On CSP refinement tests that run multiple copies of a process

Gavin Lowe

Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, OX1 3QD, UK.
gavin.lowe@comlab.ox.ac.uk

Abstract

In this paper we consider CSP stable failures refinement checks, where the right hand side of the refinement contains n copies of a process P. We show that such refinement checks capture precisely those predicates of the form $\forall f_1, \ldots, f_n \in \mathit{failures}(P) \bullet R(f_1, \ldots, f_n)$ for some n-ary relation R. The construction of the refinement test is, in general, infinitary; however, we show how, given R, one can often calculate a finite state refinement check.

Keywords: CSP, stable failures refinement checks, expressiveness, testing harnesses.

1 Introduction

This paper considers particular stable failures refinement checks using divergence-free CSP processes P. The majority of such checks that one typically considers are so-called *simple refinement checks* of the form

$$\mathit{Spec} \sqsubseteq_F P,$$

i.e., where the right hand side of the refinement is simply P, and the left hand side of the refinement is independent of P. It is well known that these capture precisely satisfiable failures behavioural specifications [Ros97, Section 3.3], i.e., satisfiable predicates of the form

$$\forall f \in \mathit{failures}(P) \bullet R(f)$$

for some unary relation R over stable failures.

Recently, a number of checks have been considered of the form

$$\mathit{Spec} \sqsubseteq_F \mathit{Harness}(P),$$

where $\mathit{Harness}(P)$ is some testing harness that uses multiple copies of P, say n copies, combined together using CSP operators. For example, the testing harness might run n copies of P in parallel, maybe with some central controller. We call such refinement tests *n-copy refinement tests*.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

One question we ask in this paper is: what predicates over P can we capture using n-copy refinement tests? The answer is precisely predicates of the form

$$Prop_R(P) \mathrel{\widehat{=}} \forall\, f_1, \ldots, f_n \in \mathit{failures}(P) \bullet R(f_1, \ldots, f_n),$$

for n-ary relations R over stable failures. We call such predicates *n-ary failures predicates*. In Section 2, we give some examples of n-ary predicates that have appeared in the literature.

In Section 3, we show that every n-ary failures predicate can be captured as an n-copy refinement test. In Section 4 we show the converse: that every n-copy refinement test captures an n-ary failures predicate.

The construction we give is infinitary in general. In Section 5, we show how the construction can often be reduced to a finite-state refinement check (at least, when P itself is finite-state), suitable for carrying out using a model checker such as FDR [Ros94]. We illustrate the technique on examples in Section 6: we derive the finite-state checks in a semi-systematic way. Many inexperienced users of FDR find it difficult to express requirements as refinement checks: techniques, such as this, for calculating the refinement check should help.

Throughout this paper we restrict attention to non-divergent processes P: divergent processes are normally, ipso facto, incorrect. An advantage of this restriction is that for non-divergent processes, one can calculate the traces from the stable failures; hence the relation R may also specify which traces are allowable, so we can capture both traces and failures requirements within the same mechanism. For simplicity, we also ignore issues of termination.

Roscoe [Ros05] considers the expressive power of CSP refinement checks more generally. He considers which predicates over the failures-divergences model may be captured by refinement checks of the form $F(P) \sqsubseteq G(P)$ where F, G are uniformly continuous CSP contexts. He shows that all predicates that are closed (under the normal topology over the failures divergences model [Ros97]) can be captured in this way, and vice versa. He further shows that every closed and refinement-closed predicate can be captured by a refinement check of the form $\mathit{Spec} \sqsubseteq G(P)$, and vice versa; this includes the form of checks we consider in this paper. Finally, he shows that every satisfiable, refinement-closed and distributive predicate may be captured as a simple refinement check, and vice versa. The predicates we consider in this paper are refinement-closed, but not distributive, hence cannot be captured as simple refinement checks. He conjectures, in passing, the results of Sections 3 and 4 of this paper, but does not prove them.

We conclude this section with a brief overview of the syntax and semantics of CSP; for more details, see [Hoa85,Ros97].

CSP is a process algebra for describing programs or *processes* that interact with their environment by communication. Processes communicate via atomic events, from some set Σ. Events are often compound: the event $c.x$ represents the communication of data x on channel c. $\{|c|\}$ represents the set of all events on channel c.

The simplest process is $STOP$, which represents a deadlocked process that cannot communicate with its environment. The process $a \rightarrow P$ offers its environment the event a; if the event is performed, it then acts like P. If c is a channel then $c?a : A \rightarrow P(a)$ offers its environment the events from the set $\{c.a \mid a \in A\}$; if

the event $c.a$ is performed, the process then acts like $P(a)$. The process $b \mathbin{\&} P$ is equivalent to if b then P else $STOP$: P is enabled only if the boolean guard b is true.

The process $P \Box Q$ can act like either P or Q, the choice being made by the environment: the environment is offered the choice between the initial events of P and Q. By contrast, $P \sqcap Q$ may act like either P or Q, with the choice being made internally, and not under the control of the environment. The process $\sqcap i \in I, R(i) \bullet P(i)$ represents an indexed nondeterministic choice between the processes $P(i)$ for $i \in I$ such that $R(i)$ holds. The process $P \triangleright Q$ represents a sliding choice or timeout: the process initially acts like P, but if no event is performed then it can internally change state to act like Q; we have the following identities: $P \triangleright Q = (P \Box Q) \sqcap Q = (P \sqcap STOP) \Box Q$.

The process $Chaos(A)$ is the most nondeterministic, divergence-free process that performs events from the set A; it can be defined by

$$Chaos(A) \mathrel{\widehat{=}} STOP \sqcap \sqcap x \in A \bullet x \rightarrow Chaos(A).$$

The process $P \setminus A$ acts like P, except the events from A are hidden, i.e. turned into internal, invisible events. The process $P[\![b/a]\!]$ acts like P, except the event a is renamed to b; an indexed version is also available. If c is a channel, then the process $c.P$ acts like P except every event e in P's alphabet, αP, is renamed to $c.e$: $c.P \mathrel{\widehat{=}} P[\![c.e/e \mid e \in \alpha P]\!]$.

The process $P \underset{A}{\parallel} Q$ runs P and Q in parallel, synchronising on events from A. By contract, the process $P \mid\mid\mid Q$ runs P and Q in parallel with no synchronisation.

Prefixing binds tighter than each of the binary choice operators, which in turn bind tighter than the parallel operators.

CSP can be given both an operational and denotational semantics; it is more common to use the denotational semantics when specifying or describing the behaviours of processes, although most tools act on the (congruent) operational semantics.

A *trace* of a process is a sequence of (visible) events that a process can perform. We write $traces(P)$ for the traces of P. If tr is a trace, then $tr \upharpoonright A$ represents the restriction of tr to the events in A, whereas $tr \setminus A$ represents tr with the events from A removed; concatenation is written "$^\frown$"; $first(tr)$ represents the first event of tr.

