A Tutorial Introduction to CSP in *Unifying Theories of Programming*

Ana Cavalcanti and Jim Woodcock

Department of Computer Science
University of York
York, UK

In their *Unifying Theories of Programming* (UTP), Hoare & He use the alphabetised relational calculus to give denotational semantics to a wide variety of constructs taken from different programming paradigms. In this chapter, we give a tutorial introduction to the semantics of CSP processes, as presented in Chapter 3. We start with a summarised introduction of the alphabetised relational calculus and the theory of designs, which are pre-post specifications in the style of specification statements. Afterwards, we present in detail a theory for reactive processes. Later, we combine the theories of designs and reactive processes to provide the model for CSP processes. Finally, we compare this new model with the standard failures-divergences model for CSP.

In the next section, we give an overview of the UTP, and in Section 2 we present its most general theory: the alphabetised predicates. In the following section, we establish that this theory is a complete lattice. Section 4 restricts the general theory to designs. Section 5 presents the theory of reactive processes; Section 6 contains our treatment of CSP processes; and Section 7 relates our model to Roscoe's standard model. We summarise the work in Section 8.

1 Introduction

The book by Hoare & He [117] sets out a research programme to find a common basis in which to explain a wide variety of programming paradigms: unifying theories of programming (UTP). Their technique is to isolate important language features, and give them a denotational semantics. This allows different languages and paradigms to be compared.

The semantic model is an alphabetised version of Tarski's relational calculus, presented in a predicative style that is reminiscent of the schema calculus in the Z [257] notation. Each programming construct is formalised as a relation between an initial and an intermediate or final observation. The collection of these relations forms a *theory* of the paradigm being studied, and it contains three essential parts: an alphabet, a signature, and healthiness conditions.

The *alphabet* is a set of variable names that gives the vocabulary for the theory being studied. Names are chosen for any relevant external observations of behaviour. For instance, programming variables x, y, and z would be part of the alphabet. Also, theories for particular programming paradigms require the observation of extra information; some examples are a flag that says whether

A. Cavalcanti, A. Sampaio, and J. Woodcock (Eds.): PSSE 2004, LNCS 3167, pp. 220–268, 2006.

the program has started (*okay*); the current time (*clock*); the number of available resources (*res*); a trace of the events in the life of the program (*tr*); or a flag that says whether the program is waiting for interaction with its environment (*wait*). The *signature* gives the rules for the syntax for denoting objects of the theory. *Healthiness conditions* identify properties that characterise the theory.

Each healthiness condition embodies an important fact about the computational model for the programs being studied.

Example 1 (Healthiness conditions).

1. The variable *clock* gives us an observation of the current time, which moves ever onwards. The predicate B specifies this.

   $$B \mathrel{\widehat{=}} clock \leq clock'$$

   If we add B to the description of some activity, then the variable *clock* describes the time observed immediately before the activity starts, whereas $clock'$ describes the time observed immediately after the activity ends. If we suppose that P is a healthy program, then we must have that $P \Rightarrow B$.
2. The variable *okay* is used to record whether or not a program has started. A sensible healthiness condition is that we should not observe a program's behaviour until it has started; such programs satisfy the following equation.

   $$P = (okay \Rightarrow P)$$

   If the program has not started, its behaviour is not restricted.

Healthiness conditions can often be expressed in terms of a function ϕ that makes a program healthy. There is no point in applying ϕ twice, since we cannot make a healthy program even healthier. Therefore, ϕ must be idempotent, and a healthy P must be a fixed point: $P = \phi(P)$; this equation characterises the healthiness condition. For example, we can turn the first healthiness condition above into an equivalent equation, $P = P \wedge B$, and then the following function on predicates $and_B \mathrel{\widehat{=}} \lambda X \bullet X \wedge B$ is the required idempotent.

The relations are used as a semantic model for unified languages of specification and programming. Specifications are distinguished from programs only by the fact that the latter use a restricted signature. As a consequence of this restriction, programs satisfy a richer set of healthiness conditions.

Unconstrained relations are too general to handle the issue of program termination; they need to be restricted by healthiness conditions. The result is the theory of designs, which is the basis for the study of the other programming paradigms in [117]. Here, we present the general relational setting, and the transition to the theory of designs. Next we take a different tack, and introduce the theory of reactive processes, which we then combine with designs to form the theory of CSP [115, 225].

2 The Alphabetised Relational Calculus

The alphabetised relational calculus is similar to Z's schema calculus, except that it is untyped and rather simpler. An *alphabetised predicate* $(P, Q, \ldots, \mathbf{true})$ is an alphabet-predicate pair, where the predicate's free variables are all members of the alphabet. Relations are predicates in which the alphabet is composed of undecorated variables $(x,\ y,\ z,\ \ldots)$ and dashed variables $(x',\ a',\ \ldots)$; the former represent initial observations, and the latter, observations made at a later intermediate or final point.

The alphabet of an alphabetised predicate P is denoted αP, and may be divided into its before-variables ($in\alpha P$) and its after-variables ($out\alpha P$). A *homogeneous relation* has $out\alpha P = in\alpha P'$, where $in\alpha P'$ is the set of variables obtained by dashing all variables in the alphabet $in\alpha P$. A *condition* $(b, c, d, \ldots, true)$ has an empty output alphabet.

Standard predicate calculus operators can be used to combine alphabetised predicates. Their definitions, however, have to specify the alphabet of the combined predicate. For instance, the alphabet of a conjunction is the union of the alphabets of its components: $\alpha(P \wedge Q) = \alpha P \cup \alpha Q$. If a variable is mentioned in the alphabet of P and Q, then they are both constraining the same variable.

A distinguishing feature of the UTP is its concern with program development, and consequently program correctness. A significant achievement is that the notion of program correctness is the same in every paradigm in [117]: in every state, the behaviour of an implementation implies its specification.

If we suppose that $\alpha P = \{a, b, a', b'\}$, then the *universal closure* of P is simply $\forall\, a, b, a', b' \bullet P$, which is more concisely denoted as $[\,P\,]$. The correctness of a program P with respect to a specification S is denoted by $S \sqsubseteq P$ (S is refined by P), and is defined as follows.

$$S \sqsubseteq P \quad \text{iff} \quad [\,P \Rightarrow S\,]$$

Example 2 (Refinement). Suppose we have the specification $x' > x \wedge y' = y$, and the implementation $x' = x + 1 \wedge y' = y$. The implementation's correctness can be argued as follows.

$$\begin{array}{lll} & x' > x \wedge y' = y \ \sqsubseteq\ x' = x + 1 \wedge y' = y & \sqsubseteq \\ = & [\,x' = x + 1 \wedge y' = y \Rightarrow x' > x \wedge y' = y\,] & \text{universal one-point rule, twice} \\ = & [\,x + 1 > x \wedge y = y\,] & \text{arithmetic and reflection} \\ = & true & \end{array}$$

And so, the refinement is valid.

As a first example of the definition of a programming constructor, we consider conditionals. Hoare & He use an infix syntax for the conditional operator, and define it as follows.

$$P \lhd b \rhd Q \;\widehat{=}\; (b \wedge P) \vee (\neg\, b \wedge Q) \qquad \textbf{if } \alpha b \subseteq \alpha P = \alpha Q$$
$$\alpha(P \lhd b \rhd Q) \;\widehat{=}\; \alpha P$$

Informally, $P \lhd b \rhd Q$ means P if b else Q.

The presentation of conditional as an infix operator allows the formulation of many laws in a helpful way. Below, we reproduce some of the laws presented in [117].

L1	$P \lhd b \rhd P = P$	*idempotence*
L2	$P \lhd b \rhd Q = Q \lhd \neg\, b \rhd P$	*symmetry*
L3	$(P \lhd b \rhd Q) \lhd c \rhd R = P \lhd b \wedge c \rhd (Q \lhd c \rhd R)$	*associativity*
L4	$P \lhd b \rhd (Q \lhd c \rhd R) = (P \lhd b \rhd Q) \lhd c \rhd (P \lhd b \rhd R)$	*distributivity*
L5	$P \lhd \mathit{true} \rhd Q = P = Q \lhd \mathit{false} \rhd P$	*unit*
L6	$P \lhd b \rhd (Q \lhd b \rhd R) = P \lhd b \rhd R$	*unreachable-branch*
L7	$P \lhd b \rhd (P \lhd c \rhd Q) = P \lhd b \vee c \rhd Q$	*disjunction*
L8	$(P \odot Q) \lhd b \rhd (R \odot S) = (P \lhd b \rhd R) \odot (Q \lhd b \rhd S)$	*interchange*

In Law **L8**, the symbol $\odot$ stands for any truth-functional operator.

For each operator, Hoare & He give a definition followed by a number of algebraic laws as those above. These laws can be proved from the definition; proofs omitted here can be found in [117] or [256]. We also present extra laws that are useful in later proofs, as well as in illuminating the theory. We give the laws presented in [117] that we reproduce here the same labels used in that original work: **L1**, **L2** and so on. The extra laws that we present are numbered independently.

Since a conditional is just an abbreviation for a predicate, for reasoning, we can use laws that combine programming and predicate calculus operators. An example is our first law below, which states that negating a conditional negates its operands, but not its condition.

Law 60 (not-conditional). $\neg\,(P \lhd b \rhd Q) = (\neg\, P \lhd b \rhd \neg\, Q)$

Proof.

$$\begin{aligned}
&\neg\,(P \lhd b \rhd Q) && \text{conditional}\\
&= \neg\,((b \wedge P) \vee (\neg\, b \wedge Q)) && \text{propositional calculus}\\
&= (b \Rightarrow \neg\, P) \wedge (\neg\, b \Rightarrow \neg\, Q) && \text{propositional calculus}\\
&= (b \wedge \neg\, P) \vee (\neg\, b \wedge \neg\, Q) && \text{conditional}\\
&= (\neg\, P \lhd b \rhd \neg\, Q)
\end{aligned}$$

If we apply the law of symmetry to the last result, we see that negating a conditional can be used to negate its condition, but in this case, the operands must be both negated and reversed: $\neg\,(P \lhd b \rhd Q) = (\neg\, Q \lhd \neg\, b \rhd \neg\, P)$. Even though it does not make sense to use negation in a program, for reasoning, the flexibility is very convenient.

Below is an instance of Law **L8** with a compound truth-functional operator.

Law 61 (conditional-and-not-conditional).

$$(P \lhd b \rhd Q) \wedge \neg (R \lhd b \rhd S) = (P \wedge \neg R) \lhd b \rhd (Q \wedge \neg S)$$

Proof.

$$\begin{array}{ll} & (P \lhd b \rhd Q) \wedge \neg (R \lhd b \rhd S) & \text{Law 60} \\ = & (P \lhd b \rhd Q) \wedge (\neg R \lhd b \rhd \neg S) & \text{L8} \\ = & (P \wedge \neg R) \lhd b \rhd (Q \wedge \neg S)) & \end{array}$$

As a consequence of the interchange (**L8**) and unit (**L1**) laws, any boolean operator distributes through the conditional.

Law 62 ($\odot$-conditional).

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

The details of this simple proof and of others omitted in the sequel are left as exercises. We include here only proofs for the more surprising laws or proofs that perhaps require more elaborate arguments.

Exercise 1. Prove Law 62.

A conditional may be simplified by using a known condition.

Law 63 (known-condition).

$$b \wedge (P \lhd b \rhd Q) = (b \wedge P)$$
$$\neg\, b \wedge (P \lhd b \rhd Q) = (\neg\, b \wedge Q)$$

Two absorption laws allow a conditional's operands to be simplified.

Law 64 (assume-if-condition). $(P \lhd b \rhd Q) = ((b \wedge P) \lhd b \rhd Q)$

Law 65 (assume-else-condition). $(P \lhd b \rhd Q) = (P \lhd b \rhd (\neg\, b \wedge Q))$

Sequence is modelled as relational composition. Two relations may be composed providing the output alphabet of the first is the same as the input alphabet of the second, except only for the use of dashes.

$$P(v') \,;\, Q(v) \;\widehat{=}\; \exists\, v_0 \bullet P(v_0) \wedge Q(v_0) \qquad \textbf{if } out\alpha P = in\alpha Q' = \{v'\}$$
$$in\alpha(P(v') \,;\, Q(v)) \;\widehat{=}\; in\alpha P$$
$$out\alpha(P(v') \,;\, Q(v)) \;\widehat{=}\; out\alpha Q$$

Sequence is associative and distributes backwards through the conditional.

$$\textbf{L1} \quad P \,;\, (Q \,;\, R) = (P \,;\, Q) \,;\, R \qquad \textit{associativity}$$
$$\textbf{L2} \quad (P \lhd b \rhd Q) \,;\, R = ((P \,;\, R) \lhd b \rhd (Q \,;\, R)) \qquad \textit{left-distribution}$$

The definition of assignment is basically equality; we need, however, to be careful about the alphabet. If $A = \{x, y, \ldots, z\}$ and $\alpha e \subseteq A$, where αe is the set of free variables of the expression e, the assignment $x :=_A e$ of expression e to variable x changes only x's value.

$$x :=_A e \ \widehat{=}\ (x' = e \wedge y' = y \wedge \cdots \wedge z' = z)$$
$$\alpha(x :=_A e) \ \widehat{=}\ A \cup A'$$

There is a degenerate form of assignment that changes no variable: it has the following definition.

$$\mathit{II}_A \ \widehat{=}\ (v' = v) \qquad \textbf{if } A = \{v\}$$
$$\alpha \mathit{II}_A \ \widehat{=}\ A \cup A'$$

Here, v stands for a list of observational variables. We use $v' = v$ to denote the conjunction of equalities $x' = x$, for all x in v. When clear from the context, we omit the alphabet of assignments and II.

II is the identity of sequence.

L5 $\quad P \,;\, \mathit{II}_{\alpha P} = P = \mathit{II}_{\alpha P} \,;\, P \qquad$ *unit*

Since sequence is defined in terms of the existential quantifier, there are two one-point laws. We prove one of them; the proof of the other is a simple exercise.

Law 66 (left-one-point). $(v' = e) \,;\, P \ = \ P[e/v]$
provided $\alpha P = \{v, v'\}$ *and* v' *is not free in* e.

Law 67 (right-one-point). $P \,;\, (v = e) \ = \ P[e/v']$
provided $\alpha P = \{v, v'\}$ *and* v *is not free in* e.

Proof.

$$\begin{array}{lll}
 & P \,;\, v = \mathbf{e} & \text{sequence} \\
= & \exists\, v_0 \bullet P[v_0/v'] \wedge (v = \mathbf{e})[v_0/v] & \text{substitution} \\
= & \exists\, v_0 \bullet P[v_0/v'] \wedge (v_0 = \mathbf{e}) & \text{predicate calculus and } v \text{ not free in } e \\
= & P[v_0/v'][\mathbf{e}/v_0] & \text{substitution} \\
= & P[\mathbf{e}/v'] &
\end{array}$$

Exercise 2. Prove Law **L7** above.

In theories of programming, nondeterminism may arise in one of two ways: either as the result of run-time factors, such as distributed processing; or as the under-specification of implementation choices. Either way, nondeterminism is modelled by choice; the semantics is simply disjunction.

$$P \sqcap Q \ \widehat{=}\ P \vee Q \qquad \textbf{if } \alpha P = \alpha Q$$
$$\alpha(P \sqcap Q) \ \widehat{=}\ \alpha P$$

The alphabet must be the same for both arguments.

Variable blocks are split into the commands $\mathbf{var}\, x$, which declares and introduces x in scope, and $\mathbf{end}\, x$, which removes x from scope. Their definitions are presented below, where A is an alphabet containing x and x'.

$$\mathbf{var}\, x \mathrel{\widehat=} (\exists\, x \bullet \Pi_A) \qquad \alpha(\mathbf{var}\, x) \mathrel{\widehat=} A \setminus \{x\}$$

$$\mathbf{end}\, x \mathrel{\widehat=} (\exists\, x' \bullet \Pi_A) \qquad \alpha(\mathbf{end}\, x) \mathrel{\widehat=} A \setminus \{x'\}$$

The relation $\mathbf{var}\, x$ is not homogeneous, since it does not include x in its alphabet, but it does include x'; similarly, $\mathbf{end}\, x$ includes x, but not x'.

The results below state that following a variable declaration by a program Q makes x local in Q; similarly, preceding a variable undeclaration by a program Q makes x' local.

$$(\mathbf{var}\, x \,;\, Q) = (\exists\, x \bullet Q)$$

$$(Q \,;\, \mathbf{end}\, x) = (\exists\, x' \bullet Q)$$

More interestingly, we can use $\mathbf{var}\, x$ and $\mathbf{end}\, x$ to specify a variable block.

$$(\mathbf{var}\, x \,;\, Q \,;\, \mathbf{end}\, x) = (\exists\, x, x' \bullet Q)$$

In programs, we use $\mathbf{var}\, x$ and $\mathbf{end}\, x$ paired in this way, but the separation is useful for reasoning.

Exercise 3. Prove the above equality.

