

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

Ana Cavalcanti and Jim Woodcock

Department of Computer Science
University of York
Heslington, York YO10 5DD, UK
{Ana.Cavalcanti, Jim.Woodcock}@cs.york.ac.uk

Abstract. In their *Unifying Theories of Programming*, Hoare & He use the alphabetised relational calculus to give denotational semantics to a wide variety of constructs taken from different programming paradigms. We give a tutorial introduction to the semantics of CSP processes. We start with a summarised introduction of the alphabetised relational calculus and the theory of designs, which are precondition-postcondition specifications. 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.

1 Introduction

The book by Hoare & He [6] 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 [15] 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 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.

$$B \hat{=} \text{clock} \leq \text{clock}'$$

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.

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

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 $\text{and}_B \hat{=} \lambda X \bullet P \wedge B$ 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 [6]. 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 [5, 11]; we assume knowledge of CSP.

In the next section, we present the most general theory of UTP: the alphabetised predicates. In the following section, we establish that this theory is a complete lattice. Section 4 restricts the general theory to designs; an alternative characterisation of the theory of designs using healthiness conditions is also presented. 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.

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* $(P, Q, \dots, \mathbf{true})$ 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, \dots) and dashed variables (x', a', \dots) ; 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* $(b, c, d, \dots, \mathbf{true})$ 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 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 [6]: 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 is argued as follows.

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

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.

$$\begin{aligned} P \triangleleft b \triangleright Q &\hat{=} (b \wedge P) \vee (\neg b \wedge Q) && \text{if } \alpha b \sqsubseteq \alpha P = \alpha Q \\ \alpha(P \triangleleft b \triangleright Q) &\hat{=} \alpha P \end{aligned}$$

Informally, $P \triangleleft b \triangleright Q$ means P if b else Q .

The presentation of conditional as an infix operator allows the formulation of many laws in a helpful way.

<i>L1</i>	$P \triangleleft b \triangleright P = P$	<i>idempotence</i>
<i>L2</i>	$P \triangleleft b \triangleright Q = Q \triangleleft \neg b \triangleright P$	<i>symmetry</i>
<i>L3</i>	$(P \triangleleft b \triangleright Q) \triangleleft c \triangleright R = P \triangleleft b \wedge c \triangleright (Q \triangleleft c \triangleright R)$	<i>associativity</i>
<i>L4</i>	$P \triangleleft b \triangleright (Q \triangleleft c \triangleright R) = (P \triangleleft b \triangleright Q) \triangleleft c \triangleright (P \triangleleft b \triangleright R)$	<i>distributivity</i>
<i>L5</i>	$P \triangleleft \text{true} \triangleright Q = P = Q \triangleleft \text{false} \triangleright P$	<i>unit</i>
<i>L6</i>	$P \triangleleft b \triangleright (Q \triangleleft b \triangleright R) = P \triangleleft b \triangleright R$	<i>unreachable branch</i>
<i>L7</i>	$P \triangleleft b \triangleright (P \triangleleft c \triangleright Q) = P \triangleleft b \vee c \triangleright Q$	<i>disjunction</i>
<i>L8</i>	$(P \odot Q) \triangleleft b \triangleright (R \odot S) = (P \triangleleft b \triangleright R) \odot (Q \triangleleft b \triangleright S)$	<i>interchange</i>

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 [6] or [14]. We also present extra laws that are useful in later proofs, as well as in illuminating the theory.

Negating a conditional negates its operands, but not its condition.

Law 1 (not-conditional)

$$\neg (P \triangleleft b \triangleright Q) = (\neg P \triangleleft b \triangleright \neg Q)$$

Proof

$$\begin{aligned}
& \neg (P \triangleleft b \triangleright Q) && \text{[conditional]} \\
& = \neg ((b \wedge P) \vee (\neg b \wedge Q)) && \text{[propositional calculus]} \\
& = (b \Rightarrow \neg P) \wedge (\neg b \Rightarrow \neg Q) && \text{[propositional calculus]} \\
& = (b \wedge \neg P) \vee (\neg b \wedge \neg Q) && \text{[conditional]} \\
& = (\neg P \triangleleft b \triangleright \neg Q) && \square
\end{aligned}$$

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: $\neg (P \triangleleft b \triangleright Q) = (\neg Q \triangleleft \neg b \triangleright \neg P)$.

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

Law 2 (conditional-and-not-conditional)

$$(P \triangleleft b \triangleright Q) \wedge \neg (R \triangleleft b \triangleright S) = (P \wedge \neg R) \triangleleft b \triangleright (Q \wedge \neg S)$$

Proof

$$\begin{aligned}
& (P \triangleleft b \triangleright Q) \wedge \neg (R \triangleleft b \triangleright S) && \text{[Law 1]} \\
& = (P \triangleleft b \triangleright Q) \wedge (\neg R \triangleleft b \triangleright \neg S) && \text{[L8]} \\
& = (P \wedge \neg R) \triangleleft b \triangleright (Q \wedge \neg S) && \square
\end{aligned}$$

Implication distributes in both directions through the conditional.

Law 3 (implies-conditional)

$$(P \Rightarrow (Q \triangleleft b \triangleright R)) = ((P \Rightarrow Q) \triangleleft b \triangleright (P \Rightarrow R))$$

$$((P \triangleleft b \triangleright Q) \Rightarrow R) = ((P \Rightarrow R) \triangleleft b \triangleright (Q \Rightarrow R))$$

Proof (rightwards)

$$\begin{aligned} & (P \Rightarrow (Q \triangleleft b \triangleright R)) && \text{[propositional calculus and conditional]} \\ & = (\neg P \vee Q \wedge b \vee R \wedge \neg b) && \text{[case analysis]} \\ & = (\neg P \wedge (b \vee \neg b) \vee Q \wedge b \vee R \wedge \neg b) && \text{[propositional calculus]} \\ & = (\neg P \wedge b \vee \neg P \wedge \neg b \vee Q \wedge b \vee R \wedge \neg b) && \text{[propositional calculus]} \\ & = ((\neg P \vee Q) \wedge b \vee (\neg P \wedge R) \wedge \neg b) && \text{[propositional calculus]} \\ & = ((P \Rightarrow Q) \wedge b \vee \neg b \wedge (P \Rightarrow R)) && \text{[conditional]} \\ & = ((P \Rightarrow Q) \triangleleft b \triangleright (P \Rightarrow R)) \end{aligned}$$

The proof of the opposite direction is similar. □

A consequence of the exchange and unit laws is that conjunction and disjunction both distribute through the conditional.

Law 4 (conditional-conjunction)

$$(P \triangleleft b \triangleright Q) \wedge R = (P \wedge R) \triangleleft b \triangleright (Q \wedge R) \quad \square$$

Law 5 (conditional-disjunction)

$$(P \triangleleft b \triangleright Q) \vee R = (P \vee R) \triangleleft b \triangleright (Q \vee R) \quad \square$$

A conditional may be simplified by using a known condition.

Law 6 (known-condition)

$$\begin{aligned} & b \wedge (P \triangleleft b \triangleright Q) = (b \wedge P) \\ & \neg b \wedge (P \triangleleft b \triangleright Q) = (\neg b \wedge Q) \end{aligned} \quad \square$$

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

Law 7 (assume-if-condition)

$$(P \triangleleft b \triangleright Q) = ((b \wedge P) \triangleleft b \triangleright Q) \quad \square$$

Law 8 (assume-else-condition)

$$(P \triangleleft b \triangleright Q) = (P \triangleleft b \triangleright (\neg b \wedge Q)) \quad \square$$

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.

$$\begin{aligned}
P(v') ; Q(v) &\hat{=} \exists v_0 \bullet P(v_0) \wedge Q(v_0) && \text{if } \text{out}\alpha P = \text{in}\alpha Q' = \{v'\} \\
\text{in}\alpha(P(v') ; Q(v)) &\hat{=} \text{in}\alpha P \\
\text{out}\alpha(P(v') ; Q(v)) &\hat{=} \text{out}\alpha Q
\end{aligned}$$

Sequence is associative and distributes backwards through the conditional.

$$\begin{aligned}
L1 \quad P ; (Q ; R) &= (P ; Q) ; R && \text{associativity} \\
L2 \quad (P \triangleleft b \triangleright Q) ; R &= ((P ; R) \triangleleft b \triangleright (Q ; R)) && \text{left distribution}
\end{aligned}$$

The definition of assignment is basically equality; we need, however, to be careful about the alphabet. If $A = \{x, y, \dots, z\}$ 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.

$$\begin{aligned}
x :=_A e &\hat{=} (x' = e \wedge y' = y \wedge \dots \wedge z' = z) \\
\alpha(x :=_A e) &\hat{=} A \cup A'
\end{aligned}$$

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

$$\begin{aligned}
\Pi_A &\hat{=} (v' = v) && \text{if } A = \{v\} \\
\alpha \Pi_A &\hat{=} A \cup A'
\end{aligned}$$

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 Π .

Π is the identity of sequence.

$$L5 \quad P ; \Pi_{\alpha P} = P = \Pi_{\alpha P} ; P \quad \text{unit}$$

We keep the numbers of the laws presented in [6] that we reproduce here.

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 9 (left-one-point)

$$v' = e ; P = P[e/v] \quad \text{where } \alpha P = \{v, v'\} \quad \square$$

Law 10 (right-one-point)

$$P ; v = e = P[e/v'] \quad \text{where } \alpha P = \{v, v'\}$$

Proof

$$\begin{aligned}
P ; v = e & && \text{[sequence]} \\
= \exists v_0 \bullet P[v_0/v'] \wedge (v = e)[v_0/v] & && \text{[substitution]} \\
= \exists v_0 \bullet P[v_0/v'] \wedge v_0 = e & && \text{[predicate calculus]} \\
= P[v_0/v'][e/v_0] & && \text{[substitution]} \\
= P[e/v'] & && \square
\end{aligned}$$

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.

$$\begin{aligned}
P \sqcap Q &\hat{=} P \vee Q && \text{if } \alpha P = \alpha Q \\
\alpha(P \sqcap Q) &\hat{=} \alpha P
\end{aligned}$$

The alphabet must be the same for both arguments.

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

$$\begin{aligned}
\mathbf{var} \ x &\hat{=} (\exists x \bullet \mathbb{I}_A) & \alpha(\mathbf{var} \ x) &\hat{=} A \setminus \{x\} \\
\mathbf{end} \ x &\hat{=} (\exists x' \bullet \mathbb{I}_A) & \alpha(\mathbf{end} \ x) &\hat{=} A \setminus \{x'\}
\end{aligned}$$

The relation **var** x is not homogeneous, since it does not include x in its alphabet, but it does include x' ; similarly, **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.

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

More interestingly, we can use **var** x and **end** x to specify a variable block.

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

In programs, we use **var** x and **end** x paired in this way, but the separation is useful for reasoning.

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

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

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 suppose that $A = \{w, y, w', y'\}$, 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*.

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

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

3 The complete lattice

The refinement ordering is a partial order: reflexive, anti-symmetric, and transitive. Moreover, the set of alphabetised predicates with a particular alphabet A is a complete lattice under the refinement ordering. Its bottom element is denoted \perp_A , and is the weakest predicate *true*; this is the program that aborts, and 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. These properties of abort and miracle are captured in the following two laws, which hold for all P with alphabet A .

$$\begin{aligned} L1 \quad \perp_A &\sqsubseteq P && \text{bottom element} \\ L2 \quad P &\sqsubseteq \top_A && \text{top element} \end{aligned}$$

The least upper bound is not defined in terms of the relational model, but by the Law **L1** below. This law alone is enough to prove Laws **L1A** and **L1B**, which are actually more useful in proofs.

$$\begin{aligned} L1 \quad P &\sqsubseteq (\sqcap S) \text{ iff } (P \sqsubseteq X \text{ for all } X \text{ in } S) && \text{unbounded nondeterminism} \\ L1A \quad (\sqcap S) &\sqsubseteq X \text{ for all } X \text{ in } S && \text{lower bound} \\ L1B \quad \text{if } P &\sqsubseteq X \text{ for all } X \text{ in } S, \text{ then } P \sqsubseteq (\sqcap S) && \text{greatest lower bound} \end{aligned}$$

These laws characterise basic properties of least upper bounds.

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 and conjunction are not. There is a class of operators that are all monotonic: the disjunctive operators. For example, sequence is disjunctive in both arguments.

$$\begin{aligned} L6 \quad (P \sqcap Q) ; R &= (P ; R) \sqcap (Q ; R) && ; -\sqcap \text{ left distribution} \\ L7 \quad P ; (Q \sqcap R) &= (P ; Q) \sqcap (P ; R) && ; -\sqcap \text{ right distribution} \end{aligned}$$

Since alphabetised relations form a complete lattice, every construction defined solely using monotonic operators has a fixed point. 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 $X = P ; X$, and \perp is the weakest.

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

$$\mu F \hat{=} \sqcap \{ X \mid F(X) \sqsubseteq X \}$$

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.

$$\mu X \bullet \mathcal{C}(X) \hat{=} \mu F \quad \text{where } F \hat{=} \lambda X \bullet \mathcal{C}(X)$$

The standard laws that characterise weakest fixed points are valid.

$$\begin{array}{ll} \mathbf{L1} & \mu F \sqsubseteq Y \text{ if } F(Y) \sqsubseteq Y & \text{weakest fixed point} \\ \mathbf{L2} & [F(\mu F) = \mu F] & \text{fixed point} \end{array}$$

L1 establishes that μF is weaker than any fixed point; **L2** states that μF 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.

$$b * P \hat{=} \mu X \bullet ((P ; X) \triangleleft b \triangleright \mathbb{I})$$

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 $true * \mathbb{I}$, which has the semantics $\mu X \bullet X$.

$$\begin{array}{ll} \mu X \bullet X & \text{[least fixed point]} \\ = \sqcap \{ Y \mid (\lambda X \bullet X)(Y) \sqsubseteq Y \} & \text{[function application]} \\ = \sqcap \{ Y \mid Y \sqsubseteq Y \} & \text{[reflexivity of } \sqsubseteq \text{]} \\ = \sqcap \{ Y \mid true \} & \text{[property of } \sqcap \text{]} \\ = \perp & \square \end{array}$$

A surprising, but simple, consequence of Example 3 is that a program can recover from a non-terminating loop!