A *stable failure* of a process P is a pair (tr, X), which represents that P can perform the trace tr to reach a stable state (i.e. where no internal events are possible) where X can be refused, i.e., where none of the events of X is available. We write $failures(P)$ for the stable failures of P. For a divergence-free process

$$traces(P) = \{tr \mid (tr, X) \in failures(P)\}.$$

The traces T and failures F of a process also satisfy the following healthiness conditions:

F1 T is non-empty and prefix-closed;

F2 $(tr, X) \in F \wedge Y \subseteq X \Rightarrow (tr, Y) \in F$;

F3 $(tr, X) \in F \wedge (\forall a \in Y \bullet tr^\frown\langle a \rangle \notin T) \Rightarrow (tr, X \cup Y) \in F$.

Spec is refined by P in the stable failures model if all the behaviours of P are allowed by *Spec*:

$$Spec \sqsubseteq_F P \text{ iff } traces(Spec) \supseteq traces(P) \wedge failures(Spec) \supseteq failures(P).$$

FDR can be used to check this refinement. Note, though, that it considers only finite-state and finite-alphabet processes. For divergence-free processes —as considered in this paper— the traces inclusion is implied by the failures inclusion.

2 Examples

In this section we give some examples of n-ary failures predicates that have appeared in the literature.

2.1 Determinism

The best known such property is that of determinism [Ros97, Section 3.3]. P is deterministic if, whenever it can perform a after tr, it cannot refuse the a:

$$\forall\, tr, a \bullet tr^\frown\langle a\rangle \in traces(P) \Rightarrow (tr, \{a\}) \notin failures(P).$$

This can be re-written as a binary failures predicate as follows:

$$\forall (tr_1, X_1), (tr_2, X_2) \in failures(P) \bullet \forall\, a \bullet tr_1 = tr_2{}^\frown\langle a\rangle \Rightarrow a \notin X_2.$$

2.2 Refinement-closed failures non-deducibility on compositions

The question of information flow considers whether a high-level user, High, of a multi-level security system can pass information to a low-level user, Low. Let the alphabet be partitioned between High's alphabet H and Low's alphabet L.

One family of definitions of information flow is Focardi and Gorrieri's *non-deducibility on compositions* [FG95]. The idea is that however High acts, the system should appear the same to Low. In particular, process P satisfies *failures non-deducibility on compositions* [Foc96], written $FNDC(P)$, if:

$$\forall\, Hi \in CSP_H \bullet \mathcal{L}_H(P \underset{H}{\parallel} Hi) \equiv_F P \underset{H}{\parallel} Stop,$$

where $\mathcal{L}_H(Q) \mathrel{\widehat{=}} (Q \underset{H}{\parallel} Chaos(H)) \setminus H$ is Roscoe's *lazy abstraction* [Ros97, Chapter 12], and CSP_H represents all processes with alphabet H.

Unfortunately, this definition suffers from the refinement paradox: there are processes that satisfy $FNDC$ but that have refinements that do not. Nondeterminism arises in models of systems for two main reasons. Often analysis is carried out upon *designs* of systems, rather than concrete implementations; in designs, nondeterminism often represents under-specification, which is resolved at a subsequent stage of the development: we should consider a design secure only if all ways of resolving the nondeterminism lead to secure implementations. Sometimes nondeterminism represents low-level details of a system that one chooses to abstract away from, e.g. scheduling: in such cases, one should consider a system secure only if all ways in which that nondeterminism could be resolved causes the system to behave securely.

In [Low07], I therefore considered the refinement-closure of FNDC. Process P satisfies *refinement-closed, failures non-deducibility on compositions*, written $RCFNDC(P)$, if $\forall Q \sqsupseteq P \bullet FNDC(Q)$. It turns out that this property can be captured as an binary failures predicate, as follows (see [Low07]):

$$\begin{array}{l}\forall (tr_1, X_1), (tr_2, X_2) \in failures(P) \bullet \\ \quad tr_1 \upharpoonright H \neq \langle\rangle \Rightarrow \forall l \bullet tr_2 = tr_1 \upharpoonright L \hat{} \langle l \rangle \Rightarrow l \notin X_1 \\ \qquad\qquad \wedge \forall l \bullet tr_2 \hat{} \langle l \rangle = tr_1 \upharpoonright L \Rightarrow l \notin X_2.\end{array}$$

2.3 Causation

In [ML07], we consider the question of whether an agent in a multi-user system, with alphabet A, can cause some event $e \notin A$ to occur. The obvious definition for non-causation, called *traces non-causation*, is:

$$\nexists tr \bullet tr \hat{} \langle e \rangle \in traces(P) \wedge (tr \setminus A) \hat{} \langle e \rangle \notin traces(P).$$

In other words, there's no trace tr after which e can happen, but such that if no events from A had happened, e would be impossible.

As with FNDC, this property turns out not to be refinement-closed. However, the refinement-closure can be expressed as an binary failures predicate, as follows:

$$\begin{array}{l}\forall (tr_1, X_1), (tr_2, X_2) \in failures(P) \bullet \\ \quad \forall s, t \bullet tr_1 = s \hat{} t \hat{} \langle e \rangle \wedge s \upharpoonright A \neq \langle\rangle \wedge tr_2 = s \setminus A \Rightarrow first(t \setminus A \hat{} \langle e \rangle) \notin X_2.\end{array}$$

2.4 Responsiveness

In [RSR04], Reed, Sinclair and Roscoe consider the responsiveness of interoperating components: whether a component Q when connected to a system P cannot cause P to deadlock, so that Q can be used as a plugin to P. Let both P and Q have alphabet J, so that they synchronise on all events. Then an obvious definition of responsiveness is

$$\forall tr \bullet (tr, J) \in failures(P \underset{J}{\parallel} Q) \Rightarrow (tr, J) \in failures(P).$$

Unfortunately, again this property is not refinement-closed. However, the authors show that the refinement-closure of the above property is:

$$\begin{array}{l}\forall (tr, X) \in failures(P) \bullet \\ \quad initsInJ(P/tr) - X \neq \{\} \Rightarrow (tr, initsInJ(P/tr) - X) \notin failures(Q), \\ \text{where } initsInJ(P/tr) \mathrel{\widehat{=}} \{x \mid tr \hat{} \langle x \rangle \in traces(P) \wedge x \in J\}.\end{array}$$

This property is equivalent to

$$\begin{array}{l}\forall (tr_1, X_1), (tr_2, X_2) \in failures(P) \bullet \\ \quad \forall (tr_3, X_3) \in failures(Q) \bullet \forall x \in J \bullet \\ \qquad tr_1 = tr_2 \hat{} \langle x \rangle \wedge tr_2 = tr_3 \Rightarrow (\exists y \in J \bullet y \notin X_2 \cup X_3) \vee x \notin X_3,\end{array}$$

which is a binary failures predicate on P.

3 n-copy refinement tests capture all n-ary failures predicates

In this section, for each n-ary relation R, we show how to construct a testing harness $Harness(\_)$ and specification $Spec$ such that

$$\forall P \bullet Prop_R(P) \Leftrightarrow Spec \sqsubseteq Harness(P).$$

We begin by showing that, without loss of generality, we can restrict ourselves to predicates R that satisfy Definitions 3.1 and 3.2, below.