Variable blocks introduce the possibility of writing programs and equations like that below.

$$\begin{aligned}&(\mathbf{var}\, x \,;\, x := 2 * y \,;\, w := 0 \,;\, \mathbf{end}\, x)\\ &\quad = (\mathbf{var}\, x \,;\, x := 2 * y \,;\, \mathbf{end}\, x) \,;\, w := 0\end{aligned}$$

Clearly, the assignment to w may be moved out of the scope of the declaration of x, but what is the alphabet in each of the assignments to w? If the only variables are w, x, and y, and $A = \{w, y, w', y'\}$, then the assignment on the right has the alphabet A; but the alphabet of the assignment on the left must also contain x and x', since they are in scope. There is an explicit operator for making alphabet modifications such as this: *alphabet extension*.

$$\begin{aligned}&P_{+x} \mathrel{\widehat=} P \wedge x' = x && \mathbf{for}\ \ x, x' \notin \alpha P\\ &\alpha(P_{+x}) \mathrel{\widehat=} \alpha P \cup \{x, x'\}\end{aligned}$$

In our example, if the right-hand assignment is $P \mathrel{\widehat=} w :=_A 0$, then the left-hand assignment is denoted by P_{+x}.

The next programming operator of interest is recursion. We define it in the next section, where we explain that the UTP general theory of relations is a complete lattice, a notion introduced in Chapter 0.

3 The Complete Lattice

As already explained in Chapter 0, the refinement ordering is a partial order: reflexive, anti-symmetric, and transitive; this also holds for refinement as defined in the UTP. Moreover, the set of alphabetised predicates with a particular alphabet A is a complete lattice under the refinement ordering. Its bottom element is denoted $\bot_{\mathrm{A}}$, and is the weakest predicate **true**; this is the program that behaves quite arbitrarily. The top element is denoted $\top_A$, and is the strongest predicate ***false***; this is the program that performs miracles and implements every specification (see Chapter 0). These properties of abort and miracle are captured in the following two laws, which hold for all P with alphabet A.

L1 $\bot_{\mathrm{A}} \sqsubseteq P$ *bottom-element*

L2 $P \sqsubseteq \top_{\mathrm{A}}$ *top-element*

The least upper bound is not defined in terms of the relational model, but by the Law **L1** below; this is because, in general, it is not possible to give such definition. Fortunately, this law indirectly specifies the least upper bound operator; alone, it is enough to prove Laws **L1A** and **L1B**, which are actually more useful in proofs.

L1 $P \sqsubseteq (\sqcap S)$ iff $(P \sqsubseteq X$ for all X in $S)$ *unbounded-nondeterminism*

L1A $(\sqcap S) \sqsubseteq X$ for all X in S *lower-bound*

L1B if $P \sqsubseteq X$ for all X in S, then $P \sqsubseteq (\sqcap S)$ *greatest-lower-bound*

These laws characterise basic properties of least upper bounds. In particular, Law **L1B** is simply **L1**, from right to left.

A function F is *monotonic* if, and only if, $P \sqsubseteq Q \Rightarrow F(P) \sqsubseteq F(Q)$. Operators like conditional and sequence are monotonic; negation is not. There is a class of operators that are all monotonic: the disjunctive operators. For example, sequence is disjunctive in both arguments.

L6 $(P \sqcap Q)\,;\, R = (P\,;\, R) \sqcap (Q\,;\, R)$ *sequence-⊓-left-distribution*

L7 $P\,;\,(Q \sqcap R) = (P\,;\, Q) \sqcap (P\,;\, R)$ *sequence-⊓-right-distribution*

Exercise 4. Prove Law **L6** above.

Since alphabetised relations form a complete lattice, every construction defined solely using monotonic operators has a fixed point (see Chapter 1, Section 8). Even more, a result by Tarski says that the set of fixed points is a complete

lattice. The extreme points in this lattice are often of interest; for example, $\top$ is the strongest fixed point of $X = X \,;\, P$, and $\bot$ is the weakest.

The weakest fixed point of the function F is denoted by $\mu\, F$, and is simply the greatest lower bound (the *weakest*) of all the fixed points of F.

$$\mu\, F \;\widehat{=}\; \sqcap\{\, X \mid F(X) \sqsubseteq X \,\}$$

The strongest fixed point νF is the dual of the weakest fixed point.

Hoare & He use weakest fixed points to define recursion. They write a recursive program as $\mu\, X \bullet \mathcal{C}(X)$, where $\mathcal{C}(X)$ is a predicate that is constructed using monotonic operators and the variable X. As opposed to the variables in the alphabet, X stands for a predicate itself, and we call it the recursive variable. Intuitively, occurrences of X in $\mathcal{C}$ stand for recursive calls to $\mathcal{C}$ itself. The definition of recursion is as follows.

$$\mu\, X \bullet \mathcal{C}(X) \;\widehat{=}\; \mu\, F \qquad \textbf{where } F \;\widehat{=}\; \lambda\, X \bullet \mathcal{C}(X)$$

The standard laws that characterise weakest fixed points are valid.

L1 $\quad \mu\, F \sqsubseteq Y$ if $F(Y) \sqsubseteq Y$ *weakest-fixed-point*

L2 $\quad F(\mu\, F) = \mu\, F$ *fixed-point*

Law **L1** establishes that $\mu\, F$ is weaker than any fixed point; **L2** states that $\mu\, F$ is itself a fixed point. From a programming point of view, **L2** is just the copy rule.

The while loop is written $b * P$: while b is true, execute the program P. This can be defined in terms of the weakest fixed point of a conditional expression.

$$b * P \;\widehat{=}\; \mu\, X \bullet (\,(P \,;\, X) \lhd b \rhd \mathit{II}\,)$$

Example 3 (Non-termination). If b always remains true, then obviously the loop $b * P$ never terminates, but what is the semantics for this? The simplest example of such an iteration is $\mathit{true} * \mathit{II}$, which has the semantics $\mu\, X \bullet X$.

$$\begin{array}{ll}
\quad \mu\, X \bullet X & \text{least fixed point} \\
= \sqcap\{\, Y \mid (\lambda\, X \bullet X)(Y) \sqsubseteq Y \,\} & \text{function application} \\
= \sqcap\{\, Y \mid Y \sqsubseteq Y \,\} & \text{reflexivity of } \sqsubseteq \\
= \sqcap\{\, Y \mid \mathit{true} \,\} & \text{property of } \sqcap \\
= \bot &
\end{array}$$

Exercise 5. Convince yourself that $\mathit{true} * \mathit{II} = \mu\, X \bullet X$ using the laws presented so far.

Surprisingly, it is possible to use the result Example 3 to show that a program may be able to recover from a non-terminating loop!

Example 4 (Aborting loop). Suppose that the sole state variable is x and that c is a constant.

$$
\begin{array}{lr}
(\mathbf{true} * II)\ ;\ x := c & \text{Example 3} \\
= \bot\ ;\ x := c & \bot \\
= \mathbf{true}\ ;\ x := c & \text{assignment} \\
= \mathbf{true}\ ;\ x' = c & \text{sequence} \\
= \exists\, x_0 \bullet \mathbf{true} \wedge x' = c & \text{predicate calculus} \\
= x' = c & \text{assignment} \\
= x := c &
\end{array}
$$

Example 4 is rather disconcerting: in ordinary programming, there is no recovery from a non-terminating loop. It is the purpose of *designs* to overcome this deficiency in the programming model.

4 Designs

The problem pointed out above in Section 3 can be explained as the failure of general alphabetised predicates P to satisfy the equation below.

$$\mathbf{true}\ ;\ P \ =\ \mathbf{true}$$

We presented in Example 4 a program consisting of a non-terminating loop followed by an assignment, and whose overall behaviour was to ignore the loop and execute the assignment. This is not how programs work in practice. The solution to this problem is to consider a subset of the alphabetised predicates in which a particular observational variable, called *okay*, is used to record information about the start and termination of programs. The above equation holds for predicates P in this set. As an aside, note that **false** cannot possibly belong to this set, since $\mathbf{true}\ ;\ \mathbf{false} = \mathbf{false}$.

The predicates in this subset are called *designs*. They can be split into precondition-postcondition pairs, and are a basis for unifying languages and methods like B [3], VDM [134], Z [257], and refinement calculi [192, 17, 199]. They are similar to the specification statements introduced in Chapter 0.

In designs, *okay* records that the program has started, and $okay'$ that it has terminated. In implementing a design, we may assume that the precondition holds, but we have to fulfill the postcondition. In addition, we can rely on the program being started, but we must ensure that it terminates. If the precondition does not hold, or the program does not start, we are not committed to establish the postcondition nor even to make the program terminate.

A design with precondition P and postcondition Q is written $(\, P \vdash Q \,)$. It is defined as follows.

$$(\, P \vdash Q \,) \ \widehat{=}\ (\, okay \wedge P \Rightarrow okay' \wedge Q \,)$$

If the program starts in a state satisfying P, then it will terminate, and on termination Q will be true.

Example 5 (Pre-post specifications). Suppose that we have a program with state variables x and y, and we want to specify that, providing that x is strictly positive, x must be decreased whilst y is kept constant. The design that formalises this is

$$x > 0 \vdash x' < x \land y' = y$$

This is more or less the same in Morgan's refinement calculus, where the specification statement below could be used.

$$x : [\, x > 0, x < x_0 \,]$$

Notice how the postcondition uses different conventions for distinguishing between before and after variables; also notice that the specification is prefixed with a *frame* listing the variables that are permitted to change in order to satisfy the postcondition.

Abort and miracle are defined as designs in the following examples. Abort has precondition **false**: it is never guaranteed to terminate.

Example 6 (Abort).

$$\begin{array}{lr}
\quad \mathbf{false} \vdash \mathbf{false} & \text{design} \\
= \mathit{okay} \land \mathbf{false} \Rightarrow \mathit{okay}' \land \mathbf{false} & \mathbf{false} \text{ zero for conjunction} \\
= \mathbf{false} \Rightarrow \mathit{okay}' \land \mathbf{false} & \text{vacuous implication} \\
= \mathbf{true} & \text{vacuous implication} \\
= \mathbf{false} \Rightarrow \mathit{okay}' \land \mathbf{true} & \mathbf{false} \text{ zero for conjunction} \\
= \mathit{okay} \land \mathbf{false} \Rightarrow \mathit{okay}' \land \mathbf{true} & \text{design} \\
= \mathbf{false} \vdash \mathbf{true} &
\end{array}$$

Miracle has precondition **true**, and establishes the impossible: **false**.

Example 7 (Miracle).

$$\begin{array}{lr}
\quad \mathbf{true} \vdash \mathbf{false} & \text{design} \\
= \mathit{okay} \land \mathbf{true} \Rightarrow \mathit{okay}' \land \mathbf{false} & \mathbf{true} \text{ unit for conjunction} \\
= \mathit{okay} \Rightarrow \mathbf{false} & \text{contradiction} \\
= \neg\, \mathit{okay} &
\end{array}$$

Exercise 6. Prove that abort is refined by every other design, and that every design is refined by miracle.

In VDM, B, and the refinement calculus, a pre-post specification may be refined by weakening the precondition. This refinement step improves the specification, since there are some states in which execution of the original specification leads to abortion, but execution of the resulting specification has a well-defined behaviour.

A pre-post specification may also be refined by strengthening the postcondition. Again, this is an improvement, since more is known about the result.

Example 8 (Refining designs). In Example 5, the design aborts unless $x > 0$; we can improve this by requiring it to work when $x = 0$. This weakens the precondition, since $x > 0 \Rightarrow x \geq 0$. The design requires that the after-value of x should be strictly less than the before-value of x. We can strengthen this by saying how much smaller it should be. For instance, we could require that $x' = x - 1$. Moreover, we can weaken the precondition and strengthen the postcondition simultaneously. For example, the design below is a refinement of that in Example 5.

$$x \geq 0 \vdash (x > 0 \Rightarrow x' = 0) \wedge (x = 0 \Rightarrow x' = 1)$$

The behaviour for $x = 0$ is not related to that for when $x > 0$. This is an improvement in the sense that, within the old precondition, the new postcondition is stronger than the old postcondition.

We saw earlier that refinement between relations is just reverse implication; since designs are a special case of relations, it would be nice if the notion of refinement did not change. A reassuring result is that refinement of designs in the relational sense does amount to either weakening the precondition, or strengthening the postcondition in the presence of the precondition as expected. This is established by the result below.

Law 68 (refinement-of-designs).

$$P_1 \vdash Q_1 \sqsubseteq P_2 \vdash Q_2 = [\, P_1 \wedge Q_2 \Rightarrow Q_1 \,] \wedge [\, P_1 \Rightarrow P_2 \,]$$

Proof.

$$\begin{array}{lr}
& P_1 \vdash Q_1 \sqsubseteq P_2 \vdash Q_2 \qquad \sqsubseteq \\
= & [\,(\, P_2 \vdash Q_2 \,) \Rightarrow (\, P_1 \vdash Q_1 \,)\,] \qquad \text{definition of design, twice} \\
= & [\,(\, \mathit{okay} \wedge P_2 \Rightarrow \mathit{okay}' \wedge Q_2 \,) \Rightarrow (\, \mathit{okay} \wedge P_1 \Rightarrow \mathit{okay}' \wedge Q_1 \,)\,] \\
& \qquad \text{case split } \mathit{okay} \\
= & [\,(\, P_2 \Rightarrow \mathit{okay}' \wedge Q_2 \,) \Rightarrow (\, P_1 \Rightarrow \mathit{okay}' \wedge Q_1 \,)\,] \qquad \text{case split } \mathit{okay}' \\
= & [\,(\neg\, P_2 \Rightarrow \neg\, P_1 \,) \wedge ((\, P_2 \Rightarrow Q_2 \,) \Rightarrow (\, P_1 \Rightarrow Q_1 \,))\,] \\
& \qquad \text{propositional calculus} \\
= & [\,(\, P_1 \Rightarrow P_2 \,) \wedge ((\, P_2 \Rightarrow Q_2 \,) \Rightarrow (\, P_1 \Rightarrow Q_1 \,))\,] \qquad \text{predicate calculus} \\
= & [\, P_1 \Rightarrow P_2 \,] \wedge [\, P_1 \wedge Q_2 \Rightarrow Q_1 \,]
\end{array}$$

Exercise 7. Use Law 68 to prove the refinements in Example 8.

Sometimes, we need to refer to the precondition in the postcondition; this is called *exporting the precondition.*

Lemma 1 (export-precondition).

$$(P \vdash Q) = (P \vdash P \wedge Q)$$

Proofs of this lemma and of some our other lemmas and theorems can be found in Appendix B. They are usually results stated, but possibly not proved in [117], and results that we need in the proofs of our laws.

The most important result in the theory of designs, however, is that abort is a zero for sequence. This was, after all, the whole point for the introduction of designs. First, we introduce a lemma relating designs and abort.

Lemma 2 (design-abort). *When a design has not started ($\neg$ okay), it offers no guarantees.*

$$(P \vdash Q)[\mathit{false}/\mathit{okay}] = \mathbf{true}$$

This result holds for miracle as well, because even miracle cannot help if it does not start.

The left-zero law now follows from this lemma, since one possibility for abort is to make okay' false. If this is followed by a design, then the design will attempt to start in a state with *okay* false, and Lemma 2 will be relevant.

L1 $\quad \mathbf{true} \,;\, (P \vdash Q) = \mathbf{true}$ *left-zero*

Proof.

$$\begin{array}{ll}
& \mathbf{true} \,;\, (P \vdash Q) & \text{sequence} \\
= & \exists\, \mathit{okay}_0, v_0 \bullet \mathbf{true}[\mathit{okay}_0, v_0/\mathit{okay}', v'] \wedge (P \vdash Q)[\mathit{okay}_0, v_0/\mathit{okay}, v] & \\
& & \text{predicate calculus} \\
= & \exists\, \mathit{okay}_0 \bullet \exists\, v_0 \bullet \mathbf{true}[\mathit{okay}_0/\mathit{okay}'][v_0/v'] \wedge (P \vdash Q)[\mathit{okay}_0/\mathit{okay}][v_0/v] & \\
& & \text{sequence} \\
= & \exists\, \mathit{okay}_0 \bullet \mathbf{true}[\mathit{okay}_0/\mathit{okay}'] \,;\, (P \vdash Q)[\mathit{okay}_0/\mathit{okay}] & \text{case split } \mathit{okay}_0 \\
= & \mathbf{true}[\mathit{true}/\mathit{okay}'] \,;\, (P \vdash Q)[\mathit{true}/\mathit{okay}] \vee & \text{substitution, design-abort} \\
& \mathbf{true}[\mathit{false}/\mathit{okay}'] \,;\, (P \vdash Q)[\mathit{false}/\mathit{okay}] & \\
= & \mathbf{true} \,;\, (P \vdash Q)[\mathit{true}/\mathit{okay}] \vee \mathbf{true} \,;\, \mathbf{true} & \text{relational calculus} \\
= & \mathbf{true} &
\end{array}$$

In this new setting, it is necessary to redefine assignment and $\mathit{I\!I}$, as those introduced previously are not designs.

$$(x := e) \;\widehat{=}\; (\,\mathbf{true} \vdash x' = e \wedge y' = y \wedge \cdots \wedge z' = z\,)$$

$$\mathit{I\!I}_D \;\widehat{=}\; (\,\mathbf{true} \vdash \mathit{I\!I}\,)$$

Their existing laws hold, but it is necessary to prove them again, as their definitions have changed.

