Symbolic Logic*, 6(3):73–89, 1941.
[45] B. Vinter and P. Welch. Cluster computing and JCSP networking. In *Proc. Communicating Process Architectures (CPA)*, volume 60 of *Concurrent Systems Engineering*. IOS Press, 2002.
[46] J. Woodcock and A. Cavalcanti. A concurrent language for refinement. In A. Butterfield, G. Strong, and C. Pahl, editors, *Proc. 5th Irish Workshop on Formal Methods (IWFM)*, Workshops in Computing. BCS, July 2001.
[47] J. Woodcock, C. Cavalcanti, and L. Freitas. Operational semantics for model checking circus. In *Proc. 13th Intl. Symp. on Formal Methods (FM)*, volume 3582 of *LNCS*. Springer, 2005.
[48] N. Zhan, E. Y. Kang, and Z. Liu. Component publications and compositions. In *UTP*, volume 5713 of *LNCS*, pages

238–257. Springer, 2008.
[49] N. Zhan, S. Wang, and H. Zhao. *Formal Verification of Simulink/Stateflow Diagrams*. Springer, 2017.