Definition 3.1 We say that R is *subrefusal-closed* if:

$$R((tr_1, X_1), \ldots, (tr_n, X_n)) \wedge (\forall i \in 1 \,..\, n \bullet Y_i \subseteq X_i) \Rightarrow R((tr_1, Y_1), \ldots, (tr_n, Y_n)).$$

We write $R(tr_1, \ldots, tr_n)$ as a shorthand for $R((tr_1, \{\}), \ldots, (tr_n, \{\}))$; this captures that R allows the traces $tr_1, \ldots, tr_n$. Note that if R is subrefusal-closed, this is equivalent, to $\exists X_1, \ldots, X_n \bullet R((tr_1, X_1), \ldots, (tr_n, X_n))$.

Let $impossible_i(tr_1, \ldots, tr_n)$ be those events that cannot be appended onto the end of tr_i and allow R to be true:

$$impossible_i(tr_1, \ldots, tr_n) \mathrel{\widehat{=}} \{a \mid \neg R(tr_1, \ldots, tr_i \mathbin{\hat{}} \langle a \rangle, \ldots, tr_n)\}.$$

We omit the arguments of $impossible_i$ when clear from the context.

Definition 3.2 We say that R *allows the refusal of impossible events* if adding the elements of $impossible_i$ onto the i'th refusal set maintains R:

$$R((tr_1, X_1), \ldots, (tr_n, X_n)) \Rightarrow R((tr_1, X_1 \cup impossible_1), \ldots, (tr_n, X_n \cup impossible_n)).$$

R being subrefusal-closed and allowing the refusal of impossible events is very closely related to axioms F2 and F3 of the stable failures model.

Note that all the examples from Section 2 satisfy these conditions (although finding a subrefusal-closed version of responsiveness was non-trivial). In fact, in each case we have $impossible_1(tr_1, tr_2) = impossible_2(tr_1, tr_2) = \{\}$, for all tr_1 and tr_2.

If R does not satisfy the stable failures axioms, then we can replace it by the following relation: [1]

$$R^* \mathrel{\widehat{=}} \bigcup \{R' \mid R' \subseteq R \wedge R' \text{ is subrefusal-closed and allows the refusal of impossible events}\}.$$

The following two lemmas justify this.

Lemma 3.3 R^* is subrefusal-closed and allows the refusal of impossible events.

Proof: Direct from the definitions. □

Lemma 3.4 For all P, $Prop_R(P) \Leftrightarrow Prop_{R^*}(P)$.

[1] We consider an n-ary relation over failures to be a set of n-tuples of failures, in the standard way.

Proof: The right-to-left implication is immediate, since $R^* \subseteq R$.

Suppose $Prop_R(P)$. Consider the set

$$R' \mathrel{\widehat{=}} (failures(P))^n$$

(i.e. the n-fold cartesian product of $failures(P)$). Then R' is subrefusal-closed and allows the refusal of impossible events, since $failures(P)$ satisfies axioms F2 and F3 of the stable failures model. And $R' \subseteq R$, since $Prop_R(P)$. Hence $R' \subseteq R^*$ and so $Prop_{R^*}(P)$. □

So without loss of generality, assume that R respects the axioms. We now describe the construction of the testing harness.

We define the harness as follows:

$$Harness(P) \mathrel{\widehat{=}} c.1.P \;|||\; \ldots \;|||\; c.n.P.$$

For any failure (tr, X) of $Harness(P)$, we write $(tr, X) \downarrow c.i$ for the contribution of the ith copy of P, i.e. [2]

$$(tr, X) \downarrow c.i \mathrel{\widehat{=}} (tr \downarrow c.i,\ \{x \mid c.i.x \in X\}).$$

Then

$$failures(Harness(P)) = \{(tr, X) \mid tr \in \{|c|\}^* \wedge \forall\, i \in 1\,..\,n \bullet (tr, X) \downarrow c.i \in failures(P)\}.$$

We now construct an appropriate specification. We need to consider two cases. The uninteresting case is when $\neg R(\langle\rangle, \ldots, \langle\rangle)$, and so $Prop_R$ is unsatisfiable. We can take $Spec = d \rightarrow STOP$, for example, and then

$$Prop_R(P) \Leftrightarrow false \Leftrightarrow Spec \sqsubseteq Harness(P),$$

as required.

The more interesting case is where $R(\langle\rangle, \ldots, \langle\rangle)$ holds. We build a specification that keeps track of the traces communicated by each of the components, and only allows appropriate events and refusals.

After the components have performed traces $(tr_1, \ldots, tr_n)$, we will allow the specification to refuse subsets of the following *characteristic refusals*:

$$charRefs_R(tr_1, \ldots, tr_n) \mathrel{\widehat{=}} \{(X_1, \ldots, X_n) \mid R((tr_1, X_1), \ldots, (tr_n, X_n)) \wedge \forall\, i \in 1\,..\,n \bullet X_i \supseteq impossible_i(tr_1, \ldots, tr_n)\}.$$

We define the specification as follows:

$$Spec \mathrel{\widehat{=}} Spec(\langle\rangle, \ldots, \langle\rangle),$$

$$Spec(tr_1, \ldots, tr_n) \mathrel{\widehat{=}} \sqcap (X_1, \ldots, X_n) \in charRefs_R(tr_1, \ldots, tr_n) \bullet c?i?y : \Sigma - X_i \rightarrow Spec(tr_1, \ldots, tr_i \mathbin{\hat{}} \langle y \rangle, \ldots, tr_n).$$

Note that after trace tr, $Spec$ will be in state $Spec(tr \downarrow c.1, \ldots, tr \downarrow c.n)$.

Note also that if $R(tr_1, \ldots, tr_n)$ holds, then $R((tr_1, impossible_1), \ldots, (tr_n, impossible_n))$, since R allows the refusal of impossible events, and so the non-deterministic choice is over a non-empty set, containing at least $(impossible_1, \ldots, impossible_n)$.

---

[2] If tr is a trace then $tr \downarrow c.i$ represents the sequence of data transmitted over channel $c.i$ during tr.

We show that $R(tr_1, \ldots, tr_n)$ is indeed satisfied in every reachable state, by induction on the sum of the lengths of the traces. The base case holds by assumption. For the inductive step, suppose $R(tr_1, \ldots, tr_n)$, and pick $(X_1, \ldots, X_n) \in \mathit{charRefs}_R(tr_1, \ldots, tr_n)$ and $y \in \Sigma - X_i$. Then by definition of *charRefs*, $y \notin \mathit{impossible}_i$, so $R(tr_1, \ldots, tr_i \hat{} \langle y \rangle, \ldots, tr_n)$.

Conversely, if $R(tr_1, \ldots, tr_n)$ and $R(tr_1, \ldots, tr_i \hat{} \langle y \rangle, \ldots, tr_n)$ hold, then $\mathit{Spec}(tr_1, \ldots, tr_n)$ allows $c.i.y$ since $y \notin \mathit{impossible}_i$. Hence *Spec* allows trace tr precisely when $\forall\, tr' \leq tr \bullet R(tr' \downarrow c.1, \ldots, tr' \downarrow c.n)$.