L2 $\quad (v := e \,;\, v := f(v)) = (v := f(e))$ *assignment-composition*

L3 $\quad (v := e \,;\, (P \lhd b(v) \rhd Q)) = ((v := e \,;\, P) \lhd b(e) \rhd (v := e \,;\, Q))$ *assignment-conditional-left-distribution*

L4 $\quad (\mathit{I\!I}_D \,;\, (P \vdash Q)) = (P \vdash Q)$ *left-unit*

Proof of **L2**.

$$
\begin{array}{ll}
& v := e \,;\, v := f(v) & \text{assignment, twice} \\
= & (\, \mathbf{true} \vdash v' = e \,) \,;\, (\, \mathbf{true} \vdash v' = f(v) \,) & \text{sequence,case split } okay_0 \\
= & (\, (\, \mathbf{true} \vdash v' = e \,)[true/okay'] \,;\, (\, \mathbf{true} \vdash v' = f(v) \,)[true/okay] \,) \vee \\
& \neg\, okay \,;\, \mathbf{true} & \text{design} \\
= & (\, (\, okay \Rightarrow v' = e \,) \,;\, (\, okay' \wedge v' = f(v) \,) \,) \vee \neg\, okay & \text{relational calculus} \\
= & okay \Rightarrow (\, v' = e \,;\, (\, okay' \wedge v' = f(v) \,) \,) & \text{assignment composition} \\
= & okay \Rightarrow okay' \wedge v' = f(e) & \text{design} \\
= & (\, \mathbf{true} \vdash v' = f(e) \,) & \text{assignment} \\
= & v := f(e)
\end{array}
$$

When program operators are applied to designs, the result is also a design. This follows from the laws below, for choice, conditional, sequence, and recursion. These are stated in [117] as theorems. We label them **T1**, **T2**, and **T3**, and add a simplified version of **T3**, which is mentioned in [117] and we call **T3′**.

A choice between two designs is guaranteed to terminate when they both do; since either of them may be chosen, either postcondition may be established.

T1 $(\, (P_1 \vdash Q_1) \sqcap (P_2 \vdash Q_2) \,) = (\, P_1 \wedge P_2 \vdash Q_1 \vee Q_2 \,)$

Exercise 8. Prove Law **T1**.

If the choice between two designs depends on a condition b, then so do the precondition and postcondition of the resulting design.

T2 $(\, (P_1 \vdash Q_1) \lhd b \rhd (P_2 \vdash Q_2) \,) = ((P_1 \lhd b \rhd P_2) \vdash (\, Q_1 \lhd b \rhd Q_2) \,)$

A sequence of designs $(P_1 \vdash Q_1)$ and $(P_2 \vdash Q_2)$ terminates when P_1 holds (that is, $\neg\, (\neg\, P_1 \,;\, \mathbf{true})$), and Q_1 is guaranteed to establish P_2 $(\neg\, (Q_1 \,;\, \neg\, P_2))$. On termination, the sequence establishes the composition of the postconditions.

T3 $(\, (P_1 \vdash Q_1) \,;\, (P_2 \vdash Q_2) \,)$
$= (\, (\neg\, (\neg\, P_1 \,;\, \mathbf{true}) \wedge \neg\, (Q_1 \,;\, \neg\, P_2)) \vdash (Q_1 \,;\, Q_2) \,)$

We have said nothing in our discussion of designs that requires a precondition to be a simple condition rather than a relation, and this fact complicates the statement of Law **T3**; if P_1 actually is a condition, then $\neg\, (\neg\, P_1 \,;\, \mathbf{true})$ can be simplified.

T3′ $(\, (p_1 \vdash Q_1) \,;\, (P_2 \vdash Q_2) \,) = (\, (p_1 \wedge \neg\, (Q_1 \,;\, \neg\, P_2)) \vdash (Q_1 \,;\, Q_2) \,)$

To understand the simplification, consider the following lemma.

Lemma 3 (condition-right-unit). *Abort is a right-unit for conditions.*

$p \,;\, \mathbf{true} = p$

A dual result is that abort is a left-unit for conditions on after-states: that is, $(\mathbf{true} \,;\, p') = p'$. We give two last results of this kind before we move on.

Lemma 4 (abort-condition).

$$\mathbf{true} \,;\, p \;=\; \exists\, v \bullet p$$

A special case of this lemma involves *okay* or, more generally, any boolean variable used as a condition.

Lemma 5 (abort-boolean). *Provided b is a boolean variable,*

$$\mathbf{true} \,;\, b \;=\; \mathbf{true}$$

A recursively defined design has as its body a function on designs; as such, it can be seen as a function on pre-post pairs (X, Y). Moreover, since the result of the function is itself a design, it can be written in terms of a pair of functions F and G, one for the precondition and one for the postcondition.

As the recursive design is executed, the precondition F is required to hold over and over again. The strongest recursive precondition so obtained has to be satisfied, if we are to guarantee that the recursion terminates. Similarly, the postcondition is established over and over again, in the context of the precondition. The weakest result that can possibly be obtained is that which can be guaranteed by the recursion.

T4 $\quad (\,\mu\, X, Y \bullet (\, F(X, Y) \vdash G(X, Y)\,)\,) = (\, P(Q) \vdash Q\,)$
where $P(Y) = (\,\nu X \bullet F(X, Y)\,)$ and $Q = (\,\mu\, Y \bullet P(Y) \Rightarrow G(P(Y), Y)\,)$

Further intuition comes from the realisation that we want the least refined fixed point of the pair of functions. That comes from taking the strongest precondition, since the precondition of every refinement must be weaker, and the weakest postcondition, since the postcondition of every refinement must be stronger.

Like the set of general alphabetised predicates, designs form a complete lattice. We have already presented the top and the bottom (miracle and abort).

$$\top_D \;\widehat{=}\; (\,\mathbf{true} \vdash \mathbf{false}\,) = \neg\, \mathit{okay}$$

$$\bot_D \;\widehat{=}\; (\,\mathbf{false} \vdash \mathbf{true}\,) = \mathbf{true}$$

Example 9 (Abort). All useful work is discarded once a program aborts. This is shown in the following derivation.

$$\begin{array}{lr}
\quad x := e \,;\, \bot_D & \text{assignment, bottom} \\
= (\,\mathbf{true} \vdash x' = e\,) \,;\, (\,\mathbf{false} \vdash \mathbf{true}\,) & \text{design sequence} \\
= \mathbf{true} \wedge \neg\, (x' = e \,;\, \neg\, \mathbf{false}) \vdash x' = e \,;\, \mathbf{true} & \text{relational calculus} \\
= \neg\, (x' = e \,;\, \mathbf{true}) \vdash \mathbf{true} & \text{relational calculus} \\
= \neg\, \mathbf{true} \vdash \mathbf{true} & \text{propositional calculus} \\
= \mathbf{false} \vdash \mathbf{true} & \text{bottom} \\
= \bot_D &
\end{array}$$

Abort, however, is not a right-zero for sequence, since it will not take over if it is preceded by a miraculous program.

The greatest lower-bound and the least upper-bound are established in the following theorem.

Theorem 1 (meets-and-joins).

$$\sqcap_i \bullet (P_i \vdash Q_i) = (\bigwedge_i \bullet P_i) \vdash (\bigvee_i \bullet Q_i)$$

$$\sqcup_i \bullet (P_i \vdash Q_i) = (\bigvee_i \bullet P_i) \vdash (\bigwedge_i \bullet P_i \Rightarrow Q_i)$$

As with the binary choice, the choice $\sqcap_i \bullet (P_i \vdash Q_i)$ over the set of designs $(P_i \vdash Q_i)$ terminates when all the designs do, and it establishes one of the possible postconditions. The least upper-bound models a form of choice that is conditioned by termination: only the terminating designs can be chosen. The choice terminates if any of the designs do, and the postcondition established is that of any of the terminating designs.

Designs are special kinds of relations, which in turn are special kinds of predicates, and so they can be combined with the propositional operators. A design can be negated, although the result is not itself a design.

Lemma 6 (not-design).

$$\neg (P \vdash Q) = (\mathit{okay} \wedge P \wedge (\mathit{okay}' \Rightarrow \neg Q))$$

If the postcondition of a design promises the opposite of its precondition, then the design is miraculous.

Law 69 (design-contradiction). $(P \vdash \neg P) = (P \vdash \mathbf{false})$

Proof.

$$\begin{array}{lr} P \vdash \neg P & \text{export-precondition} \\ = P \vdash P \wedge \neg P & \text{propositional calculus} \\ = P \vdash \mathbf{false} & \end{array}$$

Another way of characterising the set of designs is by imposing healthiness conditions on alphabetised predicates. Hoare & He identify four healthiness conditions that they consider of interest: ***H1*** to ***H4***. We discuss two of them.

4.1 *H1*: Unpredictability

A relation P is ***H1***-healthy if and only if $P = (\mathit{okay} \Rightarrow P)$. This means that observations cannot be made before the program has started. The idempotent corresponding to this healthiness condition is defined as

$$\mathbf{H1}(P) = \mathit{okay} \Rightarrow P$$

It is indeed an idempotent, since implication is idempotent in its first argument.

Law 70 (H1-idempotent). $\boldsymbol{H1} \circ \boldsymbol{H1} = \boldsymbol{H1}$

Proof.

$$
\begin{array}{lr}
\quad \boldsymbol{H1} \circ \boldsymbol{H1}(P) & \boldsymbol{H1} \\
= \mathit{okay} \Rightarrow (\mathit{okay} \Rightarrow P) & \text{propositional calculus} \\
= \mathit{okay} \wedge \mathit{okay} \Rightarrow P & \text{propositional calculus} \\
= \mathit{okay} \Rightarrow P & \boldsymbol{H1} \\
= \boldsymbol{H1}(P)
\end{array}
$$

*Example 10 (**H1** relations).* The following are examples of **H1** relations.

1. The relation **true**, since $(\mathit{okay} \Rightarrow \mathbf{true}) = \mathbf{true}$; it is also the design $\bot_D$: abort.
2. The relation $\neg\ \mathit{okay}$, since $(\mathit{okay} \Rightarrow \neg\ \mathit{okay}) = \neg\ \mathit{okay}$; it is also the design $\top_D$: miracle.
3. The relation $(\mathit{okay} \wedge x \neq 0 \Rightarrow x' < x)$, which, when started in a state where *okay* is true and $x \neq 0$, ensures that the after value of x is strictly less than its before value.
4. The design $(x \neq 0 \vdash x' < x)$, which, when started in a state where *okay* is true and $x \neq 0$, ensures *termination* and that the after value of x is strictly less than its before value.

Healthiness conditions give a way of imposing structure on a subset of relations, and **H1**-relations have some interesting algebraic properties. First, all **H1**-relations have a left zero.

Lemma 7 (H1-left-zero). *Provided P is **H1**-healthy,*

$$\mathbf{true}\ ;\ P = \mathbf{true}$$

All **H1**-relations have a left unit.

Lemma 8 (H1-left-unit). *Provided P is **H1**-healthy,*

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

Finally, relations that have both left units and left zeros are also **H1**.

Lemma 9 (left-unit-zero-H1). *Provided P has a left unit and a left zero,*

$$P\ =\ (\mathit{okay} \Rightarrow P)$$

These three lemmas allow us to characterise **H1**-relations algebraically: they are exactly those relations that satisfy the left zero and left unit laws.

Theorem 2 (H1-healthiness).

$$(P = \boldsymbol{H1}(P))\ =\ ((\mathbf{true}\ ;\ P = \mathbf{true})\ \wedge\ (\mathit{II}_D\ ;\ P = P))$$

We conclude this section by investigating a few more of **H1**'s properties. It relates the two identities that we have seen so far.

Law 71 (II_D-*H1*-II). $\mathit{II}_D = \boldsymbol{H1}(\mathit{II})$

Proof.

$$
\begin{array}{lr}
\mathit{II}_D & \mathit{II}_D \\
= (\mathbf{true} \vdash \mathit{II}) & \text{design} \\
= (\mathit{okay} \Rightarrow \mathit{okay}' \wedge \mathit{II}) & \mathit{II} \\
= (\mathit{okay} \Rightarrow \mathit{okay}' \wedge \mathit{II} \wedge \mathit{okay}' = \mathit{okay}) & \text{propositional calculus} \\
= (\mathit{okay} \Rightarrow \mathit{II} \wedge \mathit{okay}' = \mathit{okay}) & \mathit{II} \\
= (\mathit{okay} \Rightarrow \mathit{II}) & \boldsymbol{H1} \\
= \mathit{II}
\end{array}
$$

H1 tells us that, try as we might, we simply cannot make an observation of the behaviour of a design until after it has started. A design with a rogue postcondition, such as $(\mathbf{true} \vdash (\neg\ \mathit{okay} \Rightarrow x' = 0))$, tries to violate ***H1***, but it cannot. We could simplify it by expanding the definition of a design, and then simplifying the result with propositional calculus. It is possible to avoid this expansion by applying ***H1*** directly to the postcondition.

Law 72 (design-post-*H1*). $(P \vdash Q) \ = \ (P \vdash \boldsymbol{H1}(Q))$

Proof.

$$
\begin{array}{lr}
P \vdash \boldsymbol{H1}(Q) & \boldsymbol{H1} \\
= P \vdash (\mathit{okay} \Rightarrow Q) & \text{design} \\
= \mathit{okay} \wedge P \Rightarrow \mathit{okay}' \wedge (\mathit{okay} \Rightarrow Q) & \text{propositional calculus} \\
= \mathit{okay} \wedge P \Rightarrow \mathit{okay}' \wedge Q & \text{design} \\
= P \vdash Q
\end{array}
$$

We can also push the application of ***H1*** in a postcondition through a negation.

Law 73 (design-post-not-*H1*). $(P \vdash \neg\ Q) \ = \ (P \vdash \neg\ \boldsymbol{H1}(Q))$

Proof.

$$
\begin{array}{lr}
P \vdash \neg\ \boldsymbol{H1}(Q) & \boldsymbol{H1} \\
= P \vdash \neg\ (\mathit{okay} \Rightarrow Q) & \text{propositional calculus} \\
= P \vdash \mathit{okay} \wedge \neg\ Q & \text{design} \\
= \mathit{okay} \wedge P \Rightarrow \mathit{okay}' \wedge \mathit{okay} \wedge \neg\ Q & \text{propositional calculus} \\
= \mathit{okay} \wedge P \Rightarrow \mathit{okay}' \wedge \neg\ Q & \text{design} \\
= P \vdash \neg\ Q
\end{array}
$$

H1 enjoys many other properties, some of which we see later in this chapter.

4.2 *H2*: Termination Always Possible

The second healthiness condition is $[\, P[\mathit{false}/\mathit{okay}'] \Rightarrow P[\mathit{true}/\mathit{okay}']\,]$. This means that if P is satisfied when okay' is *false*, it is also satisfied then okay' is *true*. In other words, P cannot *require* nontermination, so that termination is always a possibility.

*Example 11 (**H2** predicates).*

1. The design relations abort and miracle are both ***H2***, since they leave the value of okay' completely unconstrained.
2. The relation $(\mathit{okay}' \wedge (x' = 0))$ is ***H2***, since it insists on termination.
3. The design $(x \neq 0 \vdash x' < x)$ is ***H2***, since (a) if it is not started properly $(\neg\ \mathit{okay})$, or if $x = 0$, then it leaves okay' unconstrained; and (b) if it is started properly (okay) and $x \neq 0$, then it insists on termination.

If P is a predicate with okay' in its alphabet, we abbreviate $P[b/\mathit{okay}']$ as P^b, for boolean value b. Thus, P is ***H2***-healthy if and only if $[\, P^f \Rightarrow P^t\,]$, where f and t are used as abbreviations for **false** and **true**.

This healthiness condition may also be described in terms of an idempotent. For that, we define the following predicate.

Definition 1 (The idempotent J).

$$J \mathrel{\widehat{=}} (\mathit{okay} \Rightarrow \mathit{okay}') \wedge \mathbb{I}_{rel}^{-\mathit{okay}}$$

J permits a change in the value of *okay*, while the remaining variables stay constant: if *okay* is changed, then it can be weakened, but not strengthened. We use $\mathbb{I}_{rel}^{-\mathit{okay}}$ to denote the conjunction of equalities $v' = v$ for all variables v in the alphabet, except *okay*.

The relationship between J and ***H2*** (see Theorem 3) is based on an important property called J-split. As its name suggests, it divides a relation into two parts, but we must notice the asymmetry.

Lemma 10 (J-split). *Provided okay and okay' are in the alphabet of P,*

$$P\ ;\ J \ =\ P^f \vee (P^t \wedge \mathit{okay}')$$

The healthiness condition ***H2*** may now be expressed using J, as we show in the following theorem, which uses J-split.

Theorem 3 (*H2* equivalence). *There are two equivalent ways of characterising **H2**-healthy relations.*

$$(P = P\ ;\ J) \ =\ [\, P^f \Rightarrow P^t\,]$$

Based on this result, we use ***H2*** to refer to the function $\mathbf{H2}(P) = P;\ J$.

Interestingly, J is actually an ***H2***-healthy relation.

Lemma 11 (J is *H2*). *J is **H2**-healthy.*

$$J = \mathbf{H2}(J)$$

This lemma makes it easy to show that J really is an idempotent.