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

$$\begin{array}{ll} (b * P) ; x := c & \text{[Example 3]} \\ = \perp ; x := c & \text{[}\perp\text{]} \end{array}$$

$$\begin{aligned}
&= \mathbf{true} ; x := c && \text{[assignment]} \\
&= \mathbf{true} ; x' = c && \text{[sequence]} \\
&= \exists x_0 \bullet \mathbf{true} \wedge x' = c && \text{[predicate calculus]} \\
&= x' = c && \text{[assignment]} \\
&= x := c && \square
\end{aligned}$$

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.

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

The solution 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, we observe that *false* cannot possibly belong to this set, since $\mathbf{false} = \mathbf{false} ; \mathbf{true}$.

The predicates in this set are called designs. They can be split into precondition-postcondition pairs, and are in the same spirit as specification statements used in refinement calculi. As such, they are a basis for unifying languages and methods like B [1], VDM [7], Z, and refinement calculi [8, 2, 9].

In designs, *okay* records that the program has started, and *okay'* that it has terminated. In implementing a design, we are allowed to 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.

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

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

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

Example 5 (Abort).

$$\begin{aligned}
&\mathbf{false} \vdash \mathbf{false} && \text{[design]} \\
&= okay \wedge \mathbf{false} \Rightarrow okay' \wedge \mathbf{false} && \text{[false zero for conjunction]}
\end{aligned}$$

$$\begin{aligned}
&= \mathbf{false} \Rightarrow \mathit{okay}' \wedge \mathbf{false} && \text{[vacuous implication]} \\
&= \mathbf{true} && \text{[vacuous implication]} \\
&= \mathbf{false} \Rightarrow \mathit{okay}' \wedge \mathbf{true} && \text{[false zero for conjunction]} \\
&= \mathit{okay} \wedge \mathbf{false} \Rightarrow \mathit{okay}' \wedge \mathbf{true} && \text{[design]} \\
&= \mathbf{false} \vdash \mathbf{true} && \square
\end{aligned}$$

Miracle has precondition \mathbf{true} , and establishes the impossible: \mathbf{false} .

Example 6 (Miracle).

$$\begin{aligned}
&\mathbf{true} \vdash \mathbf{false} && \text{[design]} \\
&= \mathit{okay} \wedge \mathbf{true} \Rightarrow \mathit{okay}' \wedge \mathbf{false} && \text{[true unit for conjunction]} \\
&= \mathit{okay} \Rightarrow \mathbf{false} && \text{[contradiction]} \\
&= \neg \mathit{okay} && \square
\end{aligned}$$

A reassuring result about a design is the fact that refinement amounts to either weakening the precondition, or strengthening the postcondition in the presence of the precondition. This is established by the result below.

Law 11 *Refinement of designs*

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

The most important result, however, is that abort is a zero for sequence. This was, after all, the whole point for the introduction of designs.

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

In this new setting, it is necessary to redefine assignment and \mathbb{II} , as those introduced previously are not designs.

$$\begin{aligned}
(x := e) &\hat{=} (\mathbf{true} \vdash x' = e \wedge y' = y \wedge \dots \wedge z' = z) \\
\mathbb{II}_D &\hat{=} (\mathbf{true} \vdash \mathbb{II})
\end{aligned}$$

In the sequel, for clarity, we refer to \mathbb{II} as \mathbb{II}_{rel} .

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

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

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

$$T2 \quad ((P_1 \vdash Q_1) \triangleleft b \triangleright (P_2 \vdash Q_2)) = ((P_1 \triangleleft b \triangleright P_2) \vdash (Q_1 \triangleleft b \triangleright Q_2))$$

A sequence of designs $(P_1 \vdash Q_1)$ and $(P_2 \vdash Q_2)$ terminates when P_1 holds, and

Q_1 is guaranteed to establish P_2 . On termination, the sequence establishes the composition of the postconditions.

$$\begin{aligned} T3 \quad & ((P_1 \vdash Q_1); (P_2 \vdash Q_2)) \\ & = ((\neg(\neg P_1; \mathbf{true}) \wedge \neg(Q_1; \neg P_2)) \vdash (Q_1; Q_2)) \end{aligned}$$

Preconditions can be relations, and this fact complicates the statement of Law **T3**; if P_1 is a condition instead, then the law is simplified as follows.

$$T3' \quad ((p_1 \vdash Q_1); (P_2 \vdash Q_2)) = ((p_1 \wedge \neg(Q_1; \neg P_2)) \vdash (Q_1; Q_2))$$

A recursively defined design has as its body a function on designs; as such, it can be seen as a function on precondition-postcondition 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.

$$\begin{aligned} T4 \quad & (\mu X, Y \bullet (F(X, Y) \vdash G(X, Y))) = (P(Q) \vdash Q) \\ & \text{where } P(Y) = (\nu X \bullet F(X, Y)) \text{ and } Q = (\mu Y \bullet P(Y) \Rightarrow G(P(Y), Y)) \end{aligned}$$

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).

$$\begin{aligned} \top_D & \hat{=} (\mathbf{true} \vdash \mathbf{false}) = \neg \mathit{okay} \\ \perp_D & \hat{=} (\mathbf{false} \vdash \mathbf{true}) = \mathbf{true} \end{aligned}$$

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

Theorem 1. *Meets and joins*

$$\begin{aligned} \sqcap_i (P_i \vdash Q_i) & = (\bigwedge_i P_i) \vdash (\bigvee_i Q_i) \\ \sqcup_i (P_i \vdash Q_i) & = (\bigvee_i P_i) \vdash (\bigwedge_i P_i \Rightarrow Q_i) \quad \square \end{aligned}$$

As with the binary choice, the choice $\sqcap_i (P_i \vdash Q_i)$ 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 does, 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 1 (not-design).

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

Proof

$$\begin{aligned} & \neg (p \vdash Q) && [design] \\ = & \neg (okay \wedge p \Rightarrow okay' \wedge Q) && [\neg (P \Rightarrow Q) = P \wedge \neg Q, \text{ twice}] \\ = & okay \wedge p \wedge (okay' \Rightarrow \neg Q) && \square \end{aligned}$$

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

Law 12 (design-contradiction)

$$(P \vdash \neg P) = (P \vdash \mathbf{false})$$

Proof

$$\begin{aligned} & P \vdash \neg P && [design] \\ = & okay \wedge P \Rightarrow okay' \wedge \neg P && [propositional calculus] \\ = & \neg okay \vee \neg P \vee (okay' \wedge \neg P) && [propositional calculus] \\ = & \neg okay \vee \neg P && [propositional calculus] \\ = & \neg okay \vee \neg P \vee (okay' \wedge \mathbf{false}) && [propositional calculus] \\ = & okay \wedge P \Rightarrow okay' \wedge \mathbf{false} && [design] \\ = & P \vdash \mathbf{false} && \square \end{aligned}$$

Another way of characterising the set of designs is by imposing healthiness conditions on the 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 R is **H1** healthy if and only if $R = (okay \Rightarrow R)$. This means that observations cannot be made before the program has started. The **H1**-healthy relations R are exactly those that satisfy the left-zero and unit laws below.

$$\mathbf{true} ; R = \mathbf{true} \quad \text{and} \quad \Pi_D ; R = R$$

The idempotent corresponding to this healthiness condition is defined as

$$\mathbf{H1}(R) = okay \Rightarrow R$$

It is indeed an idempotent.

Law 13 (H1-idempotent)

$$\mathbf{H1} \circ \mathbf{H1} = \mathbf{H1}$$

Proof

$$\begin{aligned}
& \mathbf{H1} \circ \mathbf{H1}(R) && [\mathbf{H1}] \\
& = \text{okay} \Rightarrow (\text{okay} \Rightarrow R) && [\text{propositional calculus}] \\
& = \text{okay} \wedge \text{okay} \Rightarrow R && [\text{propositional calculus}] \\
& = \text{okay} \Rightarrow R && [\mathbf{H1}] \\
& = \mathbf{H1}(R) && \square
\end{aligned}$$

The healthiness condition $\mathbf{H1}$ turns the relational identity into the design \mathbb{I}_D .

Law 14 \mathbb{I}_D -H1- \mathbb{I}

$$\mathbb{I}_D = \mathbf{H1}(\mathbb{I})$$

Proof

$$\begin{aligned}
& \mathbb{I}_D && [\mathbb{I}_D] \\
& = (\mathbf{true} \vdash \mathbb{I}) && [\text{design}] \\
& = (\text{okay} \Rightarrow \text{okay}' \wedge \mathbb{I}) && [\mathbb{I}, \text{propositional calculus}] \\
& = (\text{okay} \Rightarrow \text{okay} \wedge \text{okay}' \wedge \mathbb{I} \wedge \text{okay}' = \text{okay}) && [\text{propositional calculus}] \\
& = (\text{okay} \Rightarrow \text{okay} \wedge \mathbb{I} \wedge \text{okay}' = \text{okay}) && [\mathbb{I}, \text{propositional calculus}] \\
& = (\text{okay} \Rightarrow \mathbb{I}) && [\mathbf{H1}] \\
& = \mathbf{H1}(\mathbb{I}) && \square
\end{aligned}$$

$\mathbf{H1}$ 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 $(\mathbf{true} \vdash (\neg \text{okay} \Rightarrow x' = 0))$, tries to violate $\mathbf{H1}$, but it cannot. We can 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 $\mathbf{H1}$ to the postcondition.

Law 15 (design-post-H1)

$$(P \vdash Q) = (P \vdash \mathbf{H1}(Q))$$

Proof

$$\begin{aligned}
& P \vdash \mathbf{H1}(Q) && [\mathbf{H1}] \\
& = P \vdash (\text{okay} \Rightarrow Q) && [\text{design}] \\
& = \text{okay} \wedge P \Rightarrow \text{okay}' \wedge (\text{okay} \Rightarrow Q) && [\text{propositional calculus}] \\
& = \neg \text{okay} \vee \neg P \vee (\text{okay}' \wedge Q) && [\text{design}] \\
& = P \vdash Q && \square
\end{aligned}$$

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

Law 16 (design-post-not-H1)

$$(P \vdash \neg Q) = (P \vdash \neg \mathbf{H1}(Q))$$

Proof

$$\begin{aligned}
& P \vdash \neg \mathbf{H1}(Q) && [\mathbf{H1}] \\
= & P \vdash \neg (okay \Rightarrow Q) && [\text{propositional calculus}] \\
= & P \vdash okay \wedge \neg Q && [\text{design}] \\
= & okay \wedge P \Rightarrow okay' \wedge okay \wedge \neg Q && [\text{propositional calculus}] \\
= & okay \wedge P \Rightarrow okay' \wedge \neg Q && [\text{design}] \\
= & P \vdash \neg Q && \square
\end{aligned}$$

H1 enjoys many other properties, some of which we see later in the paper.

4.2 H2: possible termination

The second healthiness condition is $[R[\text{false}/okay'] \Rightarrow R[\text{true}/okay']]$. This means that if R is satisfied when $okay'$ is *false*, it is also satisfied then $okay'$ is *true*. In other words, R cannot *require* nontermination, so that it is always possible to terminate.

If P is a predicate with $okay'$ in its alphabet, we abbreviate $P[b/okay']$ as P^b . Furthermore, we abbreviate $P[\text{false}/okay']$ as P^f and $P[\text{true}/okay']$ as P^t . Thus, P is **H2**-healthy if and only if $[P^f \Rightarrow P^t]$.

This healthiness condition may also be expressed in terms of an idempotent. For that, we define a predicate $J \hat{=} (okay \Rightarrow okay') \wedge v' = v$, for an alphabet including $okay$ and $okay'$, and the variables in the lists v and v' . As expected, v' is the list of variables obtained by dashing each of the variables in v .

Law 17 (H2-J)

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

Proof

$$\begin{aligned}
& R = R ; J && [\text{universal equality}] \\
= & \forall okay' \bullet R = R ; J && [okay' \text{ is boolean}] \\
= & (R = R ; J)^t \wedge (R = R ; J)^f && [\text{substitution}] \\
= & (R^t = R ; J^t) \wedge (R^f = R ; J^f) && [J] \\
= & (R^t = R ; (v' = v)) \wedge (R^f = R ; (\neg okay \wedge v' = v)) && [\text{right-one-point}] \\
= & (R^t = R ; (v' = v)) \wedge (R^f = R^f) && [\text{sequence}] \\
= & R^t = \exists okay' \bullet R && [okay' \text{ boolean}]
\end{aligned}$$

$$\begin{aligned}
&= R^t = R^t \vee R^f && \text{[propositional calculus]} \\
&= [R^f \Rightarrow R^t] && \square
\end{aligned}$$

The idempotent corresponding to this healthiness condition is $\mathbf{H2}(R) = R ; J$. It is indeed an idempotent.

Law 18 ($\mathbf{H2}$ -idempotent)

$$\mathbf{H2} \circ \mathbf{H2} = \mathbf{H2}$$

Proof

$$\begin{aligned}
&\mathbf{H2} \circ \mathbf{H2} && \text{[H2]} \\
&= (R ; J) ; J && \text{[associativity]} \\
&= R ; (J ; J) && \text{[J and sequence]} \\
&= R ; \exists ok_0, v_0 \bullet (okay \Rightarrow okay_0) \wedge v_0 = v \wedge (okay_0 \Rightarrow okay') \wedge v' = v_0 && \text{[predicate calculus]} \\
&= R ; \exists ok_0, v_0 \bullet (okay \Rightarrow okay_0) \wedge (okay_0 \Rightarrow okay') \wedge v' = v && \text{[predicate calculus]} \\
&= R ; (okay \Rightarrow okay_0) \wedge v' = v && \text{[J]} \\
&= R ; J && \text{[H2]} \\
&= \mathbf{H2}(R) && \square
\end{aligned}$$

An important property of healthiness conditions is commutativity. For example, $\mathbf{H1}$ and $\mathbf{H2}$ commute.