It is then easy to see that $\mathit{Spec}(tr_1, \ldots, tr_n)$ allows the components to refuse precisely those $(X_1, \ldots, X_n)$ such that $R((tr_1, X_1), \ldots, (tr_n, X_n))$: each such tuple of refusals is included within $(X_1 \cup \mathit{impossible}_1, \ldots, X_n \cup \mathit{impossible}_n)$, which is a characteristic refusal since R allows the refusal of impossible events; and all tuples of refusals allowed by *Spec* satisfy R since R is subrefusal-closed. Hence

$$\begin{aligned}\mathit{failures}(\mathit{Spec}) = \{(tr, X) \mid\ & tr \in \{|c|\}^* \wedge R((tr, X) \downarrow c.1, \ldots, (tr, X) \downarrow c.n) \\ & \wedge\, \forall\, tr' \leq tr \bullet R(tr' \downarrow c.1, \ldots, tr' \downarrow c.n)\}.\end{aligned}$$

Hence, comparing the failures of *Spec* and $\mathit{Harness}(P)$,

$$\mathit{Prop}_R(P) \Leftrightarrow \mathit{Spec} \sqsubseteq_F \mathit{Harness}(P),$$

as required.

4 n-copy refinement tests capture only n-ary failures predicates

We now show that every n-copy refinement test corresponds to an n-ary failures predicate of the form $\mathit{Prop}_R(P)$. Consider an n-copy refinement test of the form $\mathit{Spec} \sqsubseteq H(P)$ where $H(P)$ is an arbitrary harness that contains n copies of P.

The crucial point to observe is that for each CSP operator, each failure of the resulting process results from (at most) one failure of each component. Therefore, each failure of $H(P)$ results from (at most) one failure of each copy, i.e., a total of (at most) n failures of P. Therefore, the set of failures of $H(P)$ will be of the form

$$\begin{aligned}\{(tr, X) \mid\ & \exists (tr_1, X_1), \ldots, (tr_n, X_n) \in \mathit{failures}(P) \bullet \\ & S((tr, X), (tr_1, X_1), \ldots, (tr_n, X_n))\},\end{aligned}$$

where S captures the semantic equations corresponding to how the harness is constructed.

Hence $H(P)$ will refine *Spec* iff

$$\begin{aligned}&\forall (tr_1, X_1), \ldots, (tr_n, X_n) \in \mathit{failures}(P) \bullet \\ &\quad \forall (tr, X) \bullet S((tr, X), (tr_1, X_1), \ldots, (tr_n, X_n)) \Rightarrow (tr, X) \in \mathit{failures}(\mathit{Spec}).\end{aligned}$$

This is of the form $\mathit{Prop}_R(P)$ if we define $R((tr_1, X_1), \ldots, (tr_n, X_n))$ to be the predicate expressed by the second line of the above formula.

5 Making it feasible

The construction in Section 3 produced a specification process that will, in general, be infinite state. In this section, we describe how to reduce the refinement check to

one that is finite state, at least for finite state P, and so can be carried out using a model checker such as FDR.

What we will do is put a scheduler process in parallel with the harness and specification processes from the previous section. That is, we produce a scheduler process *Sched* and define

$$Harness'(P) \mathrel{\widehat{=}} Sched \underset{\{|c|\}}{\parallel} Harness(P),$$
$$Spec' \mathrel{\widehat{=}} Sched \underset{\{|c|\}}{\parallel} Spec.$$

For later convenience, we will allow the alphabet of *Sched* to contain events outside $\{|c|\}$. We will then consider refinements of the form

$$Spec' \sqsubseteq_F Harness'(P).$$

In many cases, we will be able to calculate a finite state process that is equivalent to $Spec'$. Further, the use of the scheduler will normally mean that the state space of $Harness'(P)$ is considerably smaller than that of $Harness(P)$.

We present, in Definition 5.1, sufficient conditions on *Sched* for the above refinement to hold precisely when $Prop_R(P)$ is true.

Definition 5.1 Let *Sched* be a deterministic process whose traces are some set S such that:

(i) For all traces $tr_1, \ldots, tr_n$ corresponding to failures where R does not hold, S includes a corresponding trace, i.e.:

$$\forall tr_1, \ldots, tr_n, X_1, \ldots, X_n \bullet \neg R((tr_1, X_1), \ldots, (tr_n, X_n)) \Rightarrow \exists tr \in S \bullet \forall i \in \{1 \,..\, n\} \bullet tr \downarrow c.i = tr_i.$$

In other words, *Sched* allows the harness to get into a state where each falsity of R could be demonstrated.

(ii) For all failures $(tr \downarrow c.1, X_1), \ldots, (tr \downarrow c.n, X_n)$ where R does not hold, *Sched* allows the harness to demonstrate the falsity of R; it does this by allowing all events in some (possibly different) sets $X'_1 \subseteq X_1, \ldots, X'_n \subseteq X_n$ such that $\neg R((tr \downarrow c.1, X'_1), \ldots, (tr \downarrow c.n, X'_n))$.

$$tr \in S \wedge \neg R((tr \downarrow c.1, X_1), \ldots, (tr \downarrow c.n, X_n)) \Rightarrow \exists X'_1 \subseteq X_1, \ldots, X'_n \subseteq X_n \bullet \neg R((tr \downarrow c.1, X'_1), \ldots, (tr \downarrow c.n, X'_n)) \wedge \forall i \in 1 \,..\, n \bullet \forall a \in X'_i \bullet tr^\frown\langle c.i.a \rangle \in S.$$

Intuitively, it is the events in the X'_i that demonstrate the falsity of R: for example, in the case of determinism, $R((tr_1, X_1), (tr_2, X_2)$ is false whenever $tr_1 = tr_2{}^\frown\langle a \rangle$ and $a \in X_2$ for some a; so we can take $X'_2 = \{a\}$ and $X'_1 = \{\}$. If $Harness'(P)$ refuses the events of the X'_i then that refusal can be attributed to P rather than the scheduler.

From now on, we assume that we have fixed a suitable scheduler *Sched*, as in the above definition; in Section 6, we show that this is straightforward for the examples of Section 2. As *Sched* is deterministic with traces S,

$$failures(Sched(S)) = \{(tr, X) \mid tr \in S \wedge \forall x \in X \bullet tr^\frown\langle x \rangle \notin S\}.$$

Let $Harness'(P)$ and $Spec'(P)$ be as above. The following theorem proves the above claim.

Theorem 5.2 $Prop_R(P) \Leftrightarrow Spec' \sqsubseteq_F Harness'(P)$.

Proof: If $\neg Prop_R(P)$ then there exist $f_1 = (tr_1, X_1), \ldots, f_n = (tr_n, X_n) \in failures(P)$ such that $\neg R(f_1, \ldots, f_n)$. By condition (i) of Definition 5.1, there is some $tr \in S$ such that $\forall\, i \in 1\,..\,n \bullet tr \downarrow c.i = tr_i$. Let $X'_1, \ldots, X'_n$ be as in condition (ii) of Definition 5.1, and let $f'_i \mathrel{\widehat{=}} (tr_i, X'_i)$, for $i \in 1\,..\,n$; so $\neg R(f'_i, \ldots, f'_n)$. Then $f'_i \in failures(P)$, for $i \in 1..n$, by Axiom F2. Let $X \mathrel{\widehat{=}} \{c.i.x \mid i \in 1..n, x \in X'_i\}$ and $f \mathrel{\widehat{=}} (tr, X)$. Then $f \in failures(Harness'(P))$.

Now, $(tr \upharpoonright \{|c|\}, X) \notin failures(Spec)$, by construction. Also, for each $i \in 1..n$, for every $a \in X'_i$, we have $tr^\frown\langle c.i.a\rangle \in S$, by definition of the X'_i. Hence $(tr, \{c.i.a\}) \notin failures(Sched)$, and so $f \notin failures(Spec')$. Hence $Spec' \not\sqsubseteq_F Harness'(P)$.