Law 74 (*H2*-idempotent). $\boldsymbol{H2} \circ \boldsymbol{H2} = \boldsymbol{H2}$

Proof.

$$
\begin{array}{lr}
\quad \boldsymbol{H2} \circ \boldsymbol{H2}(P) & \boldsymbol{H2} \\
= (P \,;\, J) \,;\, J & \text{associativity} \\
= P \,;\, (J \,;\, J) & \boldsymbol{H2} \\
= P \,;\, \boldsymbol{H2}(J) & J \ \boldsymbol{H2}\text{-healthy} \\
= P \,;\, J & \boldsymbol{H2} \\
= \boldsymbol{H2}(P) &
\end{array}
$$

As we see in the next two examples, the original formulation of ***H2*** is often easier to use in demonstrating that a relation is ***H2***; however, because the description based on J is an idempotent function, it has some interesting algebraic properties. First, we prove that a relation is ***H2*** using substitution.

*Example 12 (**H2**-substitution).*

$okay' \wedge (x' = 0)$ is ***H2***

Proof.

$$
\begin{array}{lr}
\quad (okay' \wedge (x' = 0))^f \Rightarrow (okay' \wedge (x' = 0))^t & \text{substitution} \\
= \mathbf{false} \wedge (x' = 0) \Rightarrow \mathbf{true} \wedge (x' = 0) & \text{propositional calculus} \\
= \mathbf{false} \Rightarrow (x' = 0) & \text{propositional calculus} \\
= \mathbf{true} &
\end{array}
$$

Now we prove that the same relation is ***H2*** using J.

*Example 13 (**H2**-J).*

$okay' \wedge (x' = 0)$ is ***H2***

Proof.

$$
\begin{array}{lr}
\quad okay' \wedge (x' = 0) \,;\, J & J\text{-splitting} \\
= (okay' \wedge (x' = 0))^f \vee ((okay' \wedge (x' = 0))^t \wedge okay') & \text{substitution} \\
= (\mathbf{false} \wedge (x' = 0)) \vee (\mathbf{true} \wedge (x' = 0) \wedge okay') & \text{propositional calculus} \\
= \mathbf{false} \vee ((x' = 0) \wedge okay') & \text{propositional calculus} \\
= okay' \wedge (x' = 0) &
\end{array}
$$

We said at the beginning of this section that we would characterise the space of designs using our healthiness conditions, and we can now do this. If a relation is both ***H1*** and ***H2***-healthy, then it is also a design.

Lemma 12 (*H1*-*H2* is a design). *If P is a relation that is both **H1** and **H2**-healthy, then it can be expressed as the design $\neg\, P^f \vdash P^t$.*

This result also holds in the other direction, as we have already illustrated in the examples. First, we establish that designs are ***H2*** relations.

Lemma 13 (Designs are *H2*). *Provided P and Q do not have okay and okay′ in their alphabets,*

$$[\,(\,P \vdash Q\,)^f \Rightarrow (\,P \vdash Q\,)^t\,]$$

It is obvious that all designs are ***H1***: the proof is a nice little exercise. So, we have the following theorem.

Theorem 4. *A relation with alphabet including okay and okay′ is a design exactly when it is both **H1** and **H2**-healthy.*

An important property of healthiness conditions is commutativity. For example, ***H1*** and ***H2*** commute.

Law 75 (commutativity-*H2*-*H1*). $\boldsymbol{H2} \circ \boldsymbol{H1} = \boldsymbol{H1} \circ \boldsymbol{H2}$

Proof.

$$\begin{array}{lr}
\quad \boldsymbol{H1} \circ \boldsymbol{H2}(P) & \boldsymbol{H1}, \boldsymbol{H2} \\
= \mathit{okay} \Rightarrow P \,;\, J & \text{propositional calculus} \\
= \neg\, \mathit{okay} \;\vee\; P \,;\, J & \text{miracle is } \boldsymbol{H2} \\
= \boldsymbol{H2}(\neg\, \mathit{okay}) \;\vee\; P \,;\, J & \boldsymbol{H2} \\
= \neg\, \mathit{okay} \,;\, J \;\vee\; P \,;\, J & \text{relational calculus} \\
= (\neg\, \mathit{okay} \vee P) \,;\, J & \text{propositional calculus} \\
= (\mathit{okay} \Rightarrow P) \,;\, J & \boldsymbol{H1}, \boldsymbol{H2} \\
= \boldsymbol{H2} \circ \boldsymbol{H1}(P)
\end{array}$$

This means that we can apply ***H1*** and ***H2*** independently to make a relation healthy. The result is a relation that is both ***H1*** and ***H2***-healthy, and, moreover, it is the same no matter in which order we applied ***H1*** and ***H2***.

5 Reactive Processes

A reactive program interacts with its environment, which can include other programs as well as the users of the system. A reactive program's behaviour cannot be characterised by its final state alone; we need to record information about interactions with the environment. Actually, many reactive programs never terminate, and so do not even have a final state; their whole purpose is to interact with the environment. Each interaction, whether it be a synchronisation or a communication, is an event.

To model a reactive process, we use the *okay* variable and three extra observational variables: *tr*, *ref*, and *wait*, and their dashed counterparts. The finite sequences *tr* and *tr′* record the events that occurred up to the moment of the observation. The sets *ref* and *ref′* record events that may be refused. The variables

wait and $wait'$ are boolean; $wait'$ records whether the process has terminated or is in an intermediate state awaiting further interaction with the environment.

When $okay'$ is true for a design, it means that the design has reached a final state. The same is true for a reactive process that has $okay'$ true and $wait'$ false. If both $okay'$ and $wait'$ are true, then it means that the reactive process has reached an intermediate state. If $okay'$ is false, then it means that the process has reached neither an intermediate nor a final state. So, the meaning of $okay'$ is the same in both theories: when it is true, it indicates that a stable state has been reached; when it is false it indicates the opposite. The difference is that the notion of a stable state is richer for reactive processes, as it includes intermediate states. In view of this, we change our terminology: instead of saying that a design has *aborted*, we say that a process has *diverged*.

Of course, these comments apply to *okay* as well. When it is true, it means that the process is in a stable state. This, however, may be an intermediate of another process that is currently executing. The process only really starts when *wait* is false.

In summary, there are three distinct observations that may be made of *okay* and *wait*.

$okay \wedge \neg\, wait$	started in a stable state
$okay \wedge wait$	not started, but in a stable state
$\neg\, okay$	not started, but in an unstable state

Similarly, there are three observations that may be made of the final values of these two variables.

$okay' \wedge \neg\, wait'$	terminated
$okay' \wedge wait'$	in an intermediate state
$\neg\, okay'$	in an unstable state

With these observations, it is clear that a reactive process is properly started if it is initiated in a state with *wait* false; that is, if its predecessor has terminated.

We often want to refer to a predicate $P[false/wait]$, which we abbreviate as P_f. Combining this with our earlier notation, P_f^t describes a reactive process P that was properly started, and has not diverged. This substitution does not disturb healthiness conditions that do not mention *wait* and $okay'$, such as ***H1***.

Law 76 (*H1*-*wait*-$okay'$). $(\boldsymbol{H1}(P))_b^c \;=\; \boldsymbol{H1}(P_b^c)$

Not every relation is a reactive process; just like designs, some healthiness conditions need to be imposed. Before we investigate them, however, we give a simple example of a reactive process.

5.1 Reactive *II*

A reactive process is a relation with all eight observational variables in its alphabet. Perhaps the simplest example is the reactive II, which is defined as follows.

Definition 2 (II_{rea}).

$$\begin{array}{rl} \mathit{II}_{rea} \mathrel{\widehat{=}} & \neg\, okay \land tr \leq tr' \\ & \lor \\ & okay' \land tr' = tr \land wait' = wait \land ref' = ref \end{array}$$

The behaviour of II_{rea} depends on its initial state: if it was an unstable state ($\neg\, okay$), then the first disjunct applies; otherwise, the second disjunct applies. In the first case, the predicate $tr \leq tr'$ requires that tr is a prefix of tr'. The trace tr contains a record of all the events that occurred before the initial observation of II_{rea}; in the final observation, the trace tr' must be an extension of tr: the process cannot change history by modifying the sequence of events that have already occurred. In the second case, the initial state was stable, and the behaviour is the same, regardless of whether the process was started or not: all variables must remain constant.

Alternative definitions of II_{rea} can be formulated. For example, it can be defined in terms of the relational II and in terms of the conditional.

Law 77 (II_{rea}-II_{rel}). $\mathit{II}_{rea} = \neg\, okay \land tr \leq tr' \ \lor\ \mathit{II}_{rel}$

Law 78 (II_{rea}-II_{rel}-conditional). $\mathit{II}_{rea} = \mathit{II}_{rel} \lhd okay \rhd tr \leq tr'$

The law below states that in a stable state, II_{rea} is just like II_{rel}.

Law 79 ($okay$-II_{rea}-II_{rel}). $okay \land \mathit{II}_{rea} = okay \land \mathit{II}_{rel}$

As an obvious consequence, II_{rea} is a unit for sequence in a stable state. Of course, in general it is not an identity, since in an unstable state it guarantees only that the trace is either left untouched or extended.

Law 80 ($okay$-II_{rea}-sequence-unit). $okay \land \mathit{II}_{rea}\ ;\ P = okay \land P$

Exercise 9. Prove Law 80.

Is II_{rea} a design? Well, it is certainly ***H2***-healthy.

Law 81 (II_{rea}-*H2*). $\mathit{II}_{rea} = \boldsymbol{H2}(\mathit{II}_{rea})$

Proof.

$$\begin{array}{rl} & \boldsymbol{H2}(\mathit{II}_{rea}) & J\text{-splitting} \\ = & \mathit{II}_{rea}{}^{f} \lor (\mathit{II}_{rea}{}^{t} \land okay') & \mathit{II}_{rea} \\ = & (\neg\, okay \land tr \leq tr') \lor (\neg\, okay \land tr \leq tr' \ \lor\ \mathit{II}_{rel}) \land okay' & \text{propositional calculus} \\ = & (\neg\, okay \land tr \leq tr') \ \lor\ (\neg\, okay \land tr \leq tr' \land okay') \lor \mathit{II}_{rel}{}^{t} \land okay' & \text{absorption} \\ = & \neg\, okay \land tr \leq tr' \ \lor\ \mathit{II}_{rel}{}^{t} \land okay' & \text{Leibniz} \\ = & \neg\, okay \land tr \leq tr' \ \lor\ okay' \land \mathit{II}_{rel} & \mathit{II}_{rea} \\ = & \mathit{II}_{rea} \end{array}$$

Although, being $\boldsymbol{H2}$, II_{rea} is half-way to being a design, it is not $\boldsymbol{H1}$-healthy. This is because its behaviour when *okay* is false is not arbitrary as $\boldsymbol{H1}$ requires: the restriction on the traces still applies. In fact, the healthiness condition $\boldsymbol{H1}$ relates the two identities in the following way.

Law 82 ($\boldsymbol{H1}$-II_{rea}-II_{rel}). $\boldsymbol{H1}(\mathit{II}_{rea}) = \boldsymbol{H1}(\mathit{II}_{rel})$

So II_{rea} fails to be a design; in fact, *no* reactive process is a design, although as we shall see, they can all be expressed in terms of a design. So, the theory of reactive processes is a subtheory of the theory of relations that is distinct from the theory of designs. The question is: which relations are reactive processes? This answered by three healthiness conditions.

5.2 *R1*

Time travel is practically a weekly event in *Star Trek* and *Doctor Who*, and it is certainly an interesting activity that has much to offer the curious mind, but it will be outlawed by our first reactive healthiness condition, $\boldsymbol{R1}$. This requires that a relation cannot change the trace of events that have already occurred. The idempotent is as follows.

$$\boldsymbol{R1}(P) = P \wedge tr \leq tr'$$

We already saw this in the definition of II_{rea}: if its initial observation is made in an unstable state (*okay* is false), then the trace in its final observation will be an extension of the initial trace; if the initial observation is made in a stable state, then the trace is kept constant. We have that $\boldsymbol{R1}$ is an idempotent because of idempotency of conjunction.

Law 83 (R1-idempotent). $\boldsymbol{R1} \circ \boldsymbol{R1} = \boldsymbol{R1}$

The simplicity of $\boldsymbol{R1}$ leads to many obvious algebraic laws. For example, it distributes through both conjunction and disjunction, and because it is defined by conjunction, its scope may be extended over other conjunctions. As a consequence of these laws, $\boldsymbol{R1}$ distributes through the conditional and the unhealthy effects of negation can be swiftly cured. Finally, substitution for *wait* and for $okay'$ both distribute through $\boldsymbol{R1}$.

Law 84 ($\boldsymbol{R1}$-$\wedge$). $\boldsymbol{R1}(P \wedge Q) = \boldsymbol{R1}(P) \wedge \boldsymbol{R1}(Q)$

Law 85 ($\boldsymbol{R1}$-$\vee$). $\boldsymbol{R1}(P \vee Q) = \boldsymbol{R1}(P) \vee \boldsymbol{R1}(Q)$

Law 86 ($\boldsymbol{R1}$-extend-$\wedge$). $\boldsymbol{R1}(P) \wedge Q = \boldsymbol{R1}(P \wedge Q)$

Law 87 ($\boldsymbol{R1}$-conditional). $\boldsymbol{R1}(P \lhd b \rhd Q) = \boldsymbol{R1}(P) \lhd b \rhd \boldsymbol{R1}(Q)$

Law 88 ($\boldsymbol{R1}$-negate-$\boldsymbol{R1}$). $\boldsymbol{R1}(\neg\, \boldsymbol{R1}(P)) = \boldsymbol{R1}(\neg\, P)$

Law 89 ($\boldsymbol{R1}$-*wait*-$okay'$). $(\boldsymbol{R1}(P))^c_b = \boldsymbol{R1}(P^c_b)$

Both the relational and the reactive identities are ***R1***-healthy.

Law 90 (II_{rel}-*R1*). $\mathit{II}_{rel} = \boldsymbol{R1}(\mathit{II}_{rel})$

Exercise 10. Prove Law 90.

The fact that II_{rel} is ***R1*** is helpful in proving that II_{rea} is too.

Law 91 (II_{rea}-*R1*). $\mathit{II}_{rea} = \boldsymbol{R1}(\mathit{II}_{rea})$

Proof.

$$
\begin{array}{lr}
\quad \boldsymbol{R1}(\mathit{II}_{rea}) & \mathit{II}_{rea}\text{-conditional} \\
= \boldsymbol{R1}(\mathit{II}_{rel} \lhd \mathit{okay} \rhd \mathit{tr} \leq \mathit{tr}') & \boldsymbol{R1}\text{-conditional} \\
= \boldsymbol{R1}(\mathit{II}_{rel}) \lhd \mathit{okay} \rhd \boldsymbol{R1}(\mathbf{true}) & \text{propositional calculus, } \boldsymbol{R1} \\
= \boldsymbol{R1}(\mathit{II}_{rel}) \lhd \mathit{okay} \rhd \mathit{tr} \leq \mathit{tr}' & \mathit{II}_{rel}\text{-}\boldsymbol{R1} \\
= \mathit{II}_{rel} \lhd \mathit{okay} \rhd \mathit{tr} \leq \mathit{tr}' & \mathit{II}_{rea} \text{ conditional} \\
= \mathit{II}_{rea}
\end{array}
$$

By applying any of the program operators to an ***R1***-healthy process, we get another ***R1***-healthy process. That is, ***R1*** is closed under conjunction, disjunction, conditional, and sequence.

Theorem 5. *Provided P and Q are* ***R1****-healthy,*

$$
\begin{array}{ll}
\boldsymbol{R1}(P \wedge Q) = P \wedge Q & \boldsymbol{R1}\text{-}\wedge\text{-}\mathit{closure} \\
\boldsymbol{R1}(P \vee Q) = P \vee Q & \boldsymbol{R1}\text{-}\vee\text{-}\mathit{closure} \\
\boldsymbol{R1}(P \lhd b \rhd Q) = P \lhd b \rhd Q & \boldsymbol{R1}\text{-}\mathit{conditional\text{-}closure} \\
\boldsymbol{R1}(P \,;\, Q) = P \,;\, Q & \boldsymbol{R1}\text{-}\mathit{sequence\text{-}closure}
\end{array}
$$

The distribution properties are stronger than the closure properties, and this is clear from the fact that the proofs of ***R1***-$\wedge$-*closure*, ***R1***-$\vee$-*closure*, and ***R1***-*conditional-closure* follow immediately from Laws 84, 85, and 87, respectively. Law ***R1***-*sequence-closure* is rather more interesting, since ***R1*** does not distribute through sequence. First we note a result from the relational calculus that relates sequence and transitive relations.

Lemma 14 (sequence-transitive-relation). *Provided $\preceq$ is a transitive relation,*

$$[\,((P \wedge x \preceq x') \,;\, (Q \wedge x \preceq x')) \Rightarrow x \preceq x'\,]$$

This establishes that if the first program in the sequence assigns to x a value that is related to its original value by the transitive relation $\preceq$, and the second program takes this (intermediate) value and assigns to x a value that is related to it, then transitivity allows us to conclude that the original and final values of x are related by $\preceq$.

Exercise 11. Prove Lemma 14.

In our case, the transitive relation in which we are interested in sequence prefixing.