Law 19 (commutativity- $\mathbf{H2}$ - $\mathbf{H1}$)

$$\mathbf{H1} \circ \mathbf{H2} = \mathbf{H2} \circ \mathbf{H1}$$

This means that we can use $\mathbf{H1}$ and $\mathbf{H2}$ independently to make a relation healthy. The result is a relation that is both $\mathbf{H1}$ and $\mathbf{H2}$ healthy, and, moreover, it is the same no matter in which order we applied $\mathbf{H1}$ and $\mathbf{H2}$.

In the proof of this law, we use the following property of J .

Lemma 2.

$$\neg okay = \neg okay ; J$$

Proof

$$\begin{aligned}
&\neg okay ; J && \text{[J]} \\
&= \neg okay ; (okay \Rightarrow okay') \wedge v' = v && \text{[sequence]} \\
&= \neg okay \wedge \exists okay \bullet okay \Rightarrow okay' && \text{[predicate calculus]} \\
&= \neg okay && \square
\end{aligned}$$

Proof (Law 18)

$$\begin{array}{ll}
\mathbf{H1} \circ \mathbf{H2}(P) & [\mathbf{H1}] \\
= \textit{okay} \Rightarrow \mathbf{H2}(P) & [\mathbf{H2-idempotent}] \\
= \textit{okay} \Rightarrow P ; J & [\textit{propositional calculus}] \\
= \neg \textit{okay} \vee (P ; J) & [\textit{not-okay-J}] \\
= (\neg \textit{okay} ; J) \vee (P ; J) & [; -\square \textit{left distribution}] \\
= \neg (\textit{okay} \vee P) ; J & [\textit{propositional calculus}] \\
= \neg (\textit{okay} \Rightarrow P) ; J & [\mathbf{H2} \textit{ and } \mathbf{H2-idempotent}] \\
= \mathbf{H2} \circ \mathbf{H1}(P) & \square
\end{array}$$

The designs are exactly those relations that are **H1** and **H2** healthy. Relations R that are **H1** and **H2** healthy are designs that can be written as $(\neg R^f \vdash R^t)$, where $R^f = R^f$ and $R^t = R^t$. Moreover, designs are **H1** and **H2** healthy.

5 Reactive processes

A reactive program interacts with its environment, which can include other programs and the users. Its behaviour cannot be characterised by its final state only; we need to record information about the interactions. Actually, many reactive programs do not terminate, and so do not have a final state; yet, they are useful due to their interactions with the environment. The interactions are viewed as events. In the context of CSP processes, these are communications.

To model a reactive process, we need three extra observational variables: tr , $wait$, and ref , besides $okay$. The purpose of tr is to record the trace of events in which the process has engaged. The variable $wait$ is a boolean; it records whether the process has terminated or not. Finally, ref records the set of events in which the process may refuse to engage. Moreover, the variable $okay$ is given a different interpretation: it records divergence.

In view of this intuitive account of the observational variables, we can interpret the occurrences of these variables and their dashed counterparts in a description of a process P as follows.

- $okay$ records whether the process that is executing currently, or has just finished, is in a stable state or has diverged; $okay'$ records whether the next observation of P is that of a stable state or a divergent state.
- $wait$ records whether the execution of the previous process has finished or not; $wait'$ records whether the next observation is that of an intermediate or a final state of P .
- tr records the events which occurred until the last observation; tr' contains all those events that will have occurred until the next observation.
- ref records the set of events that could be refused in the last observation; ref' records the set of events that can be refused in the next observation.

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 $P[\textit{false}/\textit{wait}]$, 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 20 (**H1**-*wait-okay'*)

$$(\mathbf{H1}(P))_b^c = \mathbf{H1}(P_b^c)$$

Proof

$$\begin{aligned} & (\mathbf{H1}(P))_b^c && [\mathbf{H1}] \\ & = (\textit{okay} \Rightarrow P)_b^c && [\textit{substitution}] \\ & = \textit{okay} \Rightarrow P_b^c && [\mathbf{H1}] \\ & = \mathbf{H1}(P_b^c) && \square \end{aligned}$$

Not every relation is a reactive process. As for 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 Π

The reactive Π is defined as follows.

$$\Pi_{rea} \hat{=} \neg \textit{okay} \wedge tr \leq tr' \vee \textit{okay}' \wedge tr' = tr \wedge \textit{wait}' = \textit{wait} \wedge \textit{ref}' = \textit{ref}$$

We use $tr \leq tr'$ to state that tr is a prefix of tr' . It amounts to saying that the process cannot change history, and modify the sequence of events that have occurred previously. Π_{rea} establishes that either we are in a divergent state ($\neg \textit{okay}$), in which case we can only guarantee that the trace is extended, or we are in a stable state, and the value of all the variables is maintained.

Alternative definitions of Π_{rea} can be formulated. For example, it can be defined in terms of the relational Π , which we now call Π_{rel} . First, we introduce the following definition.

$$\Pi_{rel}^{-\textit{okay}} \hat{=} tr' = tr \wedge \textit{wait}' = \textit{wait} \wedge \textit{ref}' = \textit{ref}$$

Clearly, $\Pi_{rea} = \neg \textit{okay} \wedge tr \leq tr' \vee \textit{okay}' \wedge \Pi_{rel}^{-\textit{okay}}$. Now we see a more direct relationship between Π_{rea} and Π_{rel} .

Law 21 (Π_{rea} - Π_{rel})

$$\Pi_{rea} = \neg \textit{okay} \wedge tr \leq tr' \vee \Pi_{rel}$$

Proof

$$\begin{aligned} & \Pi_{rea} && [\Pi_{rea}] \\ & = \neg \textit{okay} \wedge tr \leq tr' \vee \textit{okay}' \wedge \Pi_{rel}^{-\textit{okay}} && [\textit{propositional calculus}] \end{aligned}$$

$$\begin{aligned}
&= (okay \vee \neg okay) \wedge (\neg okay \wedge tr \leq tr' \vee okay' \wedge \Pi_{rel}^{-okay}) \\
&\quad \text{[propositional calculus]} \\
&= (okay \wedge \neg okay \wedge tr \leq tr') \vee (okay \wedge okay' \wedge \Pi_{rel}^{-okay}) \\
&\quad \vee \\
&\quad (\neg okay \wedge \neg okay \wedge tr \leq tr') \vee (\neg okay \wedge okay' \wedge \Pi_{rel}^{-okay}) \\
&\quad \text{[propositional calculus]} \\
&= (okay \wedge okay' \wedge \Pi_{rel}^{-okay}) \\
&\quad \vee \quad \text{[}\Pi_{rel} \text{ and } \Pi_{rel}^{-okay}\text{]} \\
&\quad (\neg okay \wedge tr \leq tr') \\
&\quad \vee \\
&\quad (\neg okay \wedge okay' \wedge \Pi_{rel}^{-okay}) \\
&= (okay \wedge \Pi_{rel}) \vee (\neg okay \wedge tr \leq tr') \vee (\neg okay \wedge okay' \wedge \Pi_{rel}^{-okay}) \\
&\quad \text{[propositional calculus]} \\
&= \Pi_{rel} \vee (\neg okay \wedge tr \leq tr') \vee (\neg okay \wedge okay' \wedge \Pi_{rel}^{-okay}) \\
&\quad \text{[propositional calculus]} \\
&= \Pi_{rel} \vee (\neg okay \wedge tr \leq tr') \vee (\neg okay \wedge okay' \wedge \Pi_{rel}^{-okay} \wedge tr \leq tr') \\
&\quad \text{[propositional calculus]} \\
&= \neg okay \wedge tr \leq tr' \vee \Pi_{rel} \quad \square
\end{aligned}$$

Π_{rea} may also be defined using a conditional.

Law 22 (Π_{rea} -conditional)

$$\Pi_{rea} = \Pi_{rel} \triangleleft okay \triangleright tr \leq tr'$$

Proof

$$\begin{aligned}
&\Pi_{rea} && \text{[Law } \Pi_{rea}\text{-}\Pi_{rel}\text{]} \\
&= (\neg okay \wedge tr \leq tr') \vee \Pi_{rel} && \text{[propositional calculus]} \\
&= (\neg okay \wedge tr \leq tr') \vee (okay \wedge \Pi_{rel}) && \text{[conditional]} \\
&= \Pi_{rel} \triangleleft okay \triangleright tr \leq tr' && \square
\end{aligned}$$

The law below states that, in a non-divergent state, Π_{rea} is just like Π_{rel} .

Law 23 (okay- Π_{rea})

$$okay \wedge \Pi_{rea} = okay \wedge \Pi_{rel}$$

Proof

$$\begin{aligned}
&okay \wedge \Pi_{rea} && \text{[}\Pi_{rea}\text{]} \\
&= okay \wedge (\neg okay \wedge tr \leq tr' \vee okay' \wedge tr' = tr \wedge wait' = wait \wedge ref' = ref) \\
&\quad \text{[propositional calculus]}
\end{aligned}$$

$$\begin{aligned}
&= okay \wedge okay' \wedge tr' = tr \wedge wait' = wait \wedge ref' = ref && [equality] \\
&= okay \wedge okay = okay' \wedge tr' = tr \wedge wait' = wait \wedge ref' = ref && [def. of \Pi_{rel}] \\
&= okay \wedge \Pi_{rel} && \square
\end{aligned}$$

In this case, $okay, \Pi_{rea}$ is a unit for sequence.

Law 24 (okay- Π_{rea} -unit)

$$okay \wedge \Pi_{rea} ; P = okay \wedge P$$

Proof

$$\begin{aligned}
&okay \wedge \Pi_{rea} ; P && [property\ of\ sequence] \\
&= (okay \wedge \Pi_{rea}) ; P && [okay-\Pi_{rea}] \\
&= (okay \wedge \Pi_{rel}) ; P && [sequence] \\
&= okay \wedge \Pi_{rel} ; P && [unit] \\
&= okay \wedge P && \square
\end{aligned}$$

In general, however, it is not an identity. If the state is divergent, then it only guarantees that the trace is either left untouched or extended.

Π_{rea} is **H2** healthy.

Law 25 (Π_{rea} -H2-healthy)

$$\Pi_{rea} = \mathbf{H2}(\Pi_{rea})$$

Proof In this proof, we use v as an abbreviation for all the observational variables. Actually, we use v_0 and v' to stand for the list of corresponding zero-subscripted and dashed variables. We use the same sort of abbreviation in subsequent proofs as well.

$$\begin{aligned}
&\Pi_{rea} = \mathbf{H2}(\Pi_{rea}) && [\mathbf{H2-idempotent}] \\
&= \Pi_{rea} ; J && [J] \\
&= \Pi_{rea} ; ((okay \Rightarrow okay') \wedge tr' = tr \wedge wait' = wait \wedge ref' = ref) && [sequence] \\
&= \exists okay_0, wait_0, tr_0, ref_0 \bullet && [predicate\ calculus] \\
&\quad \Pi_{rea}[v_0/v'] \\
&\quad \wedge \\
&\quad ((okay_0 \Rightarrow okay') \wedge tr' = tr_0 \wedge wait' = wait_0 \wedge ref' = ref_0) \\
&= \exists okay_0 \bullet \Pi_{rea}[okay_0/okay'] \wedge (okay_0 \Rightarrow okay') && [\Pi_{rea}] \\
&= \exists okay_0 \bullet \\
&\quad (\neg okay \wedge tr \leq tr' \vee okay \wedge tr' = tr \wedge wait' = wait \wedge ref' = ref) \\
&\quad \wedge \\
&\quad (okay_0 \Rightarrow okay') && [predicate\ calculus]
\end{aligned}$$

$$\begin{aligned}
&= \neg \text{okay} \wedge tr \leq tr' \vee (\exists \text{okay}_0 \bullet \text{okay}_0 \Rightarrow \text{okay}') && [\text{predicate calculus}] \\
&\quad \vee \\
&\quad \exists \text{okay}_0 \bullet \text{okay}_0 \wedge tr' = tr \wedge \text{wait}' = \text{wait} \wedge \text{ref}' = \text{ref} \wedge (\text{okay}_0 \Rightarrow \text{okay}') \\
&= \neg \text{okay} \wedge tr \leq tr' \vee \text{okay}' \wedge tr' = tr \wedge \text{wait}' = \text{wait} \wedge \text{ref}' = \text{ref} && [\Pi_{rea}] \\
&= \Pi_{rea} && \square
\end{aligned}$$

Π_{rea} is not, however, **H1** healthy. This is because its behaviour when *okay* is false is not arbitrary: the restriction on the traces still applies. In conclusion, Π_{rea} is not a design; in fact, no reactive process is a design, although they can all be expressed in terms of a design. This is further explained later.

5.2 R1

The first reactive healthiness condition states that, what has happened cannot be changed: $P = P \wedge tr \leq tr'$. As explained above for Π_{rea} , this means that a process cannot change the past history of events. An important observation is that this is guaranteed even when the previous process has diverged. More precisely, even if $\neg \text{okay}$, we have this guarantee.

As a function, **R1** is defined as $\mathbf{R1}(P) = P \wedge tr \leq tr'$. It is an idempotent.

Law 26 (R1-idempotent)

$$\mathbf{R1} \circ \mathbf{R1} = \mathbf{R1}$$

Proof

$$\begin{aligned}
&\mathbf{R1} \circ \mathbf{R1}(P) && [\mathbf{R1}] \\
&= P \wedge tr \leq tr' \wedge tr \leq tr' && [\text{propositional calculus}] \\
&= P \wedge tr \leq tr' && [\mathbf{R1}] \\
&= \mathbf{R1}(P) && \square
\end{aligned}$$

Π_{rea} is **R1** healthy.

Law 27 (Π_{rea} -R1-healthy)

$$\Pi_{rea} = \mathbf{R1}(\Pi_{rea})$$

Proof

$$\begin{aligned}
&\Pi_{rea} = \mathbf{R1}(\Pi_{rea}) && [\mathbf{R1}] \\
&= \Pi_{rea} \wedge tr \leq tr' && [\Pi_{rea}] \\
&= (\neg \text{okay} \wedge tr \leq tr' && [tr' = tr \Rightarrow tr \leq tr'] \\
&\quad \vee \\
&\quad \text{okay}' \wedge tr' = tr \wedge \text{wait}' = \text{wait} \wedge \text{ref}' = \text{ref}) \wedge tr \leq tr' \\
&= \neg \text{okay} \wedge tr \leq tr' \vee \text{okay}' \wedge tr' = tr \wedge \text{wait}' = \text{wait} \wedge \text{ref}' = \text{ref} && [\Pi_{rea}] \\
&= \Pi_{rea} && \square
\end{aligned}$$

As already said, Π_{rea} is not **H1** healthy. If we use **H1** to make it healthy, and

then use **R1** to make it a reactive process again, we get back Π_{rea} .