Conversely, if $Spec' \not\sqsubseteq_F Harness'(P)$ then there is some $f = (tr, X) \in failures(Harness'(P))$ such that $f \notin failures(Spec')$. Then $tr \in traces(Sched)$ and so $tr \in S$. We show that $\neg R(f \downarrow c.1, \ldots, f \downarrow c.n)$. Suppose, otherwise. Then $(tr \upharpoonright \{|c|\}, X) \in failures(Spec)$, by construction. Now, $tr \in S$ so $(tr, \{\}) \in failures(Sched)$ and so $f \in failures(Spec')$, giving a contradiction.

Hence $\neg R(f \downarrow c.1, \ldots, f \downarrow c.n)$. Let $X'_1, \ldots X'_n$ be as in condition (ii) of Definition 5.1. Let $f_i \mathrel{\widehat{=}} (tr \downarrow c.i, X'_i)$, for $i = 1\,..\,n$. Then $\neg R(f_1, \ldots, f_n)$. Now, $X' \mathrel{\widehat{=}} \{c.i.x \mid i \in 1\,..\,n,\ x \in X'_i\} \subseteq X$, so $(tr, X') \in failures(Harness'(P))$. $Sched$ is not refusing any of the events of X'_i, by definition of the X'_i; hence those events must be refused by $c.i.P$, so $f_i \in failures(P)$, for $i = 1\,..\,n$. Hence $\neg Prop_R(P)$. □

6 Examples

In this section we use the technique of the previous section to produce finite-state refinement checks for some of the properties from Section 2.

6.1 Determinism

Recall that for determinism we have

$$R((tr_1, X_1), (tr_2, X_2)) \mathrel{\widehat{=}} \forall\, a \bullet tr_1 = tr_2{}^\frown\langle a\rangle \Rightarrow a \notin X_2.$$

We define a scheduler that alternates between $c.1$ and $c.2$, performing the same event on each:

$$Sched \mathrel{\widehat{=}} c.1?x \rightarrow c.2.x \rightarrow Sched,$$

Note that $Sched$ is deterministic, and satisfies the conditions of Definition 5.1: for condition (i), note that $Sched$ includes a trace corresponding to all cases where R is false (i.e., where $tr \downarrow c.1 = tr \downarrow c.2^\frown\langle a\rangle$, for some a); for condition (ii), if $tr \downarrow c.1 = tr_2 \downarrow c.2^\frown\langle a\rangle$, we can take $X'_1 = \{\}$ and $X'_2 = \{a\}$, and note that $Sched$ allows $c.2.a$.

Note that

$$charRefs(tr, tr) = (\mathbb{P}\,\Sigma)^2,$$
$$charRefs(tr^\frown\langle a\rangle, tr) = \{(X_1, X_2) \mid a \notin X_2\}.$$

Hence we can calculate a finite form for $Spec \parallel Sched$, as follows (we omit the synchronisation set $\{|c|\}$ from the parallel operator, for brevity):

$$
\begin{aligned}
& Spec(tr, tr) \parallel Sched \\
= & \left\langle \text{step laws: } Spec \text{ chooses over all pairs of refusals} \right\rangle \\
& \sqcap X \in \mathbb{P}\,\Sigma \bullet c.1?y : \Sigma - X \to (Spec(tr^\frown\langle y\rangle, tr) \parallel c.2.y \to Sched) \\
= & \left\langle \begin{array}{l} \text{by consideration of initial refusals;} \\ \text{step laws: } Spec \text{ does not allow } c.2.y \text{ to be refused} \end{array} \right\rangle \\
& c.1?y \to c.2.y \to (Spec(tr^\frown\langle y\rangle, tr^\frown\langle y\rangle) \parallel Sched) \sqcap STOP.
\end{aligned}
$$

Hence for all tr, $Spec(tr, tr) \parallel Sched$ is equivalent to

$$Spec' = c.1?y \to c.2.y \to Spec' \sqcap STOP.$$

(Formally, this is by the Unique Fixed Point rule [Ros97, Section 1.3].) To summarise, P is deterministic if

$$Spec' \sqsubseteq_F Sched \parallel (c.1.P \;|||\; c.2.P).$$

This is Lazić's test from [Laz99].

6.2 RCFNDC

Recall that for $RCFNDC$ we have

$$
\begin{aligned}
R((tr_1, X_1), (tr_2, X_2)) \mathrel{\widehat{=}} tr_1 \upharpoonright H \neq \langle\rangle \Rightarrow \forall\, l \bullet tr_2 = tr_1 \upharpoonright L^\frown\langle l\rangle \Rightarrow l \notin X_1 \\
\wedge\, \forall\, l \bullet tr_2{}^\frown\langle l\rangle = tr_1 \upharpoonright L \Rightarrow l \notin X_2.
\end{aligned}
$$

We define the scheduler to force the same L events to be performed on $c.1$ and $c.2$, in either order, and also allow H events on $c.1$:

$$
\begin{aligned}
Sched \mathrel{\widehat{=}} & c.1?h : H \to Sched \\
& \Box\, c.1?l : L \to Sched_1(l) \\
& \Box\, c.2?l : L \to Sched_2(l), \\
Sched_1(l) \mathrel{\widehat{=}} & c.2.l \to Sched, \\
Sched_2(l) \mathrel{\widehat{=}} & c.1.l \to Sched.
\end{aligned}
$$

Note that $Sched$ is deterministic, as required. Further, when $R((tr_1, X_1), (tr_2, X_2))$ is false, we have $tr_2 = tr_1 \upharpoonright L^\frown\langle l\rangle$ or $tr_2{}^\frown\langle l\rangle = tr_1 \upharpoonright L$ for some l; it is easy to construct a corresponding trace of $Sched$ where the equal events of $tr_1 \upharpoonright L$ and tr_2 are performed consecutively, in either order. Also, condition (ii) of Definition 5.1 is satisfied: a failure of the first disjunct in the definition of RCFNDC would correspond to a trace that would lead to the state $Sched_2(l)$, which allows $c.1.l$, as required; the second disjunct in the definition of RCFNDC is similar.