Proof of ***R1****-sequence-closure.*

$$
\begin{array}{ll}
& P \,;\, Q & \text{assumption: } P \text{ and } Q \text{ both } \boldsymbol{R1} \\
= & \boldsymbol{R1}(P) \,;\, \boldsymbol{R1}(Q) & \boldsymbol{R1}\text{, twice} \\
= & P \wedge tr \leq tr' \,;\, Q \wedge tr \leq tr' & \text{sequence-transitive-relation} \\
= & (P \wedge tr \leq tr' \,;\, Q \wedge tr \leq tr') \wedge tr \leq tr' & \boldsymbol{R1}\text{, three times} \\
= & \boldsymbol{R1}(\boldsymbol{R1}(P) \,;\, \boldsymbol{R1}(Q)) & \text{assumption: } P \text{ and } Q \text{ both } \boldsymbol{R1} \\
= & \boldsymbol{R1}(P \,;\, Q)
\end{array}
$$

$\mathbb{I}_{rea}$ is an ***R1*** relation, but as we have seen it is not ***H1***. Of course, we can make it ***H1*** by applying the healthiness condition, but then it is no longer ***R1***. If we apply ***R1*** once again, we get back to where we started.

Law 92 ($\mathbb{I}_{rea}$**-*R1*-*H1*).** $\mathbb{I}_{rea} = \boldsymbol{R1} \circ \boldsymbol{H1}(\mathbb{I}_{rea})$

Proof.

$$
\begin{array}{ll}
& \boldsymbol{R1} \circ \boldsymbol{H1}(\mathbb{I}_{rea}) & \boldsymbol{H1}\text{-}\mathbb{I}_{rea}\text{-}\mathbb{I}_{rel} \\
= & \boldsymbol{R1} \circ \boldsymbol{H1}(\mathbb{I}_{rel}) & \boldsymbol{H1} \\
= & \boldsymbol{R1}(\mathit{okay} \Rightarrow \mathbb{I}_{rel}) & \text{propositional calculus} \\
= & \boldsymbol{R1}(\neg\, \mathit{okay} \vee \mathbb{I}_{rel}) & \boldsymbol{R1}\text{-}\vee\text{-distribution} \\
= & \boldsymbol{R1}(\neg\, \mathit{okay}) \vee \boldsymbol{R1}(\mathbb{I}_{rel}) & \mathbb{I}_{rel}\text{-}\boldsymbol{R1} \\
= & \boldsymbol{R1}(\neg\, \mathit{okay}) \vee \mathbb{I}_{rel} & \boldsymbol{R1} \\
= & \neg\, \mathit{okay} \wedge tr \leq tr' \vee \mathbb{I}_{rel} & \mathbb{I}_{rea} \\
= & \mathbb{I}_{rea}
\end{array}
$$

Law 92 shows that ***R1*** and ***H1*** are not independent: they do not commute; but ***R1*** does commute with ***H2***.

Law 93 (*R1*-*H2*-commutativity). $\boldsymbol{R1} \circ \boldsymbol{H2} = \boldsymbol{H2} \circ \boldsymbol{R1}$

Proof.

$$
\begin{array}{ll}
& \boldsymbol{H2} \circ \boldsymbol{R1}(P) & \boldsymbol{R1} \\
= & \boldsymbol{H2}(P \wedge tr \leq tr') & \boldsymbol{H2}\text{-}\wedge\text{-non-}\mathit{okay} \\
= & \boldsymbol{H2}(P) \wedge tr \leq tr' & \boldsymbol{R1} \\
= & \boldsymbol{R1} \circ \boldsymbol{H2}(P)
\end{array}
$$

The space described by applying ***R1*** to designs is a complete lattice because ***R1*** is monotonic. The relevance of this fact is made clear in the next section.

5.3 *R2*

The trace of a reactive process is an observation that is useful in describing the behaviour of concurrency and communication in reactive systems. We do

not imagine that any programmer would want to include such a variable in a real program; the overhead of keeping an accurate record of all events that have occurred since a program was started would be huge, and there is no need to keep it anyway. Rather, tr and tr' play similar roles to *okay* and *okay*$'$: they are devices that allow us to give an account of the behaviour of programming language constructs.

Designs are sensitive to the initial value of *okay*: the design cannot be started unless *okay* is true. But there is no obvious reason why a reactive process should be sensitive to the initial value of tr; in fact, none of the programming language constructs that we will introduce are sensitive to its value. The only purpose given to the trace is to provide an abstract view of program behaviour. For these reasons, we introduce a second healthiness condition that requires reactive processes to be insensitive to the value of tr.

There are two alternative formulations for this healthiness condition. Intuitively, they each establish that a process description should not rely on the history that passed before its activation, and should restrict only the new events to be recorded since the last observation. These are the events in $tr' - tr$.

The first formulation requires that P is not changed if tr is replaced by an arbitrary value. Of course, if tr is changed, then a corresponding change must be made to tr', otherwise all chance of ***R1*** healthiness will be compromised.

$$\boldsymbol{R2a}(P(tr, tr')) = \sqcap_s \bullet P(s, s \frown (tr' - tr))$$

The second formulation requires that P is not changed if the value of tr is taken to be the empty sequence.

$$\boldsymbol{R2b}(P(tr, tr')) = P(\langle\rangle, tr' - tr)$$

R2a and ***R2b*** are different functions. To see this, compare what happens when each function is applied to the relation $tr = \langle a \rangle$.

$$\begin{array}{ll} & \boldsymbol{R2a}(tr = \langle a \rangle) \\ = & \sqcap s \bullet s = \langle a \rangle \\ = & \mathbf{true} \sqcap \mathbf{false} \\ = & \mathbf{true} \end{array} \qquad \begin{array}{ll} & \boldsymbol{R2b}(tr = \langle a \rangle) \\ = & (tr = \langle a \rangle)[\langle\rangle, tr' - tr / tr, tr'] \\ = & (\langle\rangle = \langle a \rangle) \\ = & \mathbf{false} \end{array}$$

Even though they are different functions, they do have much in common. First, every ***R2b***-healthy relation is also ***R2a***-healthy; that is, for every relation P, $\boldsymbol{R2b}(P)$ is a fixed point of ***R2a***.

Law 94 (*R2b*-*R2a*). $\boldsymbol{R2b} = \boldsymbol{R2a} \circ \boldsymbol{R2b}$

Proof.

$$\begin{array}{ll} & \boldsymbol{R2a} \circ \boldsymbol{R2b}(P(tr, tr')) & \boldsymbol{R2b} \\ = & \boldsymbol{R2a}(P(\langle\,\rangle, tr' - tr)) & \boldsymbol{R2a} \end{array}$$

$$
\begin{array}{ll}
= \sqcap s \bullet P(\langle\rangle, tr' - tr)(s, s \frown (tr' - tr)) & \text{substitution} \\
= \sqcap s \bullet P(\langle\rangle, s \frown (tr' - tr) - s) & \text{property of } - \\
= \sqcap s \bullet P(\langle\rangle, tr' - tr) & \text{property of } \sqcap \\
= P(\langle\rangle, tr' - tr) & \boldsymbol{R2b} \\
= \boldsymbol{R2b}(P) &
\end{array}
$$

Similarly, every ***R2a***-healthy relation is also ***R2b***-healthy; that is, for every relation P, $\boldsymbol{R2a}(P)$ is a fixed point of ***R2b***.

Law 95 (*R2a*-*R2b*). $\boldsymbol{R2a} = \boldsymbol{R2b} \circ \boldsymbol{R2a}$

Proof.

$$
\begin{array}{ll}
\boldsymbol{R2b} \circ \boldsymbol{R2a}(P(tr, tr')) & \boldsymbol{R2a} \\
= \boldsymbol{R2b}(\sqcap s \bullet P(s, s \frown (tr' - tr))) & \boldsymbol{R2b} \\
= (\sqcap s \bullet P(s, s \frown (tr' - tr)))(\langle\rangle, tr' - tr) & \text{substitution} \\
= \sqcap s \bullet P(s, s \frown (tr' - tr) - \langle\rangle) & \text{property of } - \\
= \sqcap s \bullet P(s, s \frown (tr' - tr)) & \boldsymbol{R2a} \\
= \boldsymbol{R2a}(P) &
\end{array}
$$

Laws 94 and 95 show us that ***R2a*** and ***R2b*** have the same image; that is, they characterise the same set of healthy predicates. We adopt ***R2b*** as our second healthiness condition for reactive processes, and actually refer to it as ***R2***.

$$\boldsymbol{R2} = \boldsymbol{R2b}$$

Not all properties of ***R2b*** that we prove in the sequel hold for ***R2a***; so this is an important point.

The healthiness condition ***R2*** is an idempotent.

Law 96 (R2-idempotent). $\boldsymbol{R2} \circ \boldsymbol{R2} = \boldsymbol{R2}$

Again, the programming operators are closed with respect to ***R2***. For the conditional, we have a result for quite specific conditions. For brevity, we omit proofs.

Theorem 6. *Provided P and Q are* ***R2****-healthy, and tr and tr' are not in the alphabet of b,*

$$
\begin{array}{ll}
\boldsymbol{R2}(P \wedge Q) = P \wedge Q & \boldsymbol{R1}\text{-}\wedge\text{-closure} \\
\boldsymbol{R2}(P \vee Q) = P \vee Q & \boldsymbol{R1}\text{-}\vee\text{-closure} \\
\boldsymbol{R2}(P \lhd tr' = tr \rhd Q) = P \lhd tr' = tr \rhd Q & \boldsymbol{R2}\text{-conditional-closure-1} \\
\boldsymbol{R2}(P \lhd b \rhd Q) = P \lhd b \rhd Q & \boldsymbol{R2}\text{-conditional-closure-2} \\
\boldsymbol{R2}(P \,;\, Q) = P \,;\, Q & \boldsymbol{R2}\text{-sequence-closure}
\end{array}
$$

Conditionals whose condition involves tr or tr' are problematic, but as shown above, the particular condition $tr = tr'$ does not hamper distribution.

Our proof of Law ***R2*-sequence-closure** is based on a suggestion due to Chen Yifeng. ***R2*** does not distribute through the sequence $P \,;\, Q$ because it cannot constrain the hidden value of the trace that exists between the behaviours of P and Q. For example, we consider the sequence below.

$$tr' = tr \frown \langle\, a \,\rangle;\ \text{last}\ \ tr = a \wedge tr \leq tr'$$

It is an ***R2*** process, and so it is not changed by an application of ***R2***. The second process, however, is not ***R2*** as it relies on a particular property of the initial value of tr; namely, that its last element is a. If we apply ***R2*** to it, we get **false** as a result. Therefore,

$$\boldsymbol{R2}(tr' = tr \frown \langle\, a \,\rangle);\ \boldsymbol{R2}(\text{last}\ \ tr = a \wedge tr \leq tr')$$

is also **false**.

Exercise 12. Give an algebraic proof that the sequence above is ***R2***-healthy, or, in other words, $\boldsymbol{R2}(tr' = tr \frown \langle\, a \,\rangle;\ \text{last}\ \ tr = a \wedge tr \leq tr')$ is equal to $tr' = tr \frown \langle\, a \,\rangle;\ \text{last}\ \ tr = a \wedge tr \leq tr'$ itself, and that

$$\boldsymbol{R2}(tr' = tr \frown \langle\, a \,\rangle);\ \boldsymbol{R2}(\text{last}\ \ tr = a \wedge tr \leq tr') = \mathbf{false}$$

The proof below for ***R2*-sequence-closure** is based on the Laws *left-one-point* and *right-one-point* for sequences.

Proof of ***R2*-sequence-closure**.

$$\begin{array}{lr}
\quad \boldsymbol{R2}(P(tr, tr') \,;\, Q(tr, tr')) & \text{sequence, predicate calculus} \\
= \boldsymbol{R2}(\, P(tr, tr'_0) \,;\, Q(tr_0, tr')\,) & \boldsymbol{R2} \\
= (\, P(tr, tr'_0) \,;\, Q(tr_0, tr')\,)(\langle\rangle, tr' - tr) & \text{substitution} \\
= P(\langle\rangle, tr'_0) \,;\, Q(tr_0, tr' - tr) & \text{assumption: } Q \text{ is } \boldsymbol{R2} \\
= P(\langle\rangle, tr'_0) \,;\, Q(\langle\rangle, tr' - tr)(tr_0, tr' - tr) & \text{substitution} \\
= P(\langle\rangle, tr'_0) \,;\, Q(\langle\rangle, tr' - tr - tr_0) & \text{sequence property} \\
= P(\langle\rangle, tr'_0) \,;\, Q(\langle\rangle, tr' - (tr \frown tr_0)) & \text{substitution} \\
= P(\langle\rangle, tr'_0) \,;\, Q(\langle\rangle, tr' - tr)[tr \frown tr_0/tr] & \text{left-one-point} \\
= P(\langle\rangle, tr'_0) \,;\, tr' = tr \frown tr_0 \,;\, Q(\langle\rangle, tr' - tr) & \text{sequence property} \\
= P(\langle\rangle, tr'_0) \,;\, tr_0 = tr' - tr \,;\, Q(\langle\rangle, tr' - tr) & \text{right-one-point} \\
= P(\langle\rangle, tr' - tr) \,;\, Q(\langle\rangle, tr' - tr) & \text{assumption: } P \text{ and } Q \text{ are } \boldsymbol{R2} \\
= P \,;\, Q &
\end{array}$$

A by-product of the above proof is the following law.

Law 97 (*R2* composition). $\boldsymbol{R2}(P \,;\, \boldsymbol{R2}(Q)) \;=\; \boldsymbol{R2}(P) \,;\, \boldsymbol{R2}(Q)$

Since ***R2*** constrains only tr and tr', substitution for $wait$ and $okay'$ distribute through its application.

Law 98 (*R2-wait-okay′*). $(\boldsymbol{R2}(P))^c_b = \boldsymbol{R2}(P^c_b)$

If J (see Definition 1) is lifted to an alphabet containing the reactive observations, then it keeps the trace constant. It is therefore ***R2***-healthy.

Law 99 (*J-R2*). $J = \boldsymbol{R2}(J)$

R2 is independent from ***H1***, ***H2***, and ***R1***: it commutes with each of them.

Law 100 (commutativity-*R2-H1*). $\boldsymbol{R2} \circ \boldsymbol{H1} = \boldsymbol{H1} \circ \boldsymbol{R2}$

Law 101 (commutativity-*R2-H2*). $\boldsymbol{R2} \circ \boldsymbol{H2} = \boldsymbol{H2} \circ \boldsymbol{R2}$

Proof.

$$
\begin{array}{lr}
\quad \boldsymbol{R2} \circ \boldsymbol{H2}(P) & \boldsymbol{H2} \\
= \boldsymbol{R2}(P \,;\, J) & J\ \boldsymbol{R2} \\
= \boldsymbol{R2}(P \,;\, \boldsymbol{R2}(J)) & \boldsymbol{R2}\ \text{composition} \\
= \boldsymbol{R2}(P) \,;\, \boldsymbol{R2}(J) & J\ \boldsymbol{R2} \\
= \boldsymbol{R2}(P) \,;\, J & \boldsymbol{H2} \\
= \boldsymbol{H2} \circ \boldsymbol{R2}(P)
\end{array}
$$

Law 102 (commutativity-*R2-R1*). $\boldsymbol{R2} \circ \boldsymbol{R1} = \boldsymbol{R1} \circ \boldsymbol{R2}$

Proof.

$$
\begin{array}{lr}
\quad \boldsymbol{R2} \circ \boldsymbol{R1}(P(tr, tr')) & \boldsymbol{R1}, \boldsymbol{R2} \\
= (P \wedge tr \leq tr')(\langle\rangle, tr' - tr) & \text{substitution} \\
= P(\langle\rangle, tr' - tr) \wedge \langle\rangle \leq tr' - tr & \leq \text{and} - \\
= P(\langle\rangle, tr' - tr) \wedge tr \leq tr' & \boldsymbol{R1}, \boldsymbol{R2} \\
= \boldsymbol{R1} \circ \boldsymbol{R2}(P(tr, tr'))
\end{array}
$$

The space of relations produced by applying ***R2*** to designs is again a complete lattice, since ***R2*** is also monotonic.

5.4 *R3*

The third healthiness condition makes relational composition behave like a program sequence. To see its relevance, consider the process

$$P \;=\; okay' \wedge wait' \wedge tr' = tr$$

P forever occupies a state that is both stable and waiting for interaction with the environment, but none ever comes, since the trace never changes. What are we to make of the sequence $P \,;\, Q$, where

$$Q \;=\; okay' \wedge \neg\, wait' \wedge tr' = tr \frown \langle a \rangle$$

Q immediately terminates, having added an a event to the trace? The relational composition ignores the behaviour of P.

$$
\begin{array}{ll}
P \,;\, Q & \text{sequence} \\
= \exists\, okay_0, wait_0, tr_0, ref_0 \bullet & P \text{ and } Q \\
\quad P[okay_0, wait_0, tr_0, ref_0 / okay', wait', tr', ref'] \wedge & \\
\quad Q[okay_0, wait_0, tr_0, ref_0 / okay, wait, tr, ref] & \\
= \exists\, okay_0, wait_0, tr_0, ref_0 \bullet & \text{predicate calculus} \\
\quad okay_0 \wedge wait_0 \wedge tr_0 = tr \wedge & \\
\quad okay' \wedge \neg\, wait' \wedge tr' = tr_0 \frown \langle a \rangle & \\
= \exists\, ref_0 \bullet okay' \wedge \neg\, wait' \wedge tr' = tr \frown \langle a \rangle & \text{predicate calculus} \\
= okay' \wedge \neg\, wait' \wedge tr' = tr \frown \langle a \rangle & Q \\
= Q &
\end{array}
$$

We expect quite the opposite: that $P \,;\, Q = P$. If P is forever waiting, then $P \,;\, Q$ should be forever waiting *in the same state*. We formalise this requirement as a healthiness condition, ***R3***.

$$\boldsymbol{R3}(P) = (\mathbb{I}_{rea} \lhd wait \rhd P)$$

An ***R3***-healthy process does not start until its predecessor has terminated. Now we have that $P \,;\, \boldsymbol{R3}(Q) = P$, as we wanted.