Law 28 (Π_{rea} -**R1-H1**)

$$\Pi_{rea} = \mathbf{R1} \circ \mathbf{H1}(\Pi_{rea})$$

Proof

$$\begin{aligned}
& \mathbf{R1} \circ \mathbf{H1}(\Pi_{rea}) && [\mathbf{R1} \text{ and } \mathbf{H1}] \\
= & (okay \Rightarrow \Pi_{rea}) \wedge tr \leq tr' && [\text{propositional calculus}] \\
= & \neg okay \wedge tr \leq tr' \vee \Pi_{rea} \wedge tr \leq tr' && [\Pi_{rea}] \\
= & \neg okay \wedge tr \leq tr' \vee \neg okay \wedge tr \leq tr' && [\text{propositional calculus}] \\
& \vee \\
& okay' \wedge tr' = tr \wedge wait' = wait \wedge ref' = ref \\
= & \neg okay \wedge tr \leq tr' \vee okay' \wedge tr' = tr \wedge wait' = wait \wedge ref' = ref && [\Pi_{rea}] \\
= & \Pi_{rea} && \square
\end{aligned}$$

This shows that **R1** and **H1** are not independent: they do not commute. **R1**, however, commutes with **H2**.

Law 29 (commutativity-**R1-H2**)

$$\mathbf{H2} \circ \mathbf{R1} = \mathbf{R1} \circ \mathbf{H2}$$

Proof We now use v_0 to abbreviate the list of 0-subscripted variables corresponding to the variables of the alphabet, except $okay_0$. Similarly for v' . In other proofs that follow, we use this sort of abbreviation whenever convenient.

$$\begin{aligned}
& \mathbf{H2}(\mathbf{R1}(P)) && [\text{Law H2-idempotent}] \\
= & \mathbf{R1}(P); J && [\mathbf{R1}] \\
= & (P \wedge tr \leq tr'); J && [J \text{ and sequence}] \\
= & (\exists okay_0 \bullet P[okay_0/okay'] \wedge (okay_0 \Rightarrow okay')) \wedge tr \leq tr' && [\text{predicate calculus}] \\
= & \exists okay_0, v_0 \bullet P[v_0/v'] \wedge tr \leq tr_0 \wedge (okay_0 \Rightarrow okay') \wedge v' = v_0 && [\text{predicate calculus}] \\
= & (\exists okay_0, v_0 \bullet P[v_0/v'] \wedge (okay_0 \Rightarrow okay') \wedge v' = v_0) \wedge tr \leq tr' && [J \text{ and sequence}] \\
= & (P; J) \wedge tr \leq tr' && [\mathbf{R1}] \\
= & \mathbf{R1}(P; J) && [\text{Law H2-idempotent}] \\
= & \mathbf{R1}(\mathbf{H2}(P)) && \square
\end{aligned}$$

Although we have already proved it as a theorem, the fact that Π_{rea} is both **R1**

and **H1** healthy follows immediately from the next law, which embeds \mathbb{I}_{rel} in the space of reactive processes.

Law 30

$$\mathbb{I}_{rea} = \mathbf{R1} \circ \mathbf{H1}(\mathbb{I}_{rel})$$

Proof

$$\begin{aligned}
& \mathbb{I}_{rea} && [\mathbb{I}_{rea} - \mathbb{I}_{rel}] \\
& = (\neg \textit{okay} \wedge tr \leq tr') \vee \mathbb{I}_{rel} && [\mathbb{I}_{rel}] \\
& = (\neg \textit{okay} \wedge tr \leq tr') \vee (\mathbb{I}_{rel} \wedge tr \leq tr') && [\textit{propositional calculus}] \\
& = (\textit{okay} \Rightarrow \mathbb{I}_{rel}) \wedge tr \leq tr' && [\mathbf{H1}] \\
& = \mathbf{H1}(\mathbb{I}_{rel}) \wedge tr \leq tr' && [\mathbf{R1}] \\
& = \mathbf{R1} \circ \mathbf{H1}(\mathbb{I}_{rel}) && \square
\end{aligned}$$

Recall from Law 14 that $\mathbf{H1}(\mathbb{I}_{rel})$ is simply \mathbb{I}_D, \mathbb{I} in the space of designs.

Closure is another important issue for a theory. By applying any of the program operators to an **R1**-healthy process, we get another **R1**-healthy process.

Law 31 (closure- \wedge -R1**)**

$$\mathbf{R1}(P \wedge Q) = P \wedge Q \quad \textit{provided } P \textit{ and } Q \textit{ are } \mathbf{R1} \textit{ healthy}$$

Proof

$$\begin{aligned}
& \mathbf{R1}(P \wedge Q) && [\mathbf{R1}] \\
& = P \wedge Q \wedge tr \leq tr' && [\textit{propositional calculus}] \\
& = P \wedge tr \leq tr' \wedge Q \wedge tr \leq tr' && [\mathbf{R1}] \\
& = \mathbf{R1}(P) \wedge \mathbf{R1}(Q) && [\textit{assumption}] \\
& = P \wedge Q && \square
\end{aligned}$$

Law 32 (closure- \vee -R1**)**

$$\mathbf{R1}(P \vee Q) = P \vee Q \quad \textit{provided } P \textit{ and } Q \textit{ are } \mathbf{R1} \textit{ healthy}$$

Proof Similar to that of Law closure- \wedge -**R1**. \square

Law 33 (closure- \triangleleft - \triangleright -R1**)**

$$\mathbf{R1}(P \triangleleft b \triangleright Q) = P \triangleleft b \triangleright Q \quad \textit{provided } P \textit{ and } Q \textit{ are } \mathbf{R1} \textit{ healthy}$$

Proof

$$\begin{aligned}
& \mathbf{R1}(P \triangleleft b \triangleright Q) && [\mathbf{R1}] \\
& = (P \triangleleft b \triangleright Q) \wedge tr \leq tr' && [\textit{conditional-conjunction}]
\end{aligned}$$

$$\begin{aligned}
&= (P \wedge tr \leq tr') \triangleleft b \triangleright (Q \wedge tr \leq tr') && \text{[R1]} \\
&= (\mathbf{R1}(P) \triangleleft b \triangleright \mathbf{R1}(Q)) && \text{[assumption]} \\
&= (P \triangleleft b \triangleright Q) && \square
\end{aligned}$$

Law 34 (closure-;-R1)

$$\mathbf{R1}(P ; Q) = P ; Q \quad \text{provided } P \text{ and } Q \text{ are } \mathbf{R1} \text{ healthy}$$

Proof

$$\begin{aligned}
&\mathbf{R1}(P ; Q) && \text{[assumption]} \\
&= \mathbf{R1}(\mathbf{R1}(P) ; \mathbf{R1}(Q)) && \text{[R1]} \\
&= ((P \wedge tr \leq tr') ; (Q \wedge tr \leq tr')) \wedge tr \leq tr' && \text{[sequence]} \\
&= \exists tr_0, v_0 \bullet && \text{[property of } \leq \text{]} \\
&\quad P[tr_0, v_0/tr', v'] \wedge tr \leq tr_0 \wedge Q[tr_0, v_0/tr, v] \wedge tr_0 \leq tr' \wedge tr \leq tr' \\
&= \exists tr_0, v_0 \bullet P[tr_0, v_0/tr', v'] \wedge tr \leq tr_0 \wedge Q[tr_0, v_0/tr, v] \wedge tr_0 \leq tr' \\
& && \text{[sequence]} \\
&= \mathbf{R1}(P) ; \mathbf{R1}(Q) && \text{[assumption]} \\
&= P ; Q && \square
\end{aligned}$$

Because **R1** is defined by conjunction, its scope may be extended over other conjunctions.

Law 35 (R1-extends-over-and)

$$\mathbf{R1}(P) \wedge Q = \mathbf{R1}(P \wedge Q)$$

Proof

$$\begin{aligned}
&\mathbf{R1}(P) \wedge Q && \text{[R1]} \\
&= P \wedge tr \leq tr' \wedge Q && \text{[propositional calculus]} \\
&= P \wedge Q \wedge tr \leq tr' && \text{[R1]} \\
&= \mathbf{R1}(P \wedge Q) && \square
\end{aligned}$$

A consequence of this law is that, when an **R1** process is negated, within the scope of another **R1** process, the healthiness condition is not negated.

Law 36 (R1-and-not-R1)

$$\mathbf{R1}(P) \wedge \neg \mathbf{R1}(Q) = \mathbf{R1}(P \wedge \neg Q)$$

Proof

$$\begin{aligned}
&\mathbf{R1}(P) \wedge \neg \mathbf{R1}(Q) && \text{[R1]} \\
&= P \wedge tr \leq tr' \wedge \neg (Q \wedge tr \leq tr') && \text{[propositional calculus]} \\
&= P \wedge tr \leq tr' \wedge \neg Q && \text{[R1]} \\
&= \mathbf{R1}(P \wedge \neg Q) && \square
\end{aligned}$$

Substitution for *wait* and *okay'* distribute through the **R1** healthiness condition.

Law 37 (R1-wait)

$$(\mathbf{R1}(P))_b = \mathbf{R1}(P_b)$$

Proof

$$\begin{aligned} & (\mathbf{R1}(P))_b && [\mathbf{R1}] \\ & = (P \wedge tr \leq tr')_b && [\textit{substitution}] \\ & = P_b \wedge tr \leq tr' && [\mathbf{R1}] \\ & = \mathbf{R1}(P_b) && \square \end{aligned}$$

Law 38 (R1-okay')

$$(\mathbf{R1}(P))^c = \mathbf{R1}(P^c)$$

Proof

$$\begin{aligned} & (\mathbf{R1}(P))^c && [\mathbf{R1}] \\ & = (P \wedge tr \leq tr')^c && [\textit{substitution}] \\ & = P^c \wedge tr \leq tr' && [\mathbf{R1}] \\ & = \mathbf{R1}(P^c) && \square \end{aligned}$$

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

There are two formulations for this healthiness condition. Intuitively, it establishes 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 $tr' - tr$.

$$\begin{aligned} \mathbf{R2}(P(tr, tr')) & \hat{=} \sqcap s \bullet P(s, s \frown (tr' - tr)) \\ \mathbf{R2b}(P(tr, tr')) & \hat{=} P(\langle \rangle, tr' - tr) \end{aligned}$$

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

These formulations characterise the same set of processes: an **R2**-healthy process is also **R2b** healthy, and vice-versa.

Law 39 (R2b-is-R2)

$$\mathbf{R2b} = \mathbf{R2} \circ \mathbf{R2b}$$

Proof

$$\begin{aligned} & \mathbf{R2} \circ \mathbf{R2b}(P(tr, tr')) && [\mathbf{R2b}] \\ & = \mathbf{R2}(P(\langle \rangle, tr' - tr)) && [\mathbf{R2}] \end{aligned}$$

$$\begin{aligned}
&= \Box s \bullet P(\langle \rangle, tr' - tr)(s, s \hat{\wedge} (tr' - tr)) && \text{[substitution]} \\
&= \Box s \bullet P(\langle \rangle, s \hat{\wedge} (tr' - tr) - s) && \text{[property of -]} \\
&= \Box s \bullet P(\langle \rangle, tr' - tr) && \text{[property of } \Box \text{]} \\
&= P(\langle \rangle, tr' - tr) && \text{[R2b]} \\
&= \mathbf{R2b}(P) && \square
\end{aligned}$$

Law 40 (R2-is-R2b)

$$\mathbf{R2} = \mathbf{R2b} \circ \mathbf{R2}$$

Proof

$$\begin{aligned}
&\mathbf{R2b} \circ \mathbf{R2}(P(tr, tr')) && \text{[R2]} \\
&= \mathbf{R2b}(\Box s \bullet P(s, s \hat{\wedge} (tr' - tr))) && \text{[R2b]} \\
&= (\Box s \bullet P(s, s \hat{\wedge} (tr' - tr)))(\langle \rangle, tr' - tr) && \text{[substitution]} \\
&= \Box s \bullet P(s, s \hat{\wedge} (tr' - tr) - \langle \rangle) && \text{[property of -]} \\
&= \Box s \bullet P(s, s \hat{\wedge} (tr' - tr)) && \text{[R2]} \\
&= \mathbf{R2}(P) && \square
\end{aligned}$$

In the sequel, we adopt **R2b** as our second healthiness condition for reactive processes, and actually refer to it as **R2**. Not all properties of **R2b** that we prove in the sequel hold for **R2**; so this is an important point.

R2 is an idempotent.

Law 41 (R2-idempotent)

$$\mathbf{R2} \circ \mathbf{R2} = \mathbf{R2}$$

Proof

$$\begin{aligned}
&\mathbf{R2} \circ \mathbf{R2}(P(tr, tr')) && \text{[R2]} \\
&= P(\langle \rangle, tr' - tr)(\langle \rangle, tr' - tr) && \text{[substitution]} \\
&= P(\langle \rangle, (tr' - tr) - \langle \rangle) && \text{[property of -]} \\
&= P(\langle \rangle, tr' - tr) && \text{[R2]} \\
&= \mathbf{R2}(P) && \square
\end{aligned}$$

R2 is independent from **H1**, **H2**, and **R1**.