We now consider all states reachable from $Sched \parallel Spec(\langle\rangle, \langle\rangle)$. Throughout the following, suppose $tr_1 \upharpoonright L = tr_2$. We need to consider five cases; in the first three cases, R is true for all X_1 and X_2 with the given traces, so $Spec$ nondeterministically chooses over *all* values of X_1 and X_2. We can calculate as follows.

$$
\begin{aligned}
& Sched \parallel Spec(tr_1, tr_2) = \\
& \left(\begin{array}{l} c.1?h : H \to (Sched \parallel Spec(tr_1{}^\frown\langle h\rangle, tr_2)) \\ \Box\, c.1?l : L \to (Sched_1(l) \parallel Spec(tr_1{}^\frown\langle l\rangle, tr_2)) \\ \Box\, c.2?l : L \to (Sched_2(l) \parallel Spec(tr_1, tr_2{}^\frown\langle l\rangle)) \end{array} \right) \sqcap STOP.
\end{aligned}
$$

If $tr_1 \upharpoonright H = \langle\rangle$:

$$\begin{aligned} &Sched_1(l) \parallel Spec(tr_1{}^\frown\langle l\rangle, tr_2) = \\ &\quad c.2.l \rightarrow (Sched \parallel Spec(tr_1{}^\frown\langle l\rangle, tr_2{}^\frown\langle l\rangle)) \sqcap STOP, \\ &Sched_2(l) \parallel Spec(tr_1, tr_2{}^\frown\langle l\rangle) = \\ &\quad c.1.l \rightarrow (Sched \parallel Spec(tr_1{}^\frown\langle l\rangle, tr_2{}^\frown\langle l\rangle)) \sqcap STOP. \end{aligned}$$

If $tr_1 \upharpoonright H \neq \langle\rangle$ then $Spec$ can't refuse respectively $c.2.l$ and $c.1.l$ in the following cases:

$$\begin{aligned} &Sched_1(l) \parallel Spec(tr_1{}^\frown\langle l\rangle, tr_2) = c.2.l \rightarrow (Sched \parallel Spec(tr_1{}^\frown\langle l\rangle, tr_2{}^\frown\langle l\rangle)), \\ &Sched_2(l) \parallel Spec(tr_1, tr_2{}^\frown\langle l\rangle) = c.1.l \rightarrow (Sched \parallel Spec(tr_1{}^\frown\langle l\rangle, tr_2{}^\frown\langle l\rangle)). \end{aligned}$$

So we can define $Spec'$ as follows; the two clauses correspond to $Sched \parallel Spec(tr_1, tr_2)$ with $tr_1 \upharpoonright H \neq \langle\rangle$ and $tr_1 \upharpoonright H \neq \langle\rangle$, respectively; the other cases have been absorbed into these clauses.

$$\begin{aligned} Spec' &\mathrel{\widehat{=}} \begin{pmatrix} c.1?h : H \rightarrow Spec'' \\ \Box\ c.1?l : L \rightarrow (c.2.l \rightarrow Spec' \sqcap STOP) \\ \Box\ c.2?l : L \rightarrow (c.1.l \rightarrow Spec' \sqcap STOP) \end{pmatrix} \sqcap STOP, \\ Spec'' &\mathrel{\widehat{=}} \begin{pmatrix} c.1?h : H \rightarrow Spec'' \\ \Box\ c.1?l : L \rightarrow c.2.l \rightarrow Spec'' \\ \Box\ c.2?l : L \rightarrow c.1.l \rightarrow Spec'') \end{pmatrix} \sqcap STOP. \end{aligned}$$

This is equivalent to, but defined rather differently from, the test in [Low07] [3] .

6.3 Causation

Recall that for causation we have

$$\begin{aligned} &R((tr_1, X_1), (tr_2, X_2)) \mathrel{\widehat{=}} \\ &\quad \forall\, s, t \bullet tr_1 = s{}^\frown t{}^\frown\langle e\rangle \wedge s \upharpoonright A \neq \langle\rangle \wedge tr_2 = s \setminus A \Rightarrow \mathit{first}(t \setminus A{}^\frown\langle e\rangle) \notin X_2. \end{aligned}$$

Without loss of generality, we may assume that t, in the above equation, does not start with an event from A: any such event could be transferred onto the end of s.

Let $B \mathrel{\widehat{=}} \alpha P - A$. We introduce a fresh event, *ping*, to indicate the start of t; this event also makes it easier to define the scheduler as a deterministic process, as required. The scheduler alternates the same B events on $c.1$ and $c.2$, and also allows A events on $c.1$; these events correspond to s in the definition of R. After at least one A event has occurred (state $Sched'$) we allow a *ping* after a $c.1.b$ event, that b being $\mathit{first}(t \setminus A{}^\frown\langle e\rangle)$. We then (state $Sched''(b)$) allow arbitrary events on $c.1$, corresponding to the rest of t, and also allow $c.2.b$.

$$\begin{aligned} Sched &\mathrel{\widehat{=}} c.1?a : A \rightarrow Sched' \ \Box\ c.1?b : B \rightarrow c.2.b \rightarrow Sched, \\ Sched' &\mathrel{\widehat{=}} c.1?a : A \rightarrow Sched' \\ &\qquad \Box\ c.1?b : B \rightarrow (c.2.b \rightarrow Sched' \ \Box\ ping \rightarrow Sched''(b)) \\ Sched''(b) &\mathrel{\widehat{=}} c.1?d : \alpha P \rightarrow Sched''(b) \ \Box\ c.2.b \rightarrow STOP. \end{aligned}$$

[3] The test in [Low07] schedules the two processes in a slightly different way; further, it allows additional H events between corresponding $c.1.l$ and $c.2.l$ events, and so has a slightly larger search space.

Note that $R((tr_1, X_1), (tr_2, X_2))$ is false only when, for some s and t, $tr_1 = s \frown t \frown \langle e \rangle \wedge s \upharpoonright A \neq \langle\rangle \wedge tr_2 = s \setminus A$. Such a trace is reflected by the traces of *Sched* that end in state $Sched''(b)$. Note also that condition (ii) of Definition 5.1 is satisfied. The only non-trivial case to check is after a trace as above; taking $X_2' = \{b\}$, we can check that $c.2.b$ is available both in state $Sched''(b)$ and directly after the $c.1.b$ event in state $Sched'$.

In the appendix we show that $Sched \parallel Spec$ is equivalent to the following process.

$$Spec_1 \mathrel{\widehat{=}} (c.1?a : A \to Spec_2 \;\Box\; c.1?b : B \to (c.2.b \to Spec_1 \sqcap STOP)) \sqcap STOP,$$

$$Spec_2 \mathrel{\widehat{=}} \left(\begin{array}{l} c.1?a : A \to Spec_2 \\ \Box\; c.1?b : B \to \text{if } b \neq e \\ \qquad\qquad \text{then } c.2.b \to Spec_2 \triangleright ping \to Spec_3(b, b) \\ \qquad\qquad \text{else } c.2.e \to Spec_2 \;\Box\; ping \to Spec_3(e, e) \end{array}\right) \sqcap STOP,$$

$$\begin{aligned} Spec_3(b, c) \mathrel{\widehat{=}} \text{if } c \neq e \\ \text{then } (c.1?d : \alpha P \to Spec_3(b, d) \;\Box\; c.2.b \to STOP) \sqcap STOP \\ \text{else } c.1?d : \alpha P \to Spec_3(b, d) \triangleright c.2.b \to STOP. \end{aligned}$$

This is similar to the test from [ML07].