Exercise 13. Prove that $P \,;\, \boldsymbol{R3}(Q) = P$, where P and Q are the processes defined above.

The following laws characterise the behaviour of ***R3*** processes in particular circumstances. ***R3*** depends on the *wait* observation, so substitution for that variable cannot distribute through the healthiness condition. Instead, it serves to simplify ***R3***'s conditional. If *true* is substituted, then the result is $\mathbb{I}_{rea}$, but with the substitution applied to that as well. On the other hand, if *false* is substituted for *wait* in $\boldsymbol{R3}(P)$, then the result is simply P, again with the substitution applied. Substitution for $okay'$ interferes with $\mathbb{I}_{rea}$, and so does not distribute through its application.

Law 103 (*R3*-*wait*-true). $(\boldsymbol{R3}(P))_t = (\mathbb{I}_{rea})_t$

Law 104 (*R3*-not-*wait*-false). $(\boldsymbol{R3}(P))_f = P_f$

Law 105 (*R3*-$okay'$). $(\boldsymbol{R3}(P))^c = ((\mathbb{I}_{rea})^c \lhd wait \rhd P^c)$

Closure properties are also available for ***R3***.

Theorem 7. *Provided P and Q are* ***R3****,*

$$
\begin{array}{ll}
\boldsymbol{R3}(P \wedge Q) = P \wedge Q & \textit{R3-}\wedge\textit{-closure} \\
\boldsymbol{R3}(P \vee Q) = P \vee Q & \textit{R3-}\vee\textit{-closure} \\
\boldsymbol{R3}(P \lhd c \rhd Q) = P \lhd c \rhd Q & \textit{R3-conditional-closure}
\end{array}
$$

For sequence, we actually require that one of the processes is ***R1*** as well.

Theorem 8. *Provided P is **R3**, and Q is **R1** and **R3**,*

$$\boldsymbol{R3}(P \mathbin{;} Q) = P \mathbin{;} Q \qquad \textit{R3-sequence-closure}$$

This is not a problem because, as detailed in the next section, we actually work with the theory characterised by all healthiness conditions.

As required, ***R3*** is an idempotent.

Law 106 (R3-idempotent). $\boldsymbol{R3} \circ \boldsymbol{R3} = \boldsymbol{R3}$

Since $\mathbb{I}_{rea}$ specifies behaviour for when $\neg$ *okay* holds, it should not be a big surprise that ***R3*** also does not commute with ***H1***. It does commute with the other healthiness conditions, though.

Law 107 (commutativity-*R3*-*H2*). $\boldsymbol{R3} \circ \boldsymbol{H2} = \boldsymbol{H2} \circ \boldsymbol{R3}$

Law 108 (commutativity-*R3*-*R1*). $\boldsymbol{R3} \circ \boldsymbol{R1} = \boldsymbol{R1} \circ \boldsymbol{R3}$

Law 109 (commutativity-*R3*-*R2*). $\boldsymbol{R3} \circ \boldsymbol{R2} = \boldsymbol{R2} \circ \boldsymbol{R3}$

Moreover, if all that there is about a process that is not ***H1*** is the fact that it specifies the behaviour required by ***R1***, then we have a commutativity property. This sort of property is important because we are going to express reactive processes as reactive designs.

*Example 14 (**R3**-**H1**-non-commutativity).* Why do ***R3*** and ***H1*** not commute?

$$\begin{array}{lr}
\quad \boldsymbol{H1} \circ \boldsymbol{R3}(P) & \boldsymbol{H1}, \boldsymbol{R3} \\
= \textit{okay} \Rightarrow (\mathbb{I}_{rea} \lhd \textit{wait} \rhd P) & \odot\text{-conditional} \\
= (\textit{okay} \Rightarrow \mathbb{I}_{rea}) \lhd \textit{wait} \rhd (\textit{okay} \Rightarrow P) & \boldsymbol{H1} \\
= \boldsymbol{H1}(\mathbb{I}_{rea}) \lhd \textit{wait} \rhd \boldsymbol{H1}(P) & \mathbb{I}_{rea} \text{ is not } \boldsymbol{H1} \\
\neq \mathbb{I}_{rea} \lhd \textit{wait} \rhd \boldsymbol{H1}(P) & \boldsymbol{R3} \\
= \boldsymbol{R3} \circ \boldsymbol{H1}(P) &
\end{array}$$

This derivation shows the precise reason: it is because $\mathbb{I}_{rea}$ is not ***H1***.

This last example explains the need for the weaker commutativity laws below.

Law 110 (*R3*-*H1* sub-commutativity). $\boldsymbol{H1} \circ \boldsymbol{R3} = \boldsymbol{H1} \circ \boldsymbol{R3} \circ \boldsymbol{H1}$

Law 111 (*R3*-*H1*-*R1* sub-commutativity).

$$\boldsymbol{R3} \circ \boldsymbol{R1} \circ \boldsymbol{H1} = \boldsymbol{R1} \circ \boldsymbol{H1} \circ \boldsymbol{R3}$$

Just like ***R1*** and ***R2***, ***R3*** is monotonic and so gives us a complete lattice when applied to the space of designs.

5.5 $\boldsymbol{R}$

A reactive process is a relation that includes in its alphabet *okay*, *tr*, *wait*, and *ref*, and their dashed counterparts, and satisfies the three healthiness conditions $\boldsymbol{R1}$, $\boldsymbol{R2}$, and $\boldsymbol{R3}$. We define $\boldsymbol{R}$ as the composition of these three functions.

$$\boldsymbol{R} \mathrel{\widehat{=}} \boldsymbol{R1} \circ \boldsymbol{R2} \circ \boldsymbol{R3}$$

Since each of the healthiness conditions $\boldsymbol{R1}$, $\boldsymbol{R2}$, and $\boldsymbol{R3}$ commute, their order in the definition above is irrelevant.

Reactive processes have a left zero.

Law 112 (reactive-left-zero). $(tr \leq tr') \mathbin{;} P = tr \leq tr'$

Proof. First, we expand $\boldsymbol{R3}(P)$.

$$\begin{array}{lr}
& \boldsymbol{R3}(P) \\
= & \mathbb{I}_{rea} \lhd wait \rhd P & \mathbb{I}_{rea} \\
= & (\mathbb{I}_{rel} \lhd okay \rhd tr \leq tr') \lhd wait \rhd P & \text{conditional} \\
= & (\neg\, okay \wedge wait \wedge tr \leq tr') \vee (okay \wedge wait \wedge \mathbb{I}_{rel}) \vee (\neg\, wait \wedge P)
\end{array}$$

Now we can prove our result.

$$\begin{array}{lr}
& tr \leq tr';\ P & \text{assumption: } P \text{ is } \boldsymbol{R3} \\
= & tr \leq tr';\ \boldsymbol{R3}(P) & \boldsymbol{R3}(P) \text{ expansion} \\
= & tr \leq tr';\ \neg\, okay \wedge wait \wedge tr \leq tr' & \text{right-one-point, } \mathbb{I}_{rel} \\
& \vee\ tr \leq tr';\ okay \wedge wait \wedge \mathbb{I}_{rel} & \\
& \vee\ tr \leq tr';\ \neg\, wait \wedge P & \\
= & (tr \leq tr';\ tr \leq tr') \vee (tr \leq tr') \vee (tr \leq tr';\ P) & \text{sequence} \\
= & (tr \leq tr') \vee (tr \leq tr';\ \wedge P) & \text{assumption: } P \text{ is } \boldsymbol{R1} \\
= & (tr \leq tr') \vee (tr \leq tr';\ \wedge P \wedge tr \leq tr') & \text{sequence transitivity} \\
= & tr \leq tr' \vee ((tr \leq tr';\ \wedge P \wedge tr \leq tr') \wedge tr \leq tr') & \text{absorption} \\
= & tr \leq tr'
\end{array}$$

Reactive processes also have a restricted identity.

Law 113 (reactive-restricted-identity).

$$\mathbb{I}_{rea} \mathbin{;} P \;=\; P \lhd okay \rhd tr \leq tr'$$

Substitution for *wait* cannot distribute through $\boldsymbol{R}$, since it does not distribute through $\boldsymbol{R3}$; however, it does have the expected simplification properties. Finally, substitution for $okay'$ does not quite distribute through $\boldsymbol{R}$, since it interferes with $\mathbb{I}_{rea}$. The following reductions hold for these substitutions.

Law 114 ($\boldsymbol{R}$-*wait*-false). $(\boldsymbol{R}(P))_f = \boldsymbol{R1} \circ \boldsymbol{R2}(P_f)$

Law 115 (*R-wait*-true). $(\boldsymbol{R}(P))_t = (\mathit{II}_{rea})_t$

Law 116 (*R-okay′*). $(\boldsymbol{R}(P))^c = ((\mathit{II}_{rea})^c \lhd \mathit{wait} \rhd \boldsymbol{R1} \circ \boldsymbol{R2}(P^c))$

The set of reactive processes is closed under the program operators.

Theorem 9. *Provided P and Q are $\boldsymbol{R}$-healthy,*

$$\boldsymbol{R}(P \wedge Q) = P \wedge Q \qquad \textit{R-}\wedge\textit{-closure}$$
$$\boldsymbol{R}(P \vee Q) = P \vee Q \qquad \textit{R-}\vee\textit{-closure}$$
$$\boldsymbol{R}(P \lhd tr' = tr \rhd Q) = P \lhd tr' = tr \rhd Q \qquad \textit{R-conditional-closure}$$
$$\boldsymbol{R}(P \,;\, Q) = P \,;\, Q \qquad \textit{R-sequence-closure}$$

Since ***R1***, ***R2***, and ***R3*** are all monotonic, so is their composition, and so the set of reactive processes is a complete lattice. The ***R***-image of any complete lattice is also a complete lattice. In particular, the ***R***-image of the lattice of designs is a complete lattice. This image turns out to be the set of CSP processes, as we establish in the next section.

6 CSP Processes

A CSP process is a reactive process satisfying two other healthiness conditions.

6.1 *CSP1*

The first healthiness condition requires that, in case of divergence, extension of the trace is the only property that is guaranteed.

$$\boldsymbol{CSP1}(P) \;=\; P \;\vee\; \neg\, \mathit{okay} \wedge tr \leq tr'$$

It is important to observe that ***R1*** requires that, in whatever situation, the trace can only be increased. On the other hand, ***CSP1*** states that, if we are in a divergent state, $\neg\ \mathit{okay}$, then there is no other guarantee.

Exercise 14. Give an example of a reactive process that is ***R1***, but not ***CSP1***.

CSP1 is a combination of ***R1*** and ***H1***; however, like ***R1***, ***CSP1*** does not commute with ***H1***. The reason is the same: it specifies behaviour for when $\neg\ \mathit{okay}$ holds. The lack of commutativity means that, when applying ***R1*** and ***H1***, the order is relevant. As a matter of fact, ***CSP1*** determines the order that should be used, for processes that are already ***R1***.

Law 117 (*CSP1-R1-H1*).

$$\boldsymbol{CSP1}(P) = \boldsymbol{R1} \circ \boldsymbol{H1}(P) \textbf{ provided } P \textit{ is } \boldsymbol{R1}\textit{-healthy}$$

As expected, ***CSP1*** is an idempotent.

$$\boldsymbol{CSP1} \circ \boldsymbol{CSP1} = \boldsymbol{CSP1}$$

The usual closure properties hold for ***CSP1*** processes.

Theorem 10. *Provided P and Q are* ***CSP1****-healthy,*

$$\boldsymbol{CSP1}(P \wedge Q) = P \wedge Q \qquad \textit{CSP1-}\wedge\textit{-closure}$$
$$\boldsymbol{CSP1}(P \vee Q) = P \vee Q \qquad \textit{CSP1-}\vee\textit{-closure}$$
$$\boldsymbol{CSP1}(P \lhd c \rhd Q) = P \lhd c \rhd Q \qquad \textit{CSP1-conditional-closure}$$
$$\boldsymbol{CSP1}(P \,;\, Q) = P \,;\, Q \qquad \textit{CSP1-sequence-closure}$$

This new healthiness condition is independent from the previous ones.

Law 118 (commutativity-*CSP1*-*R1*). $\boldsymbol{CSP1} \circ \boldsymbol{R1} = \boldsymbol{R1} \circ \boldsymbol{CSP1}$

Law 119 (commutativity-*CSP1*-*R2*). $\boldsymbol{CSP1} \circ \boldsymbol{R2} = \boldsymbol{R2} \circ \boldsymbol{CSP1}$

Law 120 (commutativity-*CSP1*-*R3*). $\boldsymbol{CSP1} \circ \boldsymbol{R3} = \boldsymbol{R3} \circ \boldsymbol{CSP1}$

A reactive process defined in terms of a design is always ***CSP1***-healthy. This is because the design does not restrict the behaviour when $\neg$ *okay* holds, and ***R*** insists only that $tr \leq tr'$.

Law 121 (reactive-design-*CSP1*). $\boldsymbol{CSP1}(\boldsymbol{R}(P \vdash Q)) = \boldsymbol{R}(P \vdash Q)$

If an ***R1***-healthy predicate R appears in a design's postcondition, in the scope of another predicate that is also ***R1***, then R is ***CSP1***-healthy. This is because, for ***R1*** predicates, ***CSP1*** amounts to the composition of ***H1*** and ***R1***. A similar law applies to the negation of such a ***CSP1*** predicate.

Law 122 (design-post-and-*CSP1*).

$$P \vdash (Q \wedge \boldsymbol{CSP1}(R))) \;=\; (P \vdash Q \wedge R)$$
provided Q and R are ***R1***-healthy

Law 123 (design-post-and-not-*CSP1*).

$$P \vdash (Q \wedge \neg\, \boldsymbol{CSP1}(R))) \;=\; (P \vdash Q \wedge \neg\, R)$$
provided *Q and R are* ***R1****-healthy*

These two laws are combined in the following law that eliminates ***CSP1*** from the condition of a conditional.

Law 124 (design-post-conditional-*CSP1*).

$$(P \vdash (Q \lhd \boldsymbol{CSP1}(R) \rhd S)) \;=\; (P \vdash (Q \lhd R \rhd S))$$
provided *Q, R and S are* ***R1****-healthy*

Proof.

$$
\begin{aligned}
& P \vdash (Q \lhd \mathbf{CSP1}(R) \rhd S) && \text{conditional} \\
= {} & P \vdash (Q \wedge \mathbf{CSP1}(R)) \vee (S \wedge \neg\, \mathbf{CSP1}(R)) && \text{design, propositional calculus} \\
= {} & (P \vdash Q \wedge \mathbf{CSP1}(R)) \vee (P \vdash S \wedge \neg\, \mathbf{CSP1}(R)) \\
& && \text{design-post-and-}\mathbf{CSP1}\text{, assumption: } Q \text{ and } R \text{ are } \mathbf{R1}\text{-healthy} \\
= {} & (P \vdash Q \wedge R) \vee (P \vdash S \wedge \neg\, \mathbf{CSP1}(R)) \\
& && \text{design-post-and-not-}\mathbf{CSP1}\text{, assumption: } S \text{ is } \mathbf{R1}\text{-healthy} \\
= {} & (P \vdash Q \wedge R) \vee (P \vdash S \wedge \neg\, R) && \text{design, propositional calculus, conditional} \\
= {} & P \vdash (Q \lhd R \rhd S)
\end{aligned}
$$

Substitution for *wait* and $okay'$ distributes through $\mathbf{CSP1}$.

Law 125 ($\mathbf{CSP1}$-*wait*-$okay'$).

$$(\mathbf{CSP1}(P))^c_b \;=\; \mathbf{CSP1}(P^c_b) \textbf{ provided } P \textit{ is } \mathbf{R1}\textit{-healthy}$$

The many restrictions on these laws related to $\mathbf{R1}$ healthiness are not a problem, since $\mathbf{CSP1}$ is a healthiness condition on reactive processes.

6.2 *CSP2*

The second healthiness condition for CSP processes, $\mathbf{CSP2}$, is defined in terms of J (which was introduced in Section 4) as follows.

$$\mathbf{CSP2}(P) = P \,;\, J$$

It is a direct consequence of Theorem 3 that $\mathbf{CSP2}$ is a recast of $\mathbf{H2}$, now with an extended alphabet that includes *okay*, *wait*, *tr*, and *ref*. In other words, in the theory of CSP processes, we let go of $\mathbf{H1}$, but we retain $\mathbf{H2}$, under another disguise.

Idempotence and commutative properties for $\mathbf{CSP2}$ follow from those for $\mathbf{H2}$. We add only that it commutes with $\mathbf{CSP1}$.