Law 42 (commutativity-R2-H1)

$$\mathbf{H1} \circ \mathbf{R2} = \mathbf{R2} \circ \mathbf{H1}$$

Proof

$$\begin{aligned}
&\mathbf{H1}(\mathbf{R2}(P(tr, tr'))) && \text{[H1 and R2]} \\
&= \textit{okay} \Rightarrow P(\langle \rangle, tr' - tr) && \text{[tr and tr' are not free in okay]}
\end{aligned}$$

$$\begin{aligned}
&= (\text{okay} \Rightarrow P)(\langle \rangle, tr' - tr) && [\mathbf{H1} \text{ and } \mathbf{R2}] \\
&= \mathbf{R2}(\mathbf{H1}(P)) && \square
\end{aligned}$$

Law 43 (commutativity- $\mathbf{R2}$ - $\mathbf{H2}$)

$$\mathbf{H2} \circ \mathbf{R2} = \mathbf{R2} \circ \mathbf{H2}$$

Proof

$$\begin{aligned}
&\mathbf{H2}(\mathbf{R2}(P(tr, tr'))) && [\mathbf{R2} \text{ and } \mathbf{H2}\text{-idempotent}] \\
&= P(\langle \rangle, tr' - tr) ; J && [J \text{ and sequence}] \\
&= \exists tr_0, v_0 \bullet && [\text{predicate calculus}] \\
&\quad P(\langle \rangle, tr' - tr)[v_0/v'] \\
&\quad \wedge \\
&\quad (\text{okay}_0 \Rightarrow \text{okay}') \wedge tr' = tr_0 \wedge \text{wait}' = \text{wait}_0 \wedge \text{ref}' = \text{ref}_0 \\
&= \exists \text{okay}_0 \bullet P(\langle \rangle, tr' - tr)[\text{okay}_0/\text{okay}'] \wedge (\text{okay}_0 \Rightarrow \text{okay}') && [\text{substitution}] \\
&= (\exists \text{okay}_0 \bullet P[\text{okay}_0/\text{okay}'] \wedge (\text{okay}_0 \Rightarrow \text{okay}'))(\langle \rangle, tr' - tr) && [\text{predicate calculus}] \\
&= (\exists tr_0, v_0 \bullet && [J \text{ and sequence}] \\
&\quad P[tr_0, v_0/tr', v'] \\
&\quad \wedge \\
&\quad (\text{okay}_0 \Rightarrow \text{okay}') \wedge tr' = tr_0 \wedge \text{wait}' = \text{wait}_0 \wedge \text{ref}' = \text{ref}_0)(\langle \rangle, tr' - tr) \\
&= (P ; J)(\langle \rangle, tr' - tr) && [\mathbf{R2} \text{ and } \mathbf{H2}\text{-idempotent}] \\
&= \mathbf{R2}(\mathbf{H2}(P)) && \square
\end{aligned}$$

Law 44 (commutativity- $\mathbf{R2}$ - $\mathbf{R1}$)

$$\mathbf{R1} \circ \mathbf{R2} = \mathbf{R2} \circ \mathbf{R1} \quad \square$$

Proof

$$\begin{aligned}
&\mathbf{R1} \circ \mathbf{R2}(P(tr, tr')) && [\mathbf{R1} \text{ and } \mathbf{R2}] \\
&= P(\langle \rangle, tr' - tr) \wedge tr \leq tr' && [\leq \text{ and } -] \\
&= P(\langle \rangle, tr' - tr)\langle \rangle \leq tr' - tr && [\text{substitution}] \\
&= (P \wedge tr \leq tr')(\langle \rangle, tr' - tr) && [\mathbf{R1} \text{ and } \mathbf{R2}] \\
&= \mathbf{R2} \circ \mathbf{R1}(P(tr, tr')) && \square
\end{aligned}$$

Again, the programming operators are closed with respect to $\mathbf{R2}$. For the conditional, we have a result for a particular condition. For brevity, we omit proofs.

Law 45 (closure- \wedge - $\mathbf{R2}$)

$$\mathbf{R2}(P \wedge Q) = P \wedge Q \quad \text{provided } P \text{ and } Q \text{ are } \mathbf{R2} \text{ healthy} \quad \square$$

Law 46 (closure- \vee -R2)

$$\mathbf{R2}(P \vee Q) = P \vee Q \quad \text{provided } P \text{ and } Q \text{ are } \mathbf{R2} \text{ healthy} \quad \square$$

Law 47 (closure- $\triangleleft tr' = tr \triangleright _$ -R2)

$$\mathbf{R2}(P \triangleleft tr' = tr \triangleright Q) = P \triangleleft tr' = tr \triangleright Q \quad \text{if } P \text{ and } Q \text{ are } \mathbf{R2} \text{ healthy} \quad \square$$

Law 48 (closure- $;$ -R2)

$$\mathbf{R2}(P ; Q) = P ; Q \quad \text{provided } P \text{ and } Q \text{ are } \mathbf{R2} \text{ healthy} \quad \square$$

Since $\mathbf{R2}$ constrains only tr and tr' , substitution for $wait$ and $okay'$ distribute through its application.

Law 49 (R2- $wait$)

$$(\mathbf{R2}(P))_b = \mathbf{R2}(P_b)$$

Proof

$$\begin{aligned} & (\mathbf{R2}(P))_b && [\mathbf{R2}] \\ = & (\sqcap s \bullet P[s, s \wedge (tr' - tr)/tr, tr'])_b && [\text{substitution}] \\ = & \sqcap s \bullet (P_b)[s, s \wedge (tr' - tr)/tr, tr'] && [\mathbf{R2}] \\ = & \mathbf{R2}(P_b) && \square \end{aligned}$$

Law 50 (R2- $okay'$)

$$(\mathbf{R2}(P))^c = \mathbf{R2}(P^c)$$

Proof

$$\begin{aligned} & (\mathbf{R2}(P))^c && [\mathbf{R2}] \\ = & (\sqcap s \bullet P[s, s \wedge (tr' - tr)/tr, tr'])^c && [\text{substitution}] \\ = & \sqcap s \bullet (P^c)[s, s \wedge (tr' - tr)/tr, tr'] && [\mathbf{R2}] \\ = & \mathbf{R2}(P^c) && \square \end{aligned}$$

The space of relations produced by applying $\mathbf{R2}$ to designs is again a complete lattice, since $\mathbf{R2}$ is also monotonic.

5.4 R3

The third healthiness condition defines the behaviour of a process when that which is currently executing has not finished: $P = (\mathbf{II}_{rea} \triangleleft wait \triangleright P)$. Intuitively, this requires that, if the previous process has not finished, then P should not start: it should behave like \mathbf{II}_{rea} .

The idempotent is $\mathbf{R3}(P) = (\Pi_{rea} \triangleleft wait \triangleright P)$.

Law 51 (*R3-idempotent*)

$$\mathbf{R3} \circ \mathbf{R3} = \mathbf{R3} \quad \square$$

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

Law 52 (*commutativity-R3-H2*)

$$\mathbf{H2} \circ \mathbf{R3} = \mathbf{R3} \circ \mathbf{H2} \quad \square$$

Law 53 (*commutativity-R3-R1*)

$$\mathbf{R1} \circ \mathbf{R3} = \mathbf{R3} \circ \mathbf{R1} \quad \square$$

Law 54 (*commutativity-R3-R2*)

$$\mathbf{R2} \circ \mathbf{R3} = \mathbf{R3} \circ \mathbf{R2} \quad \square$$

Moreover, if all that there is about a process that is not $\mathbf{H1}$ is the fact that it specifies the behaviour required by $\mathbf{R1}$, then we have a commutativity property.

Law 55 (*quasi-commutativity-R3-H1*)

$$\mathbf{R3} \circ \mathbf{R1} \circ \mathbf{H1} = \mathbf{R1} \circ \mathbf{H1} \circ \mathbf{R3} \quad \square$$

This sort of property is important because we are going to express reactive processes as reactive designs.

The following lemmas characterise the behaviour of $\mathbf{R3}$ processes in particular circumstances.

Lemma 3.

$$wait \wedge P = wait \wedge \Pi_{rea} \quad \text{provided } P \text{ is } \mathbf{R3}$$

Lemma 4.

$$\neg okay \wedge wait \wedge \mathbf{R3}(P) = \neg okay \wedge wait \wedge tr \leq tr'$$

Proof

$$\begin{aligned} & \neg okay \wedge wait \wedge \mathbf{R3}(P) && [\mathbf{R3}] \\ & = \neg okay \wedge wait \wedge \Pi_{rea} \triangleleft wait \triangleright R && [\text{known condition}] \\ & = \neg okay \wedge wait \wedge \Pi_{rea} && \Pi_{rea} \\ & = \neg okay \wedge wait \wedge tr \leq tr' && \square \end{aligned}$$

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

Law 56 (closure- \wedge -R3)

$$\mathbf{R3}(P \wedge Q) = P \wedge Q \quad \text{provided } P \text{ and } Q \text{ are } \mathbf{R3} \text{ healthy} \quad \square$$

Law 57 (closure- \vee -R3)

$$\mathbf{R3}(P \vee Q) = P \vee Q \quad \text{provided } P \text{ and } Q \text{ are } \mathbf{R3} \text{ healthy} \quad \square$$

Law 58 (closure- \triangleleft - \triangle -R3)

$$\mathbf{R3}(P \triangleleft _ \triangleright Q) = P \triangleleft _ \triangleright Q \quad \text{provided } P \text{ and } Q \text{ are } \mathbf{R3} \text{ healthy} \quad \square$$

For sequence, we actually require that one of the processes is **R1** as well. This is not a problem because, as detailed in the next section, we actually work with the theory characterised by all healthiness conditions.

Law 59 (closure- $;$ -R3)

$$\mathbf{R3}(P ; Q) = P ; Q \quad \text{provided } P \text{ is } \mathbf{R3}, \text{ and } Q \text{ is } \mathbf{R1} \text{ and } \mathbf{R3} \quad \square$$

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 it as well.

Law 60 (R3-wait-true)

$$(\mathbf{R3}(P))_t = (\mathbb{I}_{rea})_t \quad \square$$

On the other hand, if *false* is substituted for *wait* in $\mathbf{R3}(P)$, then the result is simply P , again with the substitution applied.

Law 61 (R3-not-wait-false)

$$(\mathbf{R3}(P))_f = P_f$$

Proof

$$\begin{aligned} & (\mathbf{R3}(P))_f && \text{[R3]} \\ & = (\mathbb{I}_{rea} \triangleleft \text{wait} \triangleright P)_f && \text{[substitution]} \\ & = (\mathbb{I}_{rea})_f \triangleleft \text{false} \triangleright P_f && \text{[conditional unit]} \\ & = P_f && \square \end{aligned}$$

Substitution for *okay'* interferes with \mathbb{I}_{rea} , and so does not distribute through its application.

Law 62 (R3-okay')

$$(\mathbf{R3}(P))^c = ((\mathbb{I}_{rea})^c \triangleleft \text{wait} \triangleright P^c)$$

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

5.5 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 **R1**, **R2**, and **R3**. We define **R** as the composition of these three.

$$\mathbf{R} \hat{=} \mathbf{R1} \circ \mathbf{R2} \circ \mathbf{R3}$$

Since each of the healthiness conditions **R1**, **R2**, and **R3** commute, their order in the definition above is irrelevant.

For reactive processes, Π_{rea} is indeed a restricted form of identity. To prove that, we use the following lemmas.

Lemma 5. *For a reactive process P,*

$$(tr \leq tr') ; P = tr \leq tr'$$

Proof

$$\begin{aligned}
& (tr \leq tr') ; P && \text{[propositional calculus]} \\
= & (tr \leq tr') ; ((okay \vee \neg okay) \wedge (wait \vee \neg wait) \wedge P) \text{ ; } \neg \square \text{ left distribution} \\
= & (tr \leq tr') ; (okay \wedge wait \wedge P) \vee && \text{[Lemma 4 and P is R3]} \\
& (tr \leq tr') ; (okay \wedge \neg wait \wedge P) \vee \\
& (tr \leq tr') ; (\neg okay \wedge wait \wedge P) \vee \\
& (tr \leq tr') ; (\neg okay \wedge \neg wait \wedge P) \\
= & (tr \leq tr') ; (okay \wedge wait \wedge P) \vee && \text{[P is R1]} \\
& (tr \leq tr') ; (okay \wedge \neg wait \wedge P) \vee \\
& (tr \leq tr') ; (\neg okay \wedge wait \wedge P) \vee \\
& (tr \leq tr') ; (\neg okay \wedge \neg wait \wedge P) \\
= & (tr \leq tr') ; (okay \wedge wait \wedge P \wedge tr \leq tr') \vee && \text{[closure-;R1]} \\
& (tr \leq tr') ; (okay \wedge \neg wait \wedge P \wedge tr \leq tr') \vee \\
& (tr \leq tr') ; (\neg okay \wedge wait \wedge P \wedge tr \leq tr') \vee \\
& (tr \leq tr') ; (\neg okay \wedge \neg wait \wedge P \wedge tr \leq tr') \\
= & ((tr \leq tr') ; (okay \wedge wait \wedge P \wedge tr \leq tr')) \wedge tr \leq tr' \vee && \text{[sequence]} \\
& ((tr \leq tr') ; (okay \wedge \neg wait \wedge P \wedge tr \leq tr')) \wedge tr \leq tr' \vee \\
& ((tr \leq tr') ; (\neg okay \wedge wait \wedge P \wedge tr \leq tr')) \vee \\
& ((tr \leq tr') ; (\neg okay \wedge \neg wait \wedge P)) \wedge tr \leq tr' \\
= & ((tr \leq tr') ; (okay \wedge wait \wedge P \wedge tr \leq tr')) \wedge tr \leq tr' \vee && \text{[pred. calculus]} \\
& ((tr \leq tr') ; (okay \wedge \neg wait \wedge P \wedge tr \leq tr')) \wedge tr \leq tr' \vee \\
& (\exists okay_0, wait_0, tr_0 \bullet tr \leq tr_0 \wedge \neg okay_0 \wedge wait_0 \wedge tr_0 \leq tr') \vee \\
& ((tr \leq tr') ; (\neg okay \wedge \neg wait \wedge P)) \wedge tr \leq tr' \\
= & ((tr \leq tr') ; (okay \wedge wait \wedge P \wedge tr \leq tr')) \wedge tr \leq tr' \vee && \text{[pred. calculus]} \\
& ((tr \leq tr') ; (okay \wedge \neg wait \wedge P \wedge tr \leq tr')) \wedge tr \leq tr' \vee \\
& tr \leq tr' \vee \\
& ((tr \leq tr') ; (\neg okay \wedge \neg wait \wedge P)) \wedge tr \leq tr' \\
= & tr \leq tr' && \square
\end{aligned}$$