6.4 Responsiveness

Recall that for responsiveness we have

$$\begin{aligned} R((tr_1, X_1), (tr_2, X_2)) \mathrel{\widehat{=}} \\ \forall (tr_3, X_3) \in failures(Q) \bullet \forall x \in J \bullet \\ tr_1 = tr_2 \frown \langle a \rangle \wedge tr_2 = tr_3 \Rightarrow (\exists y \in J \bullet y \notin X_2 \cup X_3) \vee x \notin X_3. \end{aligned}$$

The techniques we have described so far talk about a single process P. However, R in this case talks about both P and Q. It turns out that the techniques extend directly to predicates of the form

$$\forall f_1 \in failures(P_1), \ldots, f_n \in failures(P_n) \bullet R(f_1, \ldots, f_n),$$

where some of the P_i may be the same, by considering a harness of the form [4]

$$(c.1.P_1 \;|||\; \ldots \;|||\; c.n.P_n) \parallel Sched.$$

We can replace the above R, above, by a ternary relation that includes failures of Q as its third component:

$$\begin{aligned} R'((tr_1, X_1), (tr_2, X_2), (tr_3, X_3)) \mathrel{\widehat{=}} \\ \forall x \in J \bullet tr_1 = tr_2 \frown \langle a \rangle \wedge tr_2 = tr_3 \Rightarrow (\exists y \in J \bullet y \notin X_2 \cup X_3) \vee x \notin X_3. \end{aligned}$$

We can then consider a harness of the form:

$$Harness(P, Q) = (c.1.P \;|||\; c.2.P \;|||\; c.3.Q) \parallel Sched.$$

---

[4] One way to reduce this more general harness to the previous type is to define

$$\begin{aligned} AnyP &= choose?i : 1 \,..\, n \to P_i, \\ Sched' &= c.1.choose.1.P_1 \to \ldots \to c.n.choose.n.P_n \to Sched, \\ Harness' &= (c.1.AnyP \;|||\; \ldots \;|||\; c.n.AnyP) \parallel Sched'. \end{aligned}$$

It is then clear that the behaviours of $Harness'$ are the same as those of the above harness, except with the addition of the *choose* events.

Note that R' is false only when $tr_1 = tr_2 \hat{} \langle a \rangle$ and $tr_2 = tr_3$, so we mainly force the processes to perform the same events. However, after such traces, many different refusals X_2 and X_3 make R' false, so, following condition (ii) of Definition 5.1, we allow *all* events on $c.2$ and $c.3$, but allow the processes to proceed only if the events match:

$$\begin{aligned} Sched &\mathrel{\widehat{=}} c.1?x \to Sched'(x), \\ Sched'(x) &\mathrel{\widehat{=}} c.2?y \to x = y \,\&\, c.3.x \to Sched \;\Box\; c.3?y \to x = y \,\&\, c.2.x \to Sched. \end{aligned}$$

Note that whenever *Sched* returns to its initial state, the same events have been performed on the three channels.

In the appendix, we show that $Sched \parallel Spec$ is equivalent to $Spec_0$, below:

$$\begin{aligned} Spec_0 &\mathrel{\widehat{=}} c.1?x \to Spec_1(x) \sqcap STOP, \\ Spec_1(x) &\mathrel{\widehat{=}} (c.2?y \to Spec_2(x,y) \;\Box\; c.3?y \to Spec_3(x,y)) \\ &\qquad \rhd \begin{pmatrix} \bigsqcap y \in J \bullet (c.2.y \to Spec_2(x,y) \;\Box\; c.3.y \to Spec_3(x,y)) \\ \sqcap\; c.3.x \to (c.2.x \to Spec_0 \sqcap STOP) \end{pmatrix}, \\ Spec_2(x,y) &\mathrel{\widehat{=}} x = y \,\&\, c.3.x \to Spec_0 \sqcap STOP, \\ Spec_3(x,y) &\mathrel{\widehat{=}} x = y \,\&\, c.2.x \to Spec_0 \sqcap STOP. \end{aligned}$$

The refinement test from [RSR04] is somewhat different from this: there Q is synchronised directly with the second copy of P so as to combine their refusals; further, the clause "$x \notin X_3$" in the definition of R' is dealt with using a clever trick on the right hand side of the refinement check, rather than within the specification (in state $Spec_1(x)$, above).

7 Discussion

In this paper we have shown that refinement checks of the form $Spec \sqsubseteq Harness(P)$, where $Harness(P)$ runs n copies of P, capture precisely those predicates of the form $\forall f_1, \ldots, f_n \in failures(P) \bullet R(f_1, \ldots, f_n)$. Further, we have shown how a finite-state refinement check can be calculated in many common cases.

All of the examples we have considered have been *binary* failures predicates on P, i.e., have $n = 2$ (the question of responsiveness is ternary when Q's behaviour is included). A somewhat artificial example of a non-binary failures predicate is "after every trace, at most two different events are possible":

$$\begin{aligned} &\forall (tr_1, X_1), (tr_2, X_2), (tr_3, X_3) \in failures(P) \bullet \forall tr, a, b, c \bullet \\ &\quad tr_1 = tr \hat{} \langle a \rangle \wedge tr_2 = tr \hat{} \langle b \rangle \wedge tr_3 = tr \hat{} \langle c \rangle \Rightarrow a = b \vee a = c \vee b = c. \end{aligned}$$

A corresponding refinement test can be derived as for the earlier examples.

It is interesting to ask why the natural examples are all binary. All of the examples consider whether P behaves in the same way in two different circumstances: determinism asks whether P *always* behaves in the same way; RCFNDC asks whether P behaves in the same way when High is active or inactive; causation asks whether an event e becomes available as the result of action in A; responsiveness asks whether P has the same deadlocking behaviours when Q is plugged in as when P is allowed to communicate freely. It seems fairly clear that one needs to consider two behaviours of P to decide such properties. However, it seems very

difficult to find interesting properties that require one to consider *more* than two behaviours.

It is also interesting to note that most of the predicates (all except determinism) are the refinement closures of more basic properties. It seems that properties of the form considered in the previous paragraph naturally give rise to predicates that are not refinement-closed, and so one normally needs to consider the refinement closure. This question deserves further study. Calculating the refinement-closure of such predicates is not straightforward, so it would be useful to have some general techniques.

It would be interesting to try to extend the work of this paper to capture more general predicates over multiple failures of a process. Following the above discussion, sensible predicates to consider would be those of the form

$$\forall\, f \in \mathit{failures}(P) \bullet \exists\, f' \in \mathit{failures}(P) \bullet R(f, f').$$

Of course, such predicates are not, in general, refinement-closed, so any refinement check would need to use a copy of P on the *left* hand side of the refinement.

Acknowledgements

I would like to thank the anonymous referees for useful comments on this paper.

References

[FG95] Riccardo Focardi and Roberto Gorrieri. A classification of security properties. *Journal of Computer Security*, 1995.

[Foc96] Riccardo Focardi. Comparing two information flow security properties. In *Proceedings of 9th IEEE Computer Security Foundations Workshop*, pages 116–122, 1996.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[Laz99] Ranko Lazić. *A Semantic Study of Data Independence with Applications to Model Checking*. D.Phil., Oxford University, 1999.

[Low07] Gavin Lowe. On information flow and refinement-closure. In *Proceedings of the Workshop on Issues in the Theory of Security (WITS '07)*, 2007.

[ML07] Toby Murray and Gavin Lowe. Authority analysis for least privilege environments. In *To appear in Proceedings of Foundations of Computer Security and Automated Reasoning for Security Protocol Analysis (FCS-ARSPA'07)*, 2007.

[Ros94] A. W. Roscoe. Model-checking CSP. In *A Classical Mind, Essays in Honour of C. A. R. Hoare*. Prentice-Hall, 1994.

[Ros97] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1997.

[Ros05] A. W. Roscoe. On the expressive power of CSP refinement. *Formal Aspects of Computing*, 17(2):93–112, 2005.