Law 126 (commutativity-$\mathbf{CSP2}$-$\mathbf{CSP1}$).

$$\mathbf{CSP2} \circ \mathbf{CSP1} = \mathbf{CSP1} \circ \mathbf{CSP2}$$

Closure of designs is not established considering $\mathbf{H1}$ and $\mathbf{H2}$ individually; we consider $\mathbf{H2}$, or $\mathbf{CSP2}$ rather, below. It is not closed with respect to conjunction, and it is not difficult to prove that $P \wedge Q \sqsubseteq \mathbf{CSP2}(P \wedge Q)$, providing P and Q are $\mathbf{CSP2}$.

Theorem 11. *Provided P and Q are $\mathbf{CSP2}$-healthy,*

$$\mathbf{CSP2}(P \vee Q) = P \vee Q \qquad \mathbf{CSP2}\text{-}\vee\text{-closure}$$

$$\mathbf{CSP2}(P \lhd b \rhd Q) = P \lhd b \rhd Q \qquad \mathbf{CSP2}\text{-conditional-closure}$$

$$\mathbf{CSP2}(P \,;\, Q) = P \,;\, Q \qquad \mathbf{CSP2}\text{-sequence-closure}$$

Exercise 15. Prove algebraically that

$$P \wedge Q \sqsubseteq \boldsymbol{CSP2}(P \wedge Q)$$

providing P and Q are ***CSP2***-healthy.

Substitution of *true* for $okay'$ does not distribute through ***CSP2***, but produces the disjunction of two cases.

Law 127 (*CSP2*-converge).

$$(\boldsymbol{CSP2}(P))^t = P^t \vee P^f$$

Proof.

$$\begin{array}{ll}
& (\boldsymbol{CSP2}(P))^t & \boldsymbol{CSP2} \\
= & (P \,;\, J)^t & \text{substitution} \\
= & P \,;\, J^t & J \\
= & P \,;\, ((okay \Rightarrow okay') \wedge \mathit{II}_{rel}^{-okay})^t & \text{substitution} \\
= & P \,;\, ((okay \Rightarrow \mathbf{true}) \wedge \mathit{II}_{rel}^{-okay}) & \text{propositional calculus} \\
= & P \,;\, \mathit{II}_{rel}^{-okay} & \text{propositional calculus} \\
= & P \,;\, (okay \vee \neg\, okay) \wedge \mathit{II}_{rel}^{-okay} & \text{relational calculus} \\
= & P \,;\, okay \wedge \mathit{II}_{rel}^{-okay} \vee P \,;\, \neg\, okay \wedge \mathit{II}_{rel}^{-okay} & okay\text{-boolean, } \mathit{II}_{rel}^{-okay} \\
= & P \,;\, okay = true \wedge tr = tr' \wedge ref = ref' \wedge wait = wait' & \text{one-point} \\
& \vee \\
& P \,;\, okay = false \wedge tr = tr' \wedge ref = ref' \wedge wait = wait' \\
= & P^t \vee P^f
\end{array}$$

Substitution of *false* for $okay'$ eliminates ***CSP2***.

Law 128 (*CSP2*-diverge).

$$(\boldsymbol{CSP2}(P))^f = P^f$$

Proof.

$$\begin{array}{ll}
& (\boldsymbol{CSP2}(P))^f & \boldsymbol{CSP2} \\
= & (P \,;\, J)^f & \text{substitution} \\
= & P \,;\, J^f & J \\
= & P \,;\, ((okay \Rightarrow okay') \wedge \mathit{II}_{rel}^{-okay})^f & \text{substitution} \\
= & P \,;\, ((okay \Rightarrow \mathbf{false}) \wedge \mathit{II}_{rel}^{-okay}) & \text{propositional calculus} \\
= & P \,;\, (\neg\, okay \wedge \mathit{II}_{rel}^{-okay}) & okay\text{-boolean, } \mathit{II}_{rel}^{-okay} \\
= & P \,;\, (okay = false \wedge tr = tr' \wedge ref = ref' \wedge wait = wait') & \text{one-point} \\
= & P^f
\end{array}$$

It is trivial to prove that any reactive design is ***CSP2***, since ***CSP2*** and ***H2*** are the same. A reactive process defined in terms of a design is always ***CSP2***-healthy.

Law 129 (reactive-design-*CSP2*). $\boldsymbol{CSP2}(\boldsymbol{R}(P \vdash Q)) = \boldsymbol{R}(P \vdash Q)$

A CSP process is a reactive process that is both ***CSP1*** and ***CSP2***-healthy. The following theorem shows that any CSP process can be specified in terms of a design using $\boldsymbol{R}$.

Theorem 12. *For every CSP process P,* $P = \boldsymbol{R}(\neg\, P_f^f \vdash P_f^t)$

Together with Laws 121 and 129, this theorem accounts for a style of specification for CSP processes in which we use a design to give its behaviour when the previous process has terminated and not diverged, and leave the definition of the behaviour in the other situations for the healthiness conditions. The precondition of the design characterises the conditions that guarantee that the process does not diverge: it is not the case that, having started (*wait* is *false*), then it diverges (*okay′* is *false*). The postcondition gives the behaviour when, having started, the process does not diverge (*okay′* is *true*).

Figure 1 summarises the relationship between the theories of the UTP we presented so far. In black, we have all the alphabetised predicates; in white, we have the relations: those predicates whose alphabet include only dashed and undashed variables. Designs and reactive processes are disjoint sets of relations. Finally, CSP processes are reactive; moreover, they are the $\boldsymbol{R}$-image of designs.

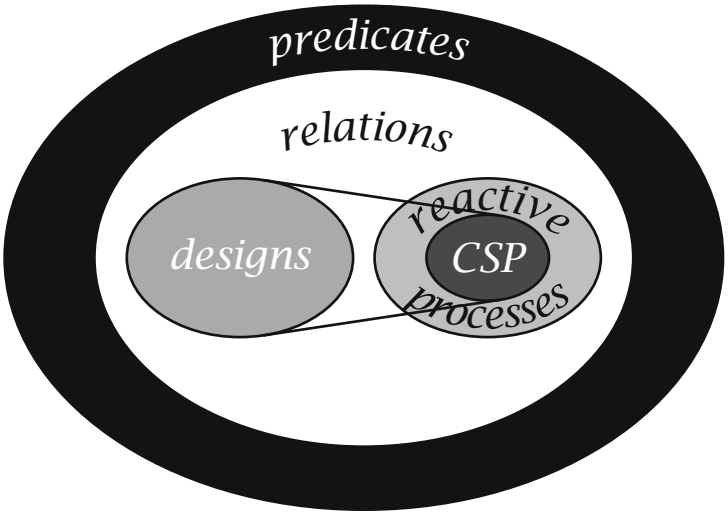

Fig. 1. UTP theories

Motivated by the result above, we express some constructs of CSP as reactive designs. We show that our definitions are the same as those in [117], with a few

exceptions that we explain. Before we proceed, however, we observe that, for CSP processes, $\mathbb{I}_{rea}$ is an identity.

Law 130 ($\mathbb{I}_{rea}$-sequence-CSP).

$$\mathbb{I}_{rea} \,;\, P = P \textbf{ provided } P \textit{ is both } \textbf{\textit{CSP1}} \textit{ and } \textbf{\textit{CSP2}}\textit{-healthy}$$

In spite of its name, $\mathbb{I}_{rea}$ is not a true identity for reactive processes that are not CSP.

Exercise 16. Give an example of a reactive process P for which $\mathbb{I}_{rea};\ P \neq P$.

6.3 *STOP*

We want the following definition for *STOP*.

$$STOP = \mathbf{R}(\mathbf{true} \vdash tr' = tr \wedge wait')$$

Since *STOP* deadlocks, it does not change the trace or terminates. Moreover, all events can be refused; so, we leave the value of ref' unrestrained: any refusal set is a valid observation.

The next law describes the effect of starting *STOP* properly and insisting that it does not diverge. The result is that it leaves the trace unchanged and it waits forever. We need to apply ***CSP1***, since we have not ruled out the possibility of its predecessor diverging.

Law 131 (*STOP*-converge). $STOP^t_f = \textbf{\textit{CSP1}}(tr' = tr \wedge wait')$

Proof.

$$\begin{array}{lr}
STOP^t_f & STOP \\
= (\mathbf{R}(\mathbf{true} \vdash tr' = tr \wedge wait'))^t_f & \textbf{\textit{R}}\textit{-wait-false}, \textbf{\textit{R1}}\textit{-okay}', \textbf{\textit{R2}}\textit{-okay}' \\
= \textbf{\textit{R1}} \circ \textbf{\textit{R2}}((\mathbf{true} \vdash tr' = tr \wedge wait')^t_f) & \text{substitution} \\
= \textbf{\textit{R1}} \circ \textbf{\textit{R2}}((\mathbf{true} \vdash tr' = tr \wedge wait')^t) & \text{design, substitution} \\
= \textbf{\textit{R1}} \circ \textbf{\textit{R2}}(okay \Rightarrow tr' = tr \wedge wait') & \textbf{\textit{R2}} \\
= \textbf{\textit{R1}}(okay \Rightarrow tr' = tr \wedge wait') & \textbf{\textit{H1}} \\
= \textbf{\textit{R1}}(\textbf{\textit{H1}}(tr' = tr \wedge wait')) & \textbf{\textit{R1}} \\
= \textbf{\textit{R1}}(\textbf{\textit{H1}}(\textbf{\textit{R1}}(tr' = tr \wedge wait'))) & \textbf{\textit{CSP1}}\textit{-}\textbf{\textit{R1}}\textit{-}\textbf{\textit{H1}} \text{ and } \textbf{\textit{R1}} \\
= \textbf{\textit{CSP1}}(tr' = tr \wedge wait') &
\end{array}$$

Now we consider the behaviour if we start *STOP* properly, but insist that it does diverge. Of course, *STOP* cannot do this, so the result is that it could not have been started.

Law 132 (*STOP*-diverge). $STOP^f_f = \textbf{\textit{R1}}(\neg\ okay)$

Proof.

$$
\begin{array}{lr}
STOP_f^f & STOP \\
= (\boldsymbol{R}(\mathbf{true} \vdash tr' = tr \wedge wait'))_f^f & \boldsymbol{R}\text{-}wait\text{-}false,\ \boldsymbol{R1}\text{-}okay',\ \boldsymbol{R2}\text{-}okay' \\
= \boldsymbol{R1} \circ \boldsymbol{R2}((\mathbf{true} \vdash tr' = tr \wedge wait')_f^f) & \text{substitution} \\
= \boldsymbol{R1} \circ \boldsymbol{R2}((\mathbf{true} \vdash tr' = tr \wedge wait')^f) & \text{design, substitution} \\
= \boldsymbol{R1} \circ \boldsymbol{R2}(\neg\ okay) & \boldsymbol{R2} \\
= \boldsymbol{R1}(\neg\ okay)
\end{array}
$$

It is possible to prove the following law for *STOP*: it is a left zero for sequence.

Law 133 (STOP-left-zero). $STOP\ ;\ P = STOP$

This gives some reassurance of the validity of our definition.

6.4 *SKIP*

In the UTP, the definition of *SKIP* is as follows.

$$SKIP \mathrel{\widehat{=}} \boldsymbol{R}(\exists\ ref \bullet \Pi_{rea})$$

We propose the formulation presented in the law below.

Law 134 (SKIP-reactive-design). $SKIP = \boldsymbol{R}(\mathbf{true} \vdash tr' = tr \wedge \neg\ wait')$

This characterises *SKIP* as the program that terminates immediately without changing the trace; the refusal set is left unspecified, as it is irrelevant after termination.

6.5 *CHAOS*

The UTP definition for *CHAOS* is $\boldsymbol{R}(\mathbf{true})$. Instead of **true**, we use a design.

Law 135 (CHAOS-reactive-design). $CHAOS = \boldsymbol{R}(\mathbf{false} \vdash \mathbf{true})$

It is perhaps not surprising that *CHAOS* is the reactive abort.

Law 136 (CHAOS-left-zero). $CHAOS\ ;\ P = CHAOS$

The new characterisation of *CHAOS* can be used in the proof of the law above.

6.6 External Choice

For CSP processes P and Q with a common alphabet, their external choice is defined as follows.

$$P \,\Box\, Q \mathrel{\widehat{=}} \boldsymbol{CSP2}((P \wedge Q) \lhd STOP \rhd (P \vee Q))$$

This says that the external choice behaves like the conjunction of P and Q if no progress has been made (that is, if no event has been observed and termination has not occurred). Otherwise, it behaves like their disjunction. This is an economical definition, and we believe that its re-expression as a reactive design is insightful. To prove the law that gives this description, we need a few lemmas, which we present below.

In order to present external choice as a reactive design, we need to calculate a meaningful description for the design $\mathbf{R}(\neg\ (P\ \Box\ Q)^f_f \vdash (P\ \Box\ Q)^t_f)$. that is indicated by Theorem 12. We start with the precondition, and calculate a result for $(P\ \Box\ Q)^f_f$.

Lemma 15 (external-choice-diverge). *Provided P and Q are **R1**-healthy,* $(P\ \Box\ Q)^f_f = (P^f_f \vee Q^f_f) \lhd okay \rhd (P^f_f \wedge Q^f_f)$

This result needs to be negated, but it remains a conditional on the value of *okay*. Since it is a precondition, this conditional may be simplified.

Lemma 16 (external-choice-precondition).

$$(\neg\ (P\ \Box\ Q)^f_f \vdash R\ )\ =\ (\neg\ (P^f_f \vee Q^f_f) \vdash R\ )$$

Now we turn our attention to the postcondition.

Lemma 17 (external-choice-converge).

$$(P\ \Box\ Q)^t_f\ =\ (P \wedge Q) \lhd STOP \rhd (P \vee Q)^t_f \vee (P \wedge Q) \lhd STOP \rhd (P \vee Q)^f_f$$

The second part of the postcondition is in contradiction with the precondition, and when we bring the two together it can be removed. The conditional on *STOP* can then be simplified.

Lemma 18 (design-external-choice-lemma).

$$(\neg\ (P\ \Box\ Q)^f_f \vdash (P\ \Box\ Q)^t_f)\ =\ ((\neg\ P^f_f \wedge Q^f_f) \vdash ((P^t_f \wedge Q^t_f) \lhd tr' = tr \wedge wait' \rhd (P^t_f \vee Q^t_f)))$$

Finally, we collect our results to give external choice as a reactive design.

Law 137 (design-external-choice).

$$P\ \Box\ Q\ =\ \mathbf{R}((\neg\ P^t_f \wedge \neg\ Q^t_f) \vdash (P^t_f \wedge Q^t_f) \lhd tr' = tr \wedge wait' \rhd (P^t_f \vee Q^t_f))$$

Proof.

$$\begin{array}{ll} & P\ \Box\ Q & \text{CSP-reactive-design} \\ = & \mathbf{R}(\neg\ (P\ \Box\ Q)^f_f \vdash (P\ \Box\ Q)^t_f) & \text{design-external-choice-lemma} \\ = & \mathbf{R}((\neg\ P^f_f \wedge \neg\ Q^f_f) \vdash (P^t_f \wedge Q^t_f \lhd tr' = tr \wedge wait' \rhd P^t_f \vee Q^t_f)) & \end{array}$$

The design in this law describes the behaviour of an external choice $P \Box Q$ when its predecessor has terminated without diverging. In this case, the external choice does not diverge if neither P nor Q does; this is captured in the precondition. The postcondition establishes that if there has been no activity, or rather, the trace has not changed and the choice has not terminated, then the behaviour is given by the conjunction of P and Q. If there has been any activity, then the choice has been made and the behaviour is either that of P or that of Q.

Exercise 17. Write out the reactive design that corresponds to the external choice below, where a and b are events.

$$\begin{array}{l} \boldsymbol{R}(true \vdash wait' \wedge tr' = tr \wedge \{\, a \,\} \notin ref' \vee \neg\, wait' \wedge tr' = tr \frown \langle\, a \,\rangle) \\ \Box \\ \boldsymbol{R}(true \vdash wait' \wedge tr' = tr \wedge \{\, b \,\} \notin ref' \vee \neg\, wait' \wedge tr' = tr \frown \langle\, a \,\rangle) \end{array}$$

Exercise 18. How can we write the process below in the notation of CSP?

$$\boldsymbol{R}(true \vdash wait' \wedge tr' = tr \wedge \{\, a \,\} \notin ref' \vee \neg\, wait' \wedge tr' = tr \frown \langle\, a \,\rangle)$$

6.7 Extra Healthiness Conditions: CSP3 and CSP4

The healthiness conditions ***CSP1*** and ***CSP2*** are not strong enough to characterise a UTP model containing only those relations that correspond to processes that can be written using the CSP operators as presented, for example, in Chapter 3. In principle, we need more healthiness conditions to further restrict the subset of reactive processes of interest. As a matter of fact, however, there are advantages to this greater flexibility. In any case, a few other healthiness conditions can be very useful, if not essential. Here, we present two of these.