Lemma 6. For a reactive process P ,

$$(\neg \text{okay} \wedge tr \leq tr') ; P = (\neg \text{okay} \wedge tr \leq tr')$$

Proof

$$\begin{aligned} & (\neg \text{okay} \wedge tr \leq tr') ; P && \text{[sequence]} \\ & = \neg \text{okay} \wedge (tr \leq tr') ; P && \text{[Lemma 5]} \\ & = \neg \text{okay} \wedge tr \leq tr' && \square \end{aligned}$$

Law 63 (Π_{rea} -sequence-reactive) Provided P is \mathbf{R} healthy,

$$\Pi_{\text{rea}} ; P = (P \triangleleft \text{okay} \triangleright tr \leq tr')$$

Proof

$$\begin{aligned} & \Pi_{\text{rea}} ; P && \text{[propositional calculus]} \\ & = ((\text{okay} \vee \neg \text{okay}) \wedge \Pi_{\text{rea}}) ; P && \text{[}; \neg \text{ left distribution]} \\ & = (\text{okay} \wedge \Pi_{\text{rea}}) ; P \vee (\neg \text{okay} \wedge \Pi_{\text{rea}}) ; P && \text{[okay-}\Pi_{\text{rea}} \text{ and unit]} \\ & = \text{okay} \wedge P \vee (\neg \text{okay} \wedge \Pi_{\text{rea}}) ; P && \text{[}\Pi_{\text{rea}}\text{]} \\ & = \text{okay} \wedge P \vee && \text{[}tr' = tr \Rightarrow tr \leq tr'\text{]} \\ & \quad (\neg \text{okay} \wedge tr \leq tr' \\ & \quad \vee \\ & \quad \neg \text{okay} \wedge \text{okay}' \wedge tr' = tr \wedge \text{wait}' = \text{wait} \wedge \text{ref}' = \text{ref}) ; P \\ & = \text{okay} \wedge P \vee (\neg \text{okay} \wedge tr \leq tr') ; P && \text{[Lemma 6]} \\ & = \text{okay} \wedge P \vee \neg \text{okay} \wedge tr \leq tr' && \text{[conditional]} \\ & = (P \triangleleft \text{okay} \triangleright tr \leq tr') && \square \end{aligned}$$

As a matter of fact, we only need $\mathbf{R1}$ and $\mathbf{R3}$ in this proof.

There are closure properties for \mathbf{R} , which follow from the closure properties for $\mathbf{R1}$, $\mathbf{R2}$, and $\mathbf{R3}$.

Law 64 (closure- \wedge - \mathbf{R})

$$\mathbf{R}(P \wedge Q) = P \wedge Q \quad \text{provided } P \text{ and } Q \text{ are } \mathbf{R} \text{ healthy} \quad \square$$

Law 65 (closure- \vee - \mathbf{R})

$$\mathbf{R}(P \vee Q) = P \vee Q \quad \text{provided } P \text{ and } Q \text{ are } \mathbf{R} \text{ healthy} \quad \square$$

Law 66 (closure- $_ \triangleleft tr' = tr \triangleright _ \mathbf{R}$)

$$\mathbf{R}(P \triangleleft tr' = tr \triangleright Q) = P \triangleleft tr' = tr \triangleright Q \quad \text{if } P \text{ and } Q \text{ are } \mathbf{R} \text{ healthy} \quad \square$$

Law 67 (closure- $;$ - \mathbf{R})

$$\mathbf{R}(P ; Q) = P ; Q \quad \text{provided } P \text{ and } Q \text{ are } \mathbf{R} \text{ healthy} \quad \square$$

Substitution for *wait* cannot distribute through \mathbf{R} , since it does not distribute

through $\mathbf{R3}$; however, it does have the expected simplification properties.

Law 68 (\mathbf{R} -wait-false)

$$(\mathbf{R}(P))_f = \mathbf{R1} \circ \mathbf{R2}(P_f)$$

Proof

$$\begin{aligned}
& (\mathbf{R}(P))_f && [\mathbf{R}] \\
& = (\mathbf{R1} \circ \mathbf{R2} \circ \mathbf{R3}(P))_f && [\mathbf{R1}\text{-wait-okay}'] \\
& = \mathbf{R1}((\mathbf{R2} \circ \mathbf{R3}(P))_f) && [\mathbf{R2}\text{-wait-okay}'] \\
& = \mathbf{R1} \circ \mathbf{R2}((\mathbf{R3}(P))_f) && [\mathbf{R3}\text{-not-wait}] \\
& = \mathbf{R1} \circ \mathbf{R2}(P_f) && \square
\end{aligned}$$

Law 69 (\mathbf{R} -wait-true)

$$(\mathbf{R}(P))_t = (\mathbb{I}_{rea})_t$$

Finally, substitution for $okay'$ does not quite distribute through \mathbf{R} , since it interferes with \mathbb{I}_{rea} .

Law 70 (\mathbf{R} -okay')

$$(\mathbf{R}(P))^c = ((\mathbb{I}_{rea})^c \triangleleft \text{wait} \triangleright \mathbf{R1} \circ \mathbf{R2}(P^c))$$

Since $\mathbf{R1}$, $\mathbf{R2}$, and $\mathbf{R3}$ are all monotonic, so is their composition, and thus \mathbf{R} maps designs to a complete lattice. This is the set of CSP processes, as we establish in the next section.

6 CSP processes

A CSP process is a reactive process that satisfies two other healthiness conditions, which we present in the sequel.

6.1 $\mathbf{CSP1}$

The first healthiness condition requires that, in case of divergence, extension of the trace is the only property that is guaranteed: $P = P \vee (\neg \text{okay} \wedge \text{tr} \leq \text{tr}')$. It is important to observe that $\mathbf{R1}$ requires that, in whatever situation, the trace can only be increased. On the other hand, $\mathbf{CSP1}$ states that, if we are in a divergent state, $\neg \text{okay}$, then there is no other guarantee.

The idempotent is $\mathbf{CSP1}(P) = P \vee \neg \text{okay} \wedge \text{tr} \leq \text{tr}'$.

Law 71 ($\mathbf{CSP1}$ -idempotent)

$$\mathbf{CSP1} \circ \mathbf{CSP1} = \mathbf{CSP1} \quad \square$$

This new healthiness condition is independent from the previous ones.

Law 72 (commutativity-*CSP1-R1*)

$$CSP1 \circ R1 = R1 \circ CSP1 \quad \square$$

Law 73 (commutativity-*CSP1-R2*)

$$CSP1 \circ R2 = R2 \circ CSP1 \quad \square$$

Law 74 (commutativity-*CSP1-R3*)

$$CSP1 \circ R3 = R3 \circ CSP1 \quad \square$$

It is interesting to observe that, like *R1*, *CSP1* does not commute with *H1*. The reason is the same: it specifies behaviour for when \neg *okay*.

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 75 (*CSP1-R1-H1*)

$$CSP1(P) = R1 \circ H1(P) \quad \textit{provided } P \textit{ is } R1 \textit{ healthy} \quad \square$$

A reactive process defined in terms of a design is always *CSP1* healthy.

Law 76 (reactive-design-*CSP1*)

$$R(P \vdash Q) = CSP1(R(P \vdash Q)) \quad \square$$

This is because the design does not restrict the behaviour when \neg *okay*, and *R* insists only that $tr \leq tr'$.

The usual closure properties hold for *CSP1* processes.

Law 77 (closure- \wedge -*CSP1*)

$$CSP1(P \wedge Q) = P \wedge Q \quad \textit{provided } P \textit{ and } Q \textit{ are } CSP1 \textit{ healthy} \quad \square$$

Law 78 (closure- \vee -*CSP1*)

$$CSP1(P \vee Q) = P \vee Q \quad \textit{provided } P \textit{ and } Q \textit{ are } CSP1 \textit{ healthy} \quad \square$$

Law 79 (closure- \triangleleft - \triangle -*CSP1*)

$$CSP1(P \triangleleft _ \triangleright Q) = P \triangleleft _ \triangleright Q \quad \textit{provided } P \textit{ and } Q \textit{ are } CSP1 \textit{ healthy} \quad \square$$

Law 80 (closure- $;$ -*CSP1*)

$$CSP1(P ; Q) = P ; Q \quad \textit{provided } P \textit{ and } Q \textit{ are } CSP1 \textit{ healthy} \quad \square$$

Substitution for *wait* and *okay'* distribute through **CSP1**.

Law 81 (**CSP1-wait-okay'**) *Provided P is R1 healthy,*

$$(\mathbf{CSP1}(P))_b^c = \mathbf{CSP1}(P_b^c)$$

Proof

$$\begin{aligned} & (\mathbf{CSP1}(P))_b^c && [\mathbf{CSP1-R1-H1}, \text{assumption}] \\ = & (\mathbf{R1} \circ \mathbf{H1}(P))_b^c && [\mathbf{R1-wait-okay'}] \\ = & \mathbf{R1}((\mathbf{H1}(P))_b^c) && [\mathbf{H1-wait-okay'}] \\ = & \mathbf{R1} \circ \mathbf{H1}(P_b^c) && [\mathbf{CSP1}] \\ = & \mathbf{CSP1}(P_b^c) && \square \end{aligned}$$

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**.

Law 82 (**design-post-and-CSP1**) *Provided Q and R are R1 healthy.*

$$(P \vdash (Q \wedge \mathbf{CSP1}(R))) = (P \vdash Q \wedge R)$$

Proof

$$\begin{aligned} & P \vdash Q \wedge \mathbf{CSP1}(R) && [\mathbf{CSP1-R1-H1}, \text{assumption: } R \text{ is } \mathbf{R1} \text{ healthy}] \\ = & P \vdash Q \wedge \mathbf{R1} \circ \mathbf{H1}(R) && [\text{assumption: } Q \text{ is } \mathbf{R1} \text{ healthy}] \\ = & P \vdash \mathbf{R1}(Q) \wedge \mathbf{R1} \circ \mathbf{H1}(R) && [\mathbf{R1-extends-over-and}] \\ = & P \vdash \mathbf{R1}(Q) \wedge \mathbf{H1}(R) && [\text{design, propositional calculus}] \\ = & (P \vdash \mathbf{R1}(Q)) \wedge (P \vdash \mathbf{H1}(R)) && [\text{design-post-H1}] \\ = & (P \vdash \mathbf{R1}(Q)) \wedge (P \vdash R) && [\text{design, propositional calculus}] \\ = & P \vdash \mathbf{R1}(Q) \wedge R && [\text{assumption: } Q \text{ is } \mathbf{R1} \text{ healthy}] \\ = & P \vdash Q \wedge R && \square \end{aligned}$$

A similar law applies to the negation of such a **CSP1** predicate.

Law 83 (**design-post-and-not-CSP1**) *Provided Q and R are R1 healthy.*

$$(P \vdash (Q \wedge \neg \mathbf{CSP1}(R))) = (P \vdash Q \wedge \neg R)$$

Proof

$$\begin{aligned} & P \vdash Q \wedge \neg \mathbf{CSP1}(R) && [\mathbf{CSP1-R1-H1}, \text{assumption: } R \text{ is } \mathbf{R1} \text{ healthy}] \\ = & P \vdash Q \wedge \neg \mathbf{R1} \circ \mathbf{H1}(R) && [\text{assumption: } Q \text{ is } \mathbf{R1} \text{ healthy}] \\ = & P \vdash \mathbf{R1}(Q) \wedge \neg \mathbf{R1} \circ \mathbf{H1}(R) && [\mathbf{R1-extends-over-and}] \\ = & P \vdash \mathbf{R1}(Q) \wedge \mathbf{R1}(\neg \mathbf{R1} \circ \mathbf{H1}(R)) && [\mathbf{R1-not-R1}] \end{aligned}$$

$$\begin{aligned}
&= P \vdash \mathbf{R1}(Q) \wedge \mathbf{R1}(\neg \mathbf{H1}(R)) \\
&\quad \quad \quad [\mathbf{R1}\text{-extends-over-and, assumption: } Q \text{ is } \mathbf{R1} \text{ healthy}] \\
&= P \vdash Q \wedge \neg \mathbf{H1}(R) \quad \quad \quad [\text{design, propositional calculus}] \\
&= (P \vdash Q) \wedge (P \vdash \neg \mathbf{H1}(R)) \quad \quad \quad [\text{design-post-not-}\mathbf{H1}] \\
&= (P \vdash \mathbf{R1}(Q)) \wedge (P \vdash \neg R) \quad \quad \quad [\text{design, propositional calculus}] \\
&= P \vdash Q \wedge \neg R \quad \quad \quad \square
\end{aligned}$$

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

Law 84 (design-post-conditional-CSP1) *Provided Q , R , and S are $\mathbf{R1}$ healthy.*

$$(P \vdash (Q \triangleleft \mathbf{CSP1}(R) \triangleright S)) = (P \vdash (Q \triangleleft R \triangleright S))$$

Proof

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

The many restrictions on these laws related to **R1** healthiness are not a problem, since **CSP1** is a healthiness condition on reactive processes.

6.2 CSP2

The second healthiness condition for CSP processes, **CSP2**, is defined in terms of J (which was introduced in Section 4.2) as $P = P ; J$. It is a direct consequence of Law 18 that **CSP2** is a recast of **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 **H1**, but we retain **H2**, under another disguise.

Idempotence and commutative properties for **CSP2** follow from those for **H2**. We add only that it commutes with **CSP1**.

Law 85 (commutativity-CSP2-CSP1)

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

Closure of designs is not established considering **H1** and **H2** individually; we consider **H2** 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 **CSP2**.

Law 86 (closure- \vee -CSP2)

$$\mathbf{CSP2}(P \vee Q) = P \vee Q \quad \text{provided } P \text{ and } Q \text{ are } \mathbf{CSP2} \text{ healthy} \quad \square$$

Law 87 (closure- \triangleleft - \triangle -CSP2)

$$\mathbf{CSP2}(P \triangleleft _ \triangleright Q) = P \triangleleft _ \triangleright Q \quad \text{provided } P \text{ and } Q \text{ are } \mathbf{CSP2} \text{ healthy} \quad \square$$

Law 88 (closure- $;$ -CSP2)

$$\mathbf{CSP2}(P ; Q) = P ; Q \quad \text{provided } Q \text{ is } \mathbf{CSP2} \text{ healthy} \quad \square$$

For CSP processes, Π_{rea} is an identity.

