

Electronic Notes in Theoretical Computer Science 250 (2009) 153–170

[www.elsevier.com/locate/entcs](http://www.elsevier.com/locate/entcs)

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](mailto: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 s.how that such refinement checks capture precisely those predicates

of the form *∀ f*1*,..., fn ∈ failures*(*P* ) *R*(*f*1*,..., fn*) 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.

# 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

*Spec ±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 [[8](#_bookmark23), Section 3.3], i.e., satisfiable predicates of the form

*∀ f ∈ failures*(*P* ) . *R*(*f* )

for some unary relation *R* over stable failures.

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

*Spec ±F Harness*(*P* )*,*

where *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*.

1571-0661 © 2009 Elsevier B.V.Open access under [CC BY-NC-ND license](http://creativecommons.org/licenses/by-nc-nd/3.0/).

doi:10.1016/j.entcs.2009.08.011

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*

(*P* ) = *∀ f ,...,f ∈ failures*(*P* ) . *R*(*f ,...,f* )*,*

*R* ^ 1 *n* 1 *n*

for *n*-ary relations *R* over stable failures. We call such predicates *n-ary failures*

*predicates*. In Section [2](#_bookmark0), we give some examples of *n*-ary predicates that have appeared in the literature.

In Section [3](#_bookmark1), we show that every *n*-ary failures predicate can be captured as an *n*-copy refinement test. In Section [4](#_bookmark6) 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](#_bookmark7), 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 [[7](#_bookmark22)]. We illustrate the technique on examples in Section [6](#_bookmark11): 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* : diver- gent 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 [[9](#_bookmark24)] considers the expressive power of CSP refinement checks more gen- erally. He considers which predicates over the failures-divergences model may be captured by refinement checks of the form *F* (*P* ) *± 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 [[8](#_bookmark23)]) 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 *Spec ± 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 re- finement checks. He conjectures, in passing, the results of Sections [3](#_bookmark1) and [4](#_bookmark6) 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 [[3](#_bookmark18),[8](#_bookmark23)].

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 commu- nication 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 can- not communicate with its environment. The process *a → 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 → P* (*a*) offers its environment the events from the set *{c.a | a ∈ A}*; if the event *c.a* is performed, the process then acts like *P* (*a*). The process *b* & *P* is equivalent to if *b* then *P* else *STOP* : *P* is enabled only if the boolean guard *b* is true.

The process *P 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 H Q* may act like either *P* or *Q*, with the choice being made

internally, and not under the control of the environment. The process *Hi ∈ I, R*(*i*) .

*P* (*i*) represents an indexed nondeterministic choice between the processes *P* (*i*) for *i ∈ I* such that *R*(*i*) holds. The process *P d 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 d Q* = (*P Q*) *H*

*Q* = (*P H STOP* ) *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*) = *STOP H Hx ∈ A* . *x → Chaos*(*A*)*.*

^

The process *P \ 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* = *P* [[*c.e/e | e ∈ αP* ]]*.*

T^he process *P*

 *Q* runs *P* and *Q* in parallel, synchronising on events from *A*.

*A*

By contract, the process *P ||| 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 be- haviours of processes, although most tools act on the (congruent) operational se- mantics.

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 |*`*A* represents the

restriction of *tr* to the events in *A*, whereas *tr \ A* represents *tr* with the events from *A* removed; concatenation is written “ˆ”; *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 |* (*tr, X*) *∈ failures*(*P* )*}.*

The traces *T* and failures *F* of a process also satisfy the following healthiness con- ditions:

**F1** *T* is non-empty and prefix-closed;

**F2** (*tr, X*) *∈ F ∧ Y ⊆ X ⇒* (*tr, Y* ) *∈ F* ;

**F3** (*tr, X*) *∈ F ∧* (*∀ a ∈ Y* . *tr*ˆ*⟨a⟩ ∈/ T* ) *⇒* (*tr, X ∪ Y* ) *∈ F* .

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

*Spec ±F P* iff *traces*(*Spec*) *⊇ traces*(*P* ) *∧ failures*(*Spec*) *⊇ 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 con- sidered in this paper— the traces inclusion is implied by the failures inclusion.

# Examples

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

* 1. *Determinism*

The best known such property is that of determinism [[8](#_bookmark23), Section 3.3]. *P* is deter- ministic if, whenever it can perform *a* after *tr*, it cannot refuse the *a*:

*∀ tr, a* . *tr*ˆ*⟨a⟩∈ traces*(*P* ) *⇒* (*tr, {a}*) *∈/ failures*(*P* )*.*

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

*∀*(*tr*1*, X*1)*,* (*tr*2*, X*2) *∈ failures*(*P* ) . *∀ a* . *tr*1 = *tr*2ˆ*⟨a⟩⇒ a ∈/ X*2*.*

* 1. *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* [[1](#_bookmark16)]. 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* [[2](#_bookmark17)], written *FNDC*(*P* ), if:

*∀ Hi ∈ CSPH* . *LH* (*P*

 *Hi*) *≡F P*

*H*

 *Stop,*

*H*

where *LH* (*Q*) = (*Q * *Chaos*(*H*)) *\ H* is Roscoe’s *lazy abstraction* [[8](#_bookmark23), Chapter 12],

^

*H*

and *CSPH* 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. Nondetermin- ism arises in models of systems for two main reasons. Often analysis is carried out

upon *designs* of systems, rather than concrete implementations; in designs, nonde- terminism 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 nonde- terminism 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 [[5](#_bookmark20)], I therefore considered the refinement-closure of FNDC. Process *P* satisfies *reﬁnement-closed, failures non-deducibility on compositions*, written *RCFNDC*(*P* ), if *∀ Q ± P* . *FNDC*(*Q*)*.* It turns out that this property can be captured as an binary

failures predicate, as follows (see [[5](#_bookmark20)]):

*∀*(*tr*1*, X*1)*,* (*tr*2*, X*2) *∈ failures*(*P* ) .