CSP3. This healthiness condition requires that the behaviour of a process does not depend on the initial value of ref. In other words, it should be the case that, when a process P starts, whatever the previous process could or could not refuse when it finished should be irrelevant. Formally, the ***CSP3*** healthiness condition is $\neg\, wait \Rightarrow (P = \exists\, ref \bullet P)$. If the previous process diverged, $\neg\, okay$, then ***CSP1*** guarantees that the behaviour of P is already independent of ref. So, the restriction imposed by ***CSP3*** is really relevant for the situation $okay \wedge \neg\, wait$, as should be expected.

We can express ***CSP3*** in terms of an idempotent defined as follows.

$$\boldsymbol{CSP3}(P) = SKIP \,;\, P$$

The following lemma establishes that this is the right idempotent.

Lemma 19. *P is **CSP3**-healthy if, and only if, $SKIP \,;\, P = P$.*

Using this idempotent, we can prove that $SKIP$ is ***CSP3***-healthy.

Law 138 (SKIP-*CSP3*). $\boldsymbol{CSP3}(SKIP) = SKIP$

With this result, it is very simple to prove that ***CSP3*** is indeed an idempotent.

$$\textbf{\textit{CSP3}} \circ \textbf{\textit{CSP3}} = \textbf{\textit{CSP3}}$$

Since CSP processes are not closed with respect to conjunction, we only worry about closure of the extra healthiness conditions with respect to the other programming operators.

Theorem 13. *Provided P and Q are* ***CSP3****-healthy,*

$$\textbf{\textit{CSP3}}(P \vee Q) = P \vee Q \qquad \textbf{\textit{CSP3}}\text{-}\vee\text{-closure}$$
$$\textbf{\textit{CSP3}}(P \lhd tr = tr' \rhd Q) = P \lhd tr = tr' \rhd Q \qquad \textbf{\textit{CSP3}}\text{-conditional-closure}$$
$$\textbf{\textit{CSP3}}(P \mathbin{;} Q) = P \mathbin{;} Q \qquad \textbf{\textit{CSP3}}\text{-sequence-closure}$$

CSP4. The second extra healthiness condition, ***CSP4***, is similar to ***CSP3***.

$$\textbf{\textit{CSP4}}(P) = P \mathbin{;} SKIP$$

It requires that, on termination or divergence, the value of ref' is irrelevant. The following lemma makes this clear.

Lemma 20.

$$\begin{aligned} P \mathbin{;} SKIP = {} & (\exists\, ref' \bullet P) \wedge okay' \wedge \neg\, wait' \vee \\ & P \wedge okay' \wedge wait' \vee \\ & (P \wedge \neg\, okay') \mathbin{;} tr \leq tr' \end{aligned}$$

This result shows that, if $P = P \mathbin{;} SKIP$, then if P has terminated without diverging ($okay' \wedge \neg\, wait'$), the value of ref' is not relevant. If P has not terminated ($okay' \wedge wait'$), then the value of ref' is as defined by P itself. Finally, if it diverges ($\not okay'$), then the only guarantee is that the trace is extended; the value of the other variables is irrelevant.

It is easy to prove that $SKIP$, $STOP$, and $CHAOS$ are ***CSP4***-healthy.

Law 139 (SKIP-*CSP4*). $\textbf{\textit{CSP4}}(SKIP) = SKIP$

Law 140 (STOP-*CSP4*). $\textbf{\textit{CSP4}}(STOP) = STOP$

Law 141 (CHAOS-*CSP4*). $\textbf{\textit{CSP4}}(CHAOS) = CHAOS$

The usual closure properties also hold.

Theorem 14. *Provided P and Q are* ***CSP4****-healthy,*

$$\textbf{\textit{CSP4}}(P \vee Q) = P \vee Q \qquad \textbf{\textit{CSP4}}\text{-}\vee\text{-closure}$$
$$\textbf{\textit{CSP4}}(P \lhd b \rhd Q) = P \lhd b \rhd Q \qquad \textbf{\textit{CSP4}}\text{-conditional-closure}$$
$$\textbf{\textit{CSP4}}(P \mathbin{;} Q) = P \mathbin{;} Q \qquad \textbf{\textit{CSP4}}\text{-sequence-closure}$$

As detailed in the next section, other healthiness conditions may be useful. We leave this search as future work; [117] presents an additional healthiness condition that we omit here: ***CSP5***.

7 Failures-Divergences Model

The failures-divergences model is the definitive reference for the semantics of CSP [225]. It is formed by a set F of pairs and a set D of traces. The pairs are the failures of the process. A failure is formed by a trace and a set of events; the trace s records a possible history of interaction, and the set includes the events that the process may refuse after the interactions in the trace. This set is the refusals of P after s. The set D of traces is the divergences of the process. After engaging in the interactions in any of these traces, the process may diverge.

Refinement in this model is defined as reverse containment. A process P_1 is refined by a process P_2 if, and only if, the set of failures and the set of divergences of P_2 are contained or equal to those of P_1.

The simpler traces model includes only a set of traces. For a process P, the set $traces_\perp(P)$ contains the set of all traces in which P can engage, including those that lead to or arise from divergence.

7.1 Failures-Divergences Healthiness Conditions

A number of healthiness conditions are imposed on the failures-divergences model. The first healthiness condition requires that the set of traces of a process is captured in its set of failures, that this set is non-empty and prefix closed. This is because the empty trace is a trace of every process, and every earlier record of interaction is a possible interaction of the process.

F1 $traces_\perp(P) = \{ t \mid (t, X) \in F \}$ is non-empty and prefix closed

The next healthiness condition requires that if (s, X) is a failure, then (s, Y) is also a failure, for all subsets Y of X. This means that, if after s the process may refuse all the events of X, then it may refuse all the events in the subsets of X.

F2 $(s, X) \in F \land Y \subseteq X \Rightarrow (s, Y) \in F$

Also concerning refusals, we have a healthiness condition that requires that if an event is not possible, according to the set of traces of the process, then it must be in the set of refusals.

F3 $(s, X) \in F \land (\forall\, a : Y \bullet s \frown \langle\, a \,\rangle \notin traces_\perp(P)) \Rightarrow (s, X \cup Y) \in F$

The event ✓ is used to mark termination. The following healthiness condition requires that, just before termination, a process can refuse all interactions. The set Σ includes all the events in which the process can engage, except ✓ itself.

F4 $s \frown \langle \checkmark \rangle \in traces_\perp(P) \Rightarrow (s, \Sigma) \in F$

The last three healthiness conditions are related to the divergences of a process. First, if a process can diverge after engaging in the events of a trace s, then it

can diverge after engaging in the events of any extension of s. The idea is that, conceptually, after divergence, any behaviour is possible. Even ✓ is included in the extended traces, and not necessarily as a final event. The set Σ^* includes all traces on events in Σ, and $\Sigma^{*\checkmark}$ includes all traces on events in $\Sigma \cup \{ \checkmark \}$.

$$\textbf{\textit{D1}} \quad s \in D \cap \Sigma^* \land t \in \Sigma^{*\checkmark} \Rightarrow s \frown t \in D$$

The next condition requires that, after divergence, all events may be refused.

$$\textbf{\textit{D2}} \quad s \in D \Rightarrow (s, X) \in F$$

The final healthiness condition requires that if a trace that marks a termination is in the set of divergences, it is because the process diverged before termination. It would not make sense to say that a process diverged after it terminated.

$$\textbf{\textit{D3}} \quad s \frown \langle \checkmark \rangle \in D \Rightarrow s \in D$$

Some of these healthiness conditions correspond to UTP healthiness conditions. Some of them are not contemplated. They are discussed individually later on.

7.2 Failures-Divergences Model of a UTP Process

We can calculate a failures-divergences representation of a UTP process. More precisely, we define a few functions that take a UTP predicate and return a component of the failures-divergences model. We first define a function *traces*; it takes a UTP predicate P and returns the set of traces of the corresponding process.

In the UTP model, the behaviour of a process is that prescribed when *okay* and $\neg$ *wait*. The behaviour in the other cases is determined by the UTP healthiness conditions, and is included in the UTP model so that sequence is simplified: it is just relational composition. In the failures-divergences model, this extra behaviour is not captured and is enforced in the definition of sequence.

The value of tr records the history of events before the start of the process; tr' carries this history forward. This simplifies the definition of sequence. In the failures-divergences model, this extra behaviour is not captured. Therefore, the traces in the set $traces(P)$ are the sequences $tr' - tr$ that arise from the behaviour of P itself.

$$\begin{aligned} traces(P) = \{\, tr' - tr \mid okay \land \neg\, wait \land P \land okay' \,\} \cup \\ \{\, (tr' - tr) \frown \langle \checkmark \rangle \mid okay \land \neg\, wait \land P \land okay' \land \neg\, wait' \,\} \end{aligned}$$

The set $traces(P)$ only includes the traces that lead to non-divergent behaviour. Moreover, if a trace $tr' - tr$ leads to termination, $\neg\, wait'$, then $traces(P)$ also includes $(tr' - tr) \frown \langle \checkmark \rangle$, since ✓ is used in the failures-divergences model to signal termination.

Exercise 19. Calculate $traces(STOP)$.

The traces that lead to or arise from divergent behaviour are those in the set $divergences(P)$ defined below.

$$divergences(P) = \{\, tr' - tr \mid okay \wedge \neg\, wait \wedge P \wedge \neg\, okay' \,\}$$

Exercise 20. Calculate $divergences(STOP)$.

The set $traces_\perp(P)$ mentioned in the healthiness conditions of the failures-divergences model includes both the divergent and non-divergent traces.

$$traces_\perp(P) = traces(P) \cup divergences(P)$$

The failures are recorded for those states that are stable (non-divergent) or final.

$$\begin{array}{l} failures(P) = \\ \quad \{\, ((tr' - tr), ref') \mid okay \wedge \neg\, wait \wedge P \wedge okay' \,\} \cup \\ \quad \{\, ((tr' - tr) \frown \langle \checkmark \rangle, ref') \mid okay \wedge \neg\, wait \wedge P \wedge okay' \wedge \neg\, wait' \,\} \cup \\ \quad \{\, ((tr' - tr) \frown \langle \checkmark \rangle, ref' \cup \{ \checkmark \}) \mid okay \wedge \neg\, wait \wedge P \wedge okay' \wedge \neg\, wait' \,\} \end{array}$$

For the final state, the extra trace $(tr' - tr) \frown \langle \checkmark \rangle$ is recorded. Also, after termination, for every refusal set ref', there is an extra refusal set $ref' \cup \{ \checkmark \}$. This is needed because ✓ is not part of the UTP model and is not considered in the definition of ref'.

Exercise 21. Calculate $failures(STOP)$ and $failures(SKIP)$.

The set of failures in the failures-divergences model includes failures for the divergent traces as well.

$$failures_\perp(P) = failures(P) \cup \{\, (s, ref) \mid s \in divergences(P) \,\}$$

For a divergent trace, there is a failure for each possible refusal set.

The functions $failures_\perp$ and $divergences$ map the UTP model to the failures-divergences model. In studying the relationship between alternative models for a language, it is usual to hope for an isomorphism between them. In our case, this would amount to finding inverses for $failures_\perp$ and $divergences$. Actually, this is not possible; UTP and the failures-divergences model are not isomorphic. This is discussed in detail below.

7.3 Relationship Between the Failures-Divergences and the UTP Model

The UTP model contains processes that cannot be represented in the failures-divergences model. Some of them are useful in a model for a language that has a richer set of constructions to specify data operations. Others may need to be ruled out by further healthiness conditions.

The failures-divergences model, for example, does not have a top element; all divergence-free deterministic processes are maximal. In the UTP model, $\boldsymbol{R}(\mathbf{true} \vdash \mathbf{false})$ is the top.

Lemma 21. *For every CSP process P, we have that $P \sqsubseteq \boldsymbol{R}(\mathbf{true} \vdash \mathbf{false})$.*

The process $\boldsymbol{R}(\mathbf{true} \vdash \mathbf{false})$ is $(\mathbb{I}_{rea} \lhd wait \rhd \neg okay \land tr \leq tr')$. Its behaviour when *okay* and $\neg$ *wait* is **false**. As such, it is mapped to the empty set of failures and divergences; in other words, it is mapped to *STOP*. Operationally, this can make sense, but *STOP* does not have the same properties of $\boldsymbol{R}(\mathbf{true} \vdash \mathbf{false})$. In particular, it does not refine every other process.

Exercise 22. Give an algebraic proof that

$$\boldsymbol{R}(\mathbf{true} \vdash \mathbf{false}) = (\mathbb{I}_{rea} \lhd wait \rhd \neg okay \land tr \leq tr')$$

Exercise 23. Take advantage of the result of the previous exercise to calculate $failures_{\bot}(\boldsymbol{R}(\mathbf{true} \vdash \mathbf{false}))$ and $divergences(\boldsymbol{R}(\mathbf{true} \vdash \mathbf{false}))$.

Exercise 24. Explain why *STOP* does not refine every other process. Consider both the UTP and the failures-divergences models.

In general terms, every process that behaves miraculously in any of its initial states cannot be accurately represented using a failures-divergences model. We do not, however, necessarily want to rule out such processes, as they can be useful as a model for a state-rich CSP.

If we analyse the range of $failures_{\bot}$ and *divergences*, we can see that it does not satisfy a few of the healthiness conditions ***F1-4*** and ***D1-3***.

F1. The set $traces_{\bot}(P)$ is empty for $P = \boldsymbol{R}(\mathbf{true} \vdash \mathbf{false})$; as discussed above, this can be seen as an advantage. Also, $traces_{\bot}(P)$ is not necessarily prefix closed. For example, the process $\boldsymbol{R}(\mathbf{true} \vdash tr' = tr \frown \langle\, a, b \,\rangle \land \neg\, wait')$ engages in the events a and b and then terminates. It does not have a stable state in which a took place, but b is yet to happen.

Exercise 25. Calculate $traces(\boldsymbol{R}(\mathbf{true} \vdash tr' = tr \frown \langle\, a, b \,\rangle \land \neg\, wait'))$. Prove that $\boldsymbol{R}(\mathbf{true} \vdash tr' = tr \frown \langle\, a, b \,\rangle \land \neg\, wait')$ is both ***CSP3*** and ***CSP4***.

F2. This is also not enforced for UTP processes. It is expected to be a consequence of a healthiness condition ***CSP5*** presented in [117].

F3. Again, it is simple to provide a counterexample.

$$\boldsymbol{R}(\mathbf{true} \vdash tr' = tr \frown \langle\, a \,\rangle \land ref' \subseteq \{\, b \,\} \land wait' \lor tr' = tr \frown \langle\, a, b \,\rangle \land \neg\, wait')$$

In this case, a is not an event that can take place again after it has already occurred, and yet it is not being refused.

Exercise 26. Calculate the failures of the process above, and check that it is both ***CSP3*** and ***CSP4***.

F4. This holds for ***CSP4***-healthy processes.

Theorem 15. *Provided P is **CSP4**-healthy,*

$$s \frown \langle \checkmark \rangle \in traces_{\perp}(P) \Rightarrow (s, \Sigma) \in failures(P)$$

D1. Again, ***CSP4*** is required to ensure ***D1***-healthy divergences.

Theorem 16. *Provided P is **CSP4**-healthy,*

$$s \in divergences(P) \cap \Sigma^* \wedge t \in \Sigma^{*\checkmark} \Rightarrow s \frown t \in divergences(P)$$

D2. This is enforced in the definition of $failures_{\perp}$.

D3. Again, this is a simple consequence of the definition (of *divergences*).

Theorem 17.

$$s \frown \langle \checkmark \rangle \in divergences(P) \Rightarrow s \in divergences(P)$$

We view the definition of extra healthiness conditions on UTP processes to ensure ***F1*** and ***F3*** as a challenging task.

8 Conclusions

We have presented two UTP theories of programming: one for pre-post specifications (designs), and one for reactive processes. They have been brought together to form a theory of CSP processes. This is the starting point for the unification of the two theories, whose logical conclusion is a theory of state-rich CSP processes. This is the basis for the semantics of a new notation called *Circus* [255, 51], which combines Z and CSP.

The theory of designs was only briefly discussed. It is the subject of a companion tutorial [256], where through a series of examples, we have presented the alphabetised relational calculus and its sub-theory of designs. In that paper, we have presented the formalisation of four different techniques for reasoning about program correctness.

Even though this is a tutorial introduction to part of the contents of [117], it contains many novel laws and proofs. Notably, the recasting of external choice as a reactive design can be illuminating. Also, the relationship with the failures-divergences model is original.

We hope to have given a didactic and accessible account of the CSP model in the unifying theories of programming. We have left out, however, the definition of many CSP constructs as reactive designs and the exploration of further healthiness conditions. These are going to be the subject of further work.

In [217], UTP is also used to give a semantics to an integration of Z and CSP, which also includes object-oriented features. In [240], the UTP is extended with constructs to capture real-time properties as a first step towards a semantic model for a timed version of *Circus*. In [83], a theory of general correctness is

characterised as an alternative to designs; instead of ***H1*** and ***H2***, a different healthiness condition is adopted to restrict general relations.

Currently, we are collaborating with colleagues to extend UTP to capture mobility, synchronicity, pointers, and object orientation. In particular, in [52] we propose a UTP model for an object-oriented extension of *Circus* based on the language and results discussed in Chapter 2; the details of that model are part of our ongoing work. As explained in Chapter 2, this model can be used to prove the laws proposed there. We hope to contribute to the development of a theory that can support all the major concepts available in modern programming languages.