Law 89 (Π_{rea} -sequence-CSP)

$$\Pi_{rea} ; P = P \quad \square$$

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

Law 90 (CSP2-converge)

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

Proof

$$\begin{aligned} & (\mathbf{CSP2}(P))^t && \text{[CSP2]} \\ & = (P ; J)^t && \text{[substitution]} \\ & = P ; J^t && \text{[J]} \\ & = P ; ((okay \Rightarrow okay') \wedge \Pi_{rel}^{-okay})^t && \text{[substitution]} \\ & = P ; ((okay \Rightarrow \mathbf{true}) \wedge \Pi_{rel}^{-okay}) && \text{[propositional calculus]} \\ & = P ; \Pi_{rel}^{-okay} && \text{[propositional calculus]} \\ & = P ; (okay \vee \neg okay) \wedge \Pi_{rel}^{-okay} && \text{[relational calculus]} \\ & = P ; okay \wedge \Pi_{rel}^{-okay} \vee P ; \neg okay \wedge \Pi_{rel}^{-okay} && \text{[okay-boolean, } \Pi_{rel}^{-okay} \text{]} \\ & = P ; okay = \mathbf{true} \wedge tr = tr' \wedge ref = ref' \wedge wait = wait' && \text{[one-point]} \\ & \quad \vee \\ & \quad P ; okay = \mathbf{false} \wedge tr = tr' \wedge ref = ref' \wedge wait = wait' \\ & = P^t \vee P^{\mathbf{false}} && \square \end{aligned}$$

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

Law 91 (CSP2-diverge)

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

Proof

$$\begin{aligned} & (\mathbf{CSP2}(P))^f && \text{[CSP2]} \\ & = (P ; J)^f && \text{[substitution]} \end{aligned}$$

$$\begin{aligned}
&= P ; J^f && [J] \\
&= P ; ((okay \Rightarrow okay') \wedge \Pi_{rel}^{-okay})^f && [substitution] \\
&= P ; ((okay \Rightarrow \mathbf{false}) \wedge \Pi_{rel}^{-okay}) && [propositional\ calculus] \\
&= P ; (\neg okay \wedge \Pi_{rel}^{-okay}) && [okay\text{-boolean}, \Pi_{rel}^{-okay}] \\
&= P ; (okay = \mathbf{false} \wedge tr = tr' \wedge ref = ref' \wedge wait = wait') && [one\text{-point}] \\
&= P^f && \square
\end{aligned}$$

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 92 (reactive-design-CSP2)

$$\mathbf{R}(P \vdash Q) = \mathbf{CSP2}(\mathbf{R}(P \vdash Q)) \quad \square$$

The following theorem shows that any CSP process can be specified in terms of a design using **R**.

Theorem 2. *For every CSP process P,*

$$P = \mathbf{R}(\neg P_f^f \vdash P_f^t) \quad \square$$

Together with Laws 76 and 92, 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.

Motivated by the result above, we express some constructs of CSP as a reactive design. We show that our definitions are the same as those in [6], with a few exceptions that we explain.

6.3 STOP

The UTP definition of *STOP* is as follows.

$$STOP \hat{=} \mathbf{R}(wait := true)$$

At this point it is not clear whether *ref* is in the alphabet or not. We assume it is, since reactive processes are required to include it in the alphabet, and CSP processes are reactive processes. In this case, we want the following definition.

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

The only difference is that we do not restrict the value of *ref'* to be that of *ref*. Since *STOP* deadlocks, all events can be refused. Therefore, 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 93 (STOP-converge)

$$STOP_f^t = CSP1(tr' = tr \wedge wait')$$

Proof

$$\begin{aligned}
& STOP_f^t && [STOP] \\
= & (\mathbf{R}(\mathbf{true} \vdash tr' = tr \wedge wait'))_f^t && [\mathbf{R}\text{-wait-false}, \mathbf{R1}\text{-okay}', \mathbf{R2}\text{-okay}'] \\
= & \mathbf{R1} \circ \mathbf{R2}((\mathbf{true} \vdash tr' = tr \wedge wait'))_f^t && [\text{substitution}] \\
= & \mathbf{R1} \circ \mathbf{R2}((\mathbf{true} \vdash tr' = tr \wedge wait')^t) && [\text{design}, \text{substitution}] \\
= & \mathbf{R1} \circ \mathbf{R2}(okay \Rightarrow tr' = tr \wedge wait') && [\mathbf{R2}] \\
= & \mathbf{R1}(okay \Rightarrow tr' = tr \wedge wait') && [\mathbf{H1}] \\
= & \mathbf{R1}(\mathbf{H1}(tr' = tr \wedge wait')) && [\mathbf{R1}] \\
= & \mathbf{R1}(\mathbf{H1}(\mathbf{R1}(tr' = tr \wedge wait'))) && [\mathbf{CSP1}\text{-}\mathbf{R1}\text{-}\mathbf{H1} \text{ and } \mathbf{R1}] \\
= & CSP1(tr' = tr \wedge wait') && \square
\end{aligned}$$

Now suppose that 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 94 (STOP-diverge)

$$STOP_f^f = \mathbf{R1}(\neg okay)$$

Proof

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

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

Law 95 (STOP-left-zero)

$$STOP ; P = STOP \quad \square$$

This is left as an exercise for the reader.

6.4 SKIP

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

$$SKIP \cong \mathbf{R}(\exists ref \bullet \mathbf{II}_{rea})$$

We propose the formulation presented in the law below.

Law 96 (SKIP-reactive-design)

$$SKIP = \mathbf{R}(\mathbf{true} \vdash tr' = tr \wedge \neg wait') \quad \square$$

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. The proof of this law is left as an exercise.

6.5 CHAOS

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

Law 97 (CHAOS-reactive-design)

$$CHAOS = \mathbf{R}(\mathbf{false} \vdash \mathbf{true}) \quad \square$$

It is perhaps not surprising that *CHAOS* is the reactive abort. An example of the use of this new characterisation can be found in the proof of the law below.

Law 98 (CHAOS-left-zero)

$$CHAOS ; P = CHAOS \quad \square$$

This proof is also left for the reader.

6.6 External choice

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

$$P \square Q \hat{=} \mathbf{CSP2}((P \wedge Q) \triangleleft STOP \triangleright (P \vee Q))$$

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 $\mathbf{R}(\neg (P \square Q)_f^f \vdash (P \square Q)_f^f)$. We start with the precondition, and calculate a result for $(P \square Q)_f^f$.

Lemma 7 (external-choice-diverge). *Provided P and Q are **R1** healthy,*

$$(P \square Q)_f^f = (P_f^f \vee Q_f^f) \triangleleft okay \triangleright (P_f^f \wedge Q_f^f)$$

Proof

$$\begin{aligned} & (P \square Q)_f^f && \text{[external-choice]} \\ = & (\mathbf{CSP2}(P \wedge Q \triangleleft STOP \triangleright P \vee Q))_f^f && \text{[CSP2-diverge]} \end{aligned}$$

$$\begin{aligned}
&= (P \wedge Q \triangleleft STOP \triangleright P \vee Q)_f^f && \text{[substitution]} \\
&= (P \wedge Q)_f^f \triangleleft STOP_f^f \triangleright (P \vee Q)_f^f && \text{[conditional]} \\
&= (P_f^f \wedge Q_f^f \wedge STOP_f^f) \vee ((P_f^f \vee Q_f^f) \wedge \neg STOP_f^f) && \text{[STOP-diverge]} \\
&= (P_f^f \wedge Q_f^f \wedge \mathbf{R1}(\neg okay)) \vee ((P_f^f \vee Q_f^f) \wedge \neg (\mathbf{R1}(\neg okay))) \\
&\hspace{15em} \text{[assumption: } P \text{ and } Q \text{ are } \mathbf{R1}\text{-healthy]} \\
&= (P_f^f \wedge (\mathbf{R1}(Q))_f^f \wedge \mathbf{R1}(\neg okay)) && \text{[}\mathbf{R1}\text{-wait, } \mathbf{R1}\text{-okay', twice]} \\
&\quad \vee \\
&\quad (((\mathbf{R1}(P))_f^f \vee (\mathbf{R1}(Q))_f^f) \wedge \neg (\mathbf{R1}(\neg okay))) \\
&= (P_f^f \wedge \mathbf{R1}(Q_f^f) \wedge \mathbf{R1}(\neg okay)) && \text{[}\mathbf{R1}\text{-extends-over-and, } \mathbf{R1}\text{-disjunctive]} \\
&\quad \vee \\
&\quad ((\mathbf{R1}(P_f^f) \vee \mathbf{R1}(Q_f^f)) \wedge \neg (\mathbf{R1}(\neg okay))) \\
&= (P_f^f \wedge \mathbf{R1}(Q_f^f) \wedge \neg okay) \vee (\mathbf{R1}(P_f^f \vee Q_f^f) \wedge \neg (\mathbf{R1}(\neg okay))) \\
&\hspace{15em} \text{[}\mathbf{R1}\text{-wait, } \mathbf{R1}\text{-okay', } \mathbf{R1}\text{-and-not-}\mathbf{R1}\text{, propositional calculus]} \\
&= (P_f^f \wedge (\mathbf{R1}(Q))_f^f \wedge \neg okay) \vee \mathbf{R1}((P_f^f \vee Q_f^f) \wedge okay) \\
&\hspace{15em} \text{[assumption: } Q \text{ is } \mathbf{R1}\text{-healthy, } \mathbf{R1}\text{-extends-over-and]} \\
&= (P_f^f \wedge Q_f^f \wedge \neg okay) \vee \mathbf{R1}(P_f^f \vee Q_f^f) \wedge okay \\
&\hspace{15em} \text{[}\mathbf{R1}\text{-disjunctive, } \mathbf{R1}\text{-wait, } \mathbf{R1}\text{-okay', assumption: } P \text{ and } Q \text{ are } \mathbf{R1}\text{-healthy]} \\
&= (P_f^f \wedge Q_f^f \wedge \neg okay) \vee ((P_f^f \vee Q_f^f) \wedge okay) && \text{[conditional]} \\
&= (P_f^f \vee Q_f^f) \triangleleft okay \triangleright (P_f^f \wedge Q_f^f) \quad \square
\end{aligned}$$

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 8 (external-choice-precondition).

$$(\neg (P \square Q)_f^f \vdash R) = (\neg (P_f^f \vee Q_f^f) \vdash R)$$

Proof

$$\begin{aligned}
&\neg (P \square Q)_f^f \vdash R && \text{[design]} \\
&= okay \wedge \neg (P \square Q)_f^f \Rightarrow okay' \wedge R && \text{[external-choice-diverge]} \\
&= okay \wedge \neg ((P_f^f \vee Q_f^f) \triangleleft okay \triangleright (P_f^f \wedge Q_f^f)) \Rightarrow okay' \wedge R && \text{[not-conditional]} \\
&= okay \wedge (\neg (P_f^f \vee Q_f^f) \triangleleft okay \triangleright \neg (P_f^f \wedge Q_f^f)) \Rightarrow okay' \wedge R \\
&\hspace{15em} \text{[known-condition]} \\
&= okay \wedge \neg (P_f^f \vee Q_f^f) \Rightarrow okay' \wedge R && \text{[design]} \\
&= \neg (P_f^f \vee Q_f^f) \vdash R \quad \square
\end{aligned}$$

Now we turn our attention to the postcondition.

Lemma 9 (external-choice-converge).


$$(P \sqcap Q)_f^t = (P \wedge Q) \triangleleft STOP \triangleright (P \vee Q)_f^t \\ \vee \\ (P \wedge Q) \triangleleft STOP \triangleright (P \vee Q)_f^f$$

Proof

$$(P \sqcap Q)_f^t \quad [external\ choice] \\ = (\mathbf{CSP2}(P \wedge Q \triangleleft STOP \triangleright P \vee Q))_f^t \quad [\mathbf{CSP2-converge}] \\ = (P \wedge Q \triangleleft STOP \triangleright P \vee Q)_f^t \vee (P \wedge Q \triangleleft STOP \triangleright P \vee Q)_f^f \quad \square$$

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 10 (design-external-choice-lemma).

$$(\neg (P \sqcap Q)_f^f \text{  \sqcap Q)_f^t = \\ ((\neg P_f^f \wedge Q_f^f) \vdash ((P_f^t \wedge Q_f^t) \triangleleft tr' = tr \wedge wait' \triangleright (P_f^t \vee Q_f^t)))$$

Proof

$$\neg (P \sqcap Q)_f^f \vdash (P \sqcap Q)_f^t \\ [external-choice-diverge, design, known-condition, propositional\ calculus] \\ = (\neg P_f^f \wedge \neg Q_f^f) \vdash (P \sqcap Q)_f^t \quad [external-choice-converge] \\ = (\neg P_f^f \wedge \neg Q_f^f) \vdash (P \wedge Q \triangleleft STOP \triangleright P \vee Q)_f^t \\ \vee \\ (P \wedge Q \triangleleft STOP \triangleright P \vee Q)_f^f \\ [design, propositional\ calculus] \\ = (\neg P_f^f \wedge \neg Q_f^f) \vdash (P \wedge Q \triangleleft STOP \triangleright P \vee Q)_f^t \quad [substitution] \\ \vee \\ (\neg P_f^f \wedge \neg Q_f^f) \vdash (P \wedge Q \triangleleft STOP \triangleright P \vee Q)_f^f \\ = (\neg P_f^f \wedge \neg Q_f^f) \vdash (P \wedge Q \triangleleft STOP \triangleright P \vee Q)_f^t \\ \vee \\ (\neg P_f^f \wedge \neg Q_f^f) \vdash (P_f^f \wedge Q_f^f \triangleleft STOP_f^f \triangleright P_f^f \vee Q_f^f) \\ [design, propositional\ calculus] \\ = (\neg P_f^f \wedge \neg Q_f^f) \vdash (P \wedge Q \triangleleft STOP \triangleright P \vee Q)_f^t \quad [design-post-or] \\ \vee \\ (\neg P_f^f \wedge \neg Q_f^f) \vdash \mathbf{false} \\ = (\neg P_f^f \wedge \neg Q_f^f) \vdash (P \wedge Q \triangleleft STOP \triangleright P \vee Q)_f^t \vee \mathbf{false} \\ [propositional\ calculus]$$

$$\begin{aligned}
&= (\neg P_f^f \wedge \neg Q_f^f) \vdash (P \wedge Q \triangleleft STOP \triangleright P \vee Q)_f^t && \text{[substitution]} \\
&= (\neg P_f^f \wedge \neg Q_f^f) \vdash (P_f^t \wedge Q_f^t \triangleleft STOP_f^t \triangleright P_f^t \vee Q_f^t) && \text{[STOP-converge]} \\
&= (\neg P_f^f \wedge \neg Q_f^f) \vdash (P_f^t \wedge Q_f^t \triangleleft \mathbf{CSP1}(tr' = tr \wedge wait') \triangleright P_f^t \vee Q_f^t) \\
&\quad \text{[design-post-conditional-CSP1, assumption: } P \text{ and } Q \text{ R1-healthy]} \\
&= (\neg P_f^f \wedge \neg Q_f^f) \vdash (P_f^t \wedge Q_f^t \triangleleft tr' = tr \wedge wait' \triangleright P_f^t \vee Q_f^t) \quad \square
\end{aligned}$$