[RSR04] J. N. Reed, J. E. Sinclair, and A. W. Roscoe. Responsiveness of interoperating components. *Formal Aspects of Computing*, 16:394–411, 2004.

A Details of derivations

In this appendix we give details of the derivations of a couple of the specification processes from Section 6.

A.1 Causation

We consider all the reachable states of $Sched \parallel Spec$ from Section 6.3. Firstly, suppose $tr_2 = tr_1 \setminus A,\ tr_1 \upharpoonright A = \langle\rangle$ and $b \in B$. Then in both the following cases, *Spec* allows all refusals.

$$Sched \parallel Spec(tr_1, tr_2) = \left(\begin{array}{l} c.1?a : A \to (Sched' \parallel Spec(tr_1{}^\frown\langle a\rangle, tr_2)) \\ \Box\ c.1?b : B \to (c.2.b \to Sched \parallel Spec(tr_1{}^\frown\langle b\rangle, tr_2)) \end{array}\right) \sqcap STOP,$$

$$c.2.b \to Sched \parallel Spec(tr_1{}^\frown\langle b\rangle, tr_2) = c.2.b \to (Sched \parallel Spec(tr_1{}^\frown\langle b\rangle, tr_2{}^\frown\langle b\rangle)) \sqcap STOP.$$

Now suppose $tr_2 = tr_1 \setminus A,\ tr_1 \upharpoonright A \neq \langle\rangle$, $b \in B$ and $b \neq e$. In the first two cases below, *Spec* allows all refusals, but in the third, *Spec* does not allow $c.2.e$ to be refused.

$$Sched' \parallel Spec(tr_1, tr_2) = \left(\begin{array}{l} c.1?a : A \to (Sched' \parallel Spec(tr_1{}^\frown\langle a\rangle, tr_2)) \\ \Box\ c.1?b : B \to (c.2.b \to Sched' \ \Box\ ping \to Sched''(b) \parallel Spec(tr_1{}^\frown\langle b\rangle, tr_2)) \end{array}\right) \sqcap STOP,$$

$$c.2.b \to Sched' \ \Box\ ping \to Sched''(b) \parallel Spec(tr_1{}^\frown\langle b\rangle, tr_2) = c.2.b \to (Sched' \parallel Spec(tr_1{}^\frown\langle b\rangle, tr_2{}^\frown\langle b\rangle)) \triangleright ping \to (Sched''(b) \parallel Spec(tr_1{}^\frown\langle b\rangle, tr_2)),$$

$$c.2.e \to Sched' \ \Box\ ping \to Sched''(b) \parallel Spec(tr_1{}^\frown\langle e\rangle, tr_2) = c.2.e \to (Sched' \parallel Spec(tr_1{}^\frown\langle e\rangle, tr_2{}^\frown\langle e\rangle)) \ \Box\ ping \to (Sched''(e) \parallel Spec(tr_1{}^\frown\langle e\rangle, tr_2)).$$

Finally, suppose $tr_2 < tr_1 \setminus A$, b is the first event in $tr_1 \setminus A$ after tr_2, $tr_1 \upharpoonright A \neq \langle\rangle$; in the "then" branch, below, *Spec* allows all refusals, but in the "else" branch, *Spec* doesn't allow $c.2.b$ to be refused.

$$Sched''(b) \parallel Spec(tr_1, tr_2) = \text{if } last\, tr_1 \neq e \text{ then } \left(\begin{array}{l} c.1?d : \alpha P \to (Sched''(b) \parallel Spec(tr_1{}^\frown\langle d\rangle, tr_2)) \\ \Box\ c.2.b \to STOP \end{array}\right) \sqcap STOP \text{ else } c.1?d : \alpha P \to (Sched''(b) \parallel Spec(tr_1{}^\frown\langle d\rangle, tr_2)) \triangleright c.2.b \to STOP.$$

We can therefore use the specification $Spec_1$ from Section 6.3 (the processes $Spec_1$, $Spec_2$ and $Spec_3(b, c)$ correspond, respectively, to the first, third, and sixth processes above, with $c = \text{last}\, tr_1$ in the final case).

A.2 Responsiveness

We consider all reachable states of $Sched \parallel Spec(\langle\rangle, \langle\rangle, \langle\rangle)$ from Section 6.4. In the following state, *Spec* allows all refusals.

$$Sched \parallel Spec(tr, tr, tr) = c.1?x \to (Sched'(x) \parallel Spec(tr{}^\frown\langle x\rangle, tr, tr)) \sqcap STOP.$$

In the following state, *Spec* cannot allow refusals X_2 and X_3 such that $x \in X_3$ and $X_2 \cup X_3 \supseteq J$.

$$
\begin{array}{l}
\quad Sched'(x) \parallel Spec(tr^\frown\langle x\rangle, tr, tr) \\
= \left\langle \text{step laws} \right\rangle \\
\quad \bigsqcap X_2, X_3 \in \mathbb{P}\, J,\ x \notin X_3 \vee X_2 \cup X_3 \subset J \bullet \\
\qquad c.2?y : J - X_2 \to (x = y \,\&\, c.3.x \to Sched \parallel Spec(tr^\frown\langle x\rangle, tr^\frown\langle y\rangle, tr)) \\
\qquad \Box\ c.3?y : J - X_3 \to (x = y \,\&\, c.2.x \to Sched \parallel Spec(tr^\frown\langle x\rangle, tr, tr^\frown\langle y\rangle)) \\
= \left\langle \text{by consideration of initial refusals} \right\rangle \\
\quad \begin{pmatrix} c.2?y \to (x = y \,\&\, c.3.x \to Sched \parallel Spec(tr^\frown\langle x\rangle, tr^\frown\langle y\rangle, tr)) \\ \Box\ c.3?y \to (x = y \,\&\, c.2.x \to Sched \parallel Spec(tr^\frown\langle x\rangle, tr, tr^\frown\langle y\rangle)) \end{pmatrix} \\
\quad \triangleright \\
\quad \left(\begin{array}{l} \bigsqcap y \in J \bullet \begin{pmatrix} c.2.y \to (x = y \,\&\, c.3.x \to Sched \parallel Spec(tr^\frown\langle x\rangle, tr^\frown\langle y\rangle, tr)) \\ \Box\ c.3.y \to (x = y \,\&\, c.2.x \to Sched \parallel Spec(tr^\frown\langle x\rangle, tr, tr^\frown\langle y\rangle)) \end{pmatrix} \\ \sqcap\ c.3.x \to (c.2.x \to Sched \parallel Spec(tr^\frown\langle x\rangle, tr, tr^\frown\langle x\rangle)) \end{array} \right)
\end{array}
$$

In the following case, *Spec* allows all refusals

$$
\begin{array}{l}
x = y \,\&\, c.3.x \to Sched \parallel Spec(tr^\frown\langle x\rangle, tr^\frown\langle y\rangle, tr) = \\
\qquad x = y \,\&\, c.3.x \to (Sched \parallel Spec(tr^\frown\langle x\rangle, tr^\frown\langle x\rangle, tr^\frown\langle x\rangle)) \sqcap STOP,
\end{array}
$$

and similarly with the roles of $c.2$ and $c.3$ reversed, and without the initial guard.

We therefore obtain the specification $Spec_0$ from Section 6.4.