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

Law 99 (design-external-choice)

$$P \square Q = \mathbf{R}((\neg P_f^t \wedge \neg Q_f^t) \vdash (P_f^t \wedge Q_f^t) \triangleleft tr' = tr \wedge wait' \triangleright (P_f^t \vee Q_f^t))$$

Proof

$$\begin{aligned}
&P \square Q && \text{[CSP-reactive-design]} \\
&= \mathbf{R}(\neg (P \square Q)_f^f \vdash (P \square Q)_f^t) && \text{[design-external-choice-lemma]} \\
&= \mathbf{R}((\neg P_f^f \wedge \neg Q_f^f) \vdash (P_f^t \wedge Q_f^t \triangleleft tr' = tr \wedge wait' \triangleright P_f^t \vee Q_f^t)) \quad \square
\end{aligned}$$

The design in this law describes the behaviour of an external choice $P \square Q$ when its predecessor has terminated without diverging. In this case, the external choice does itself 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 .

6.7 Extra healthiness conditions: CSP3 and CSP4

The healthiness conditions $\mathbf{CSP1}$ and $\mathbf{CSP2}$ are not strong enough to restrict the predicate model to only those that correspond to processes that can be written using the CSP operators. As a matter of fact, 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 requirement is $\neg wait \Rightarrow (P = \exists ref \bullet P)$. If the previous process diverged, $\neg okay$, then $\mathbf{CSP1}$ guarantees that the behaviour of P is already independent of ref . So, this restriction is really relevant for the situation $okay \wedge \neg wait$, as should be expected.

We can express $\mathbf{CSP3}$ in terms of an idempotent: $\mathbf{CSP3}(P) = \mathbf{SKIP} ; P$.

Lemma 11. P is $\mathbf{CSP3}$ if and only if $\mathbf{SKIP} ; P = P$.

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

Law 100 (SKIP-CSP3)

$$\mathbf{CSP3}(\mathit{SKIP}) = \mathit{SKIP} \quad \square$$

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

Law 101 (CSP3-idempotent)

$$\mathbf{CSP3} \circ \mathbf{CSP3} = \mathbf{CSP3} \quad \square$$

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.

Law 102 (closure- \vee -CSP3)

$$\mathbf{CSP3}(P \vee Q) = P \vee Q \quad \text{provided } P \text{ and } Q \text{ are } \mathbf{CSP3} \text{ healthy} \quad \square$$

Law 103 (closure- $\triangleleft tr' = tr \triangleright$ -CSP3)

$$\begin{aligned} \mathbf{CSP3}(P \triangleleft tr' = tr \triangleright Q) &= P \triangleleft tr' = tr \triangleright Q \\ \text{provided } P \text{ and } Q \text{ are } \mathbf{CSP3} \text{ healthy} & \quad \square \end{aligned}$$

Law 104 (closure- $;$ -CSP3)

$$\mathbf{CSP3}(P ; Q) = P ; Q \quad \text{provided } P \text{ and } Q \text{ are } \mathbf{CSP3} \text{ healthy} \quad \square$$

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

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

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

Lemma 12.

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

This result shows that, if $P = P ; \mathit{SKIP}$, then if P has terminated without diverging, the value of *ref'* is not relevant. If P has not terminated, then the value of *ref'* is as defined by P itself. Finally, if it diverges, 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 105 (SKIP-CSP₄)

$$\mathbf{CSP}_4(\mathit{SKIP}) = \mathit{SKIP} \quad \square$$

Law 106 (STOP-CSP₄)

$$\mathbf{CSP}_4(\mathit{STOP}) = \mathit{STOP} \quad \square$$

Law 107 (CHAOS-CSP₄)

$$\mathbf{CSP}_4(\mathit{CHAOS}) = \mathit{CHAOS} \quad \square$$

The usual closure properties also hold.

Law 108 (closure- \vee -CSP₄)

$$\mathbf{CSP}_4(P \vee Q) = P \vee Q \quad \text{provided } P \text{ and } Q \text{ are } \mathbf{CSP}_4 \text{ healthy} \quad \square$$

Law 109 (closure- \triangleleft - \triangle -CSP₄)

$$\mathbf{CSP}_4(P \triangleleft _ \triangle Q) = P \triangleleft _ \triangle Q \quad \text{provided } P \text{ and } Q \text{ are } \mathbf{CSP}_4 \text{ healthy} \quad \square$$

Law 110 (closure- $;$ -CSP₄)

$$\mathbf{CSP}_4(P ; Q) = P ; Q \quad \text{provided } P \text{ and } Q \text{ are } \mathbf{CSP}_4 \text{ healthy} \quad \square$$

As detailed in the next section, other healthiness conditions may be useful. We leave this search as future work; [6] 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 [11]. 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 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 are the divergences of the process. After engaging in the interactions in any of these traces, the process may diverge.

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

A number of healthiness conditions are imposed on this model. This first healthiness condition requires that the set of traces of a process are captured in its set of failures. 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.

$$\mathbf{F1} \quad \mathit{traces}_\perp(P) = \{ t \mid (t, X) \in F \} \text{ is non-empty and prefix closed}$$

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 .

$$\mathbf{F2} \quad (s, X) \in F \wedge Y \subseteq X \Rightarrow (s, Y) \in F$$

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.

$$\mathbf{F3} \quad (s, X) \in F \wedge (\forall a : Y \bullet s \hat{\ } \langle a \rangle \notin \text{traces}_{\perp}(P)) \Rightarrow (s, X \cup Y) \in F$$

The event \checkmark 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 \checkmark itself.

$$\mathbf{F4} \quad s \hat{\ } \langle \checkmark \rangle \in \text{traces}_{\perp}(P) \Rightarrow (s, \Sigma) \in F$$

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 \checkmark 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 $\Sigma \cup \{\checkmark\}$.

$$\mathbf{D1} \quad s \in D \cap \Sigma^* \wedge t \in \Sigma^{*\checkmark} \Rightarrow s \hat{\ } t \in D$$

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

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

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.

$$\mathbf{D3} \quad s \hat{\ } \langle \checkmark \rangle \in D \Rightarrow s \in D$$

Some of these healthiness conditions correspond to UTP healthiness conditions. Some of them are not contemplated.

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 .

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. The behaviour of the process itself is that prescribed when *okay* and \neg *wait*. The behaviour in the other cases is determined by the healthiness conditions, and is included 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.

$$\begin{aligned} \text{traces}(P) = & \{ tr' - tr \mid okay \wedge \neg wait \wedge P \wedge okay' \} \cup \\ & \{ (tr' - tr) \hat{\ } \langle \checkmark \rangle \mid okay \wedge \neg wait \wedge P \wedge okay' \wedge \neg wait' \} \end{aligned}$$

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

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

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

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

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

$$\text{traces}_{\perp}(P) = \text{traces}(P) \cup \text{divergences}(P)$$

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

$$\begin{aligned} \text{failures}(P) = & \{ ((tr' - tr), ref') \mid okay \wedge \neg wait \wedge P \wedge okay' \} \cup \\ & \{ ((tr' - tr) \hat{\ } \langle \checkmark \rangle, ref') \mid okay \wedge \neg wait \wedge P \wedge okay' \wedge \neg wait' \} \cup \\ & \{ ((tr' - tr) \hat{\ } \langle \checkmark \rangle, ref' \cup \{ \checkmark \}) \mid okay \wedge \neg wait \wedge P \wedge okay' \wedge \neg wait' \} \end{aligned}$$

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

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

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

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.

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, but we leave that for another day.

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

Lemma 13. *For every CSP process P , we have that $P \sqsubseteq \mathbf{R}(\mathbf{true} \vdash \mathbf{false})$.* \square

The process $\mathbf{R}(\mathbf{true} \vdash \mathbf{false})$ is $(\mathbb{I}_{rea} \triangleleft \text{wait} \triangleright \neg \text{okay} \wedge \text{tr} \leq \text{tr}')$. Its behaviour when *okay* and $\neg \text{wait}$ 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 $\mathbf{R}(\mathbf{true} \vdash \mathbf{false})$. In particular, it does not refine every other process.

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*_⊥ and *divergences*, we can see that it does not satisfy a few of the healthiness conditions **F1-4** and **D1-3**.

F1 The set $\text{traces}_{\perp}(P)$ is empty for $P = \mathbf{R}(\mathbf{true} \vdash \mathbf{false})$; as discussed above, this can be seen as an advantage. Also, $\text{traces}_{\perp}(P)$ is not necessarily prefix closed. For example, the process $\mathbf{R}(\mathbf{true} \vdash \text{tr}' = \text{tr} \hat{\ } \langle a, b \rangle \wedge \neg \text{wait}')$ 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.

F2 This is also not enforced for UTP processes. It is expected to be a consequence of a healthiness condition **CSP5** presented in [6].

F3 Again, it is simple to provide a counterexample.

$$\begin{aligned} \mathbf{R}(\mathbf{true} \vdash \text{tr}' = \text{tr} \hat{\ } \langle a \rangle \wedge \text{ref}' \subseteq \{b\} \wedge \text{wait}') \\ \vee \\ \text{tr}' = \text{tr} \hat{\ } \langle a, b \rangle \wedge \neg \text{wait}') \end{aligned}$$

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.

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

Theorem 3. *Provided P is **CSP4** healthy,*

$$s \hat{\ } \langle \surd \rangle \in \text{traces}_{\perp}(P) \Rightarrow (s, \Sigma) \in \text{failures}(P) \quad \square$$

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

Theorem 4. *Provided P is **CSP4** healthy,*

$$s \in \text{divergences}(P) \cap \Sigma^* \wedge t \in \Sigma^{*\surd} \Rightarrow s \hat{\ } t \in \text{divergences}(P) \quad \square$$

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

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

Theorem 5.

$$s \hat{\ } \langle \checkmark \rangle \in \text{divergences}(P) \Rightarrow s \in \text{divergences}(P) \quad \square$$

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

8 Conclusions

We have presented two UTP theories of programming: one for precondition-postcondition specifications (designs), and one for reactive processes. We have brought them 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* [13, 3], which combines Z and CSP.

The theory of designs was only briefly discussed. It is the subject of a companion tutorial [14], 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 [6], 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 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 [10], UTP is also used to give a semantics to an integration of Z and CSP, which also includes object-oriented features. In [12], 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 [4], 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, and object orientation. We hope to contribute to the development of a theory that can support all the major concepts available in modern programming languages.

9 Acknowledgements

This work is partially funded by the Royal Society of London and by QinetiQ Malvern. The authors are the FME (Formal Methods Europe) Lecturers at the

Pernambuco Summer School, and are grateful for their support of this event, where this tutorial was first presented. We have benefited from discussions with Yifeng Chen about closure; the proof we have for Law 48 is his.

References

1. J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
2. R. J. R. Back and J. Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, 1998.
3. A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. A Refinement Strategy for *Circus*. *Formal Aspects of Computing*, 15(2 - 3):146 — 181, 2003.
4. S. Dunne. Recasting Hoare and He's Unifying Theories of Programs in the Context of General Correctness. In A. Butterfield and C. Pahl, editors, *IWFM'01: 5th Irish Workshop in Formal Methods*, BCS Electronic Workshops in Computing, Dublin, Ireland, July 2001.
5. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.
6. C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice-Hall, 1998.
7. C. B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International, 1986.
8. C. C. Morgan. *Programming from Specifications*. Prentice-Hall, 2nd edition, 1994.
9. J. M. Morris. A Theoretical Basis for Stepwise Refinement and the Programming Calculus. *Science of Computer Programming*, 9(3):287 – 306, 1987.
10. S. Qin, J. S. Dong, and W. N. Chin. A Semantic Foundation for TCOZ in Unifying Theories of Programming. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME2003: Formal Methods*, volume 2805 of *Lecture Notes in Computer Science*, pages 321 – 340, 2003.
11. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall Series in Computer Science. Prentice-Hall, 1998.
12. A. Sherif and He Jifeng. Towards a Time Model for *Circus*. In *International Conference in Formal Engineering Methods*, pages 613 – 624, 2002.
13. J. C. P. Woodcock and A. L. C. Cavalcanti. The Semantics of *Circus*. In D. Bert, J. P. Bowen, M. C. Henson, and K. Robinson, editors, *ZB 2002: Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science*, pages 184—203. Springer-Verlag, 2002.
14. J. C. P. Woodcock and A. L. C. Cavalcanti. A Tutorial Introduction to Designs in Unifying Theories of Programming. In *IFM 2004: Integrated Formal Methods*, volume 2999 of *Lecture Notes in Computer Science*, pages 40 – 66. Springer-Verlag, 2004. Invited tutorial.
15. J. C. P. Woodcock and J. Davies. *Using Z—Specification, Refinement, and Proof*. Prentice-Hall, 1996.