*tr*1 *|*`*H /*= *⟨⟩⇒ ∀ l* . *tr*2 = *tr*1 *|*`*L*ˆ*⟨l⟩⇒ l ∈/ X*1

*∧ ∀ l* . *tr*2ˆ*⟨l⟩* = *tr*1 *|*`*L ⇒ l ∈/ X*2*.*

* 1. *Causation*

In [[6](#_bookmark21)], we consider the question of whether an agent in a multi-user system, with

alphabet *A*, can cause some event *e ∈/ A* to occur. The obvious definition for

non-causation, called *traces non-causation*, is:

*/ ∃ tr* . *tr*ˆ*⟨e⟩∈ traces*(*P* ) *∧* (*tr \ A*)ˆ*⟨e⟩ ∈/ 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:

*∀*(*tr*1*, X*1)*,* (*tr*2*, X*2) *∈ failures*(*P* ) .

*∀ s, t* . *tr*1 = *s*ˆ*t*ˆ*⟨e⟩∧ s |*`*A /*= *⟨⟩ ∧ tr*2 = *s \ A ⇒ ﬁrst* (*t \ A*ˆ*⟨e⟩*) *∈/ X*2*.*

* 1. *Responsiveness*

In [[10](#_bookmark25)], 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

*∀ tr* . (*tr, J* ) *∈ failures*(*P*

*Q*) *⇒* (*tr, J* ) *∈ failures*(*P* )*.*

*J*

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

*∀*(*tr, X*) *∈ failures*(*P* ) .

*initsInJ* (*P/tr*) *− X /*= *{} ⇒* (*tr, initsInJ* (*P/tr*) *− X*) *∈/ failures*(*Q*)*,*

where *initsInJ* (*P/tr*) = *{x | tr*ˆ*⟨x⟩∈ traces*(*P* ) *∧ x ∈ J}.*

^

This property is equivalent to

*∀*(*tr*1*, X*1)*,* (*tr*2*, X*2) *∈ failures*(*P* ) .

*∀*(*tr*3*, X*3) *∈ failures*(*Q*) . *∀ x ∈ J* .

*tr*1 = *tr*2ˆ*⟨x⟩∧ tr*2 = *tr*3 *⇒* (*∃ y ∈ J* . *y ∈/ X*2 *∪ X*3) *∨ x ∈/ X*3*,*

which is a binary failures predicate on *P* .

# *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

*∀ P* . *PropR*(*P* ) *⇔ Spec ± Harness*(*P* )*.*

We begin by showing that, without loss of generality, we can restrict ourselves to predicates *R* that satisfy Definitions [3.1](#_bookmark2) and [3.2](#_bookmark3), below.

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

*R*((*tr*1*, X*1)*,...,* (*trn, Xn*)) *∧* (*∀ i ∈* 1 *.. n* . *Yi ⊆ Xi*) *⇒*

*R*((*tr*1*, Y*1)*,...,* (*trn, Yn*))*.*

We write *R*(*tr*1*,..., trn*) as a shorthand for *R*((*tr*1*, {}*)*,...,* (*trn, {}*)); this cap- tures that *R* allows the traces *tr*1*,..., trn*. Note that if *R* is subrefusal-closed, this is equivalent, to *∃ X*1*,..., Xn* . *R*((*tr*1*, X*1)*,...,* (*trn, Xn*))*.*

Let *impossiblei*(*tr*1*,..., trn*) be those events that cannot be appended onto the end of *tri* and allow *R* to be true:

*impossible* (*tr ,..., tr* ) = *{a | ¬R*(*tr ,..., tr* ˆ*⟨a⟩,..., tr*

)*}.*

*i* 1 *n* ^

1 *i n*

We omit the arguments of *impossiblei* when clear from the context.

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

*R*((*tr*1*, X*1)*,...,* (*trn, Xn*)) *⇒*

*R*((*tr*1*, X*1 *∪ impossible*1)*,...,* (*trn, Xn ∪ impossiblen*))*.*

*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](#_bookmark0) 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](#_bookmark4)

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

*R∗* = *{R' | R' ⊆ R ∧ R'* 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* , *PropR*(*P* ) *⇔ PropR∗* (*P* ).

**Proof:** The right-to-left implication is immediate, since *R∗ ⊆ R*. Suppose *PropR*(*P* ). Consider the set

*R'* = (*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' ⊆ R*, since *PropR*(*P* ). Hence *R' ⊆ R∗* and so *PropR∗* (*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* ) = *c.*1*.P ||| ... ||| c.n.P.*

^

For any failure (*tr, X*) of *Harness*(*P* ), we write (*tr, X*) *↓ c.i* for the contribution of the *i*th copy of *P* , i.e. [2](#_bookmark5)

(*tr, X*) *↓ c.i* = (*tr ↓ c.i, {x | c.i.x ∈ X}*)*.*

^

Then

*failures*(*Harness*(*P* )) =

*{*(*tr, X*) *| tr ∈ {|c|}∗ ∧ ∀ i ∈* 1 *.. n* . (*tr, X*) *↓ c.i ∈ failures*(*P* )*}.*

We now construct an appropriate specification. We need to consider two cases. The uninteresting case is when *¬R*(*⟨⟩,..., ⟨⟩*), and so *PropR* is unsatisfiable. We can take *Spec* = *d → STOP* , for example, and then

*PropR*(*P* ) *⇔ false ⇔ Spec ± Harness*(*P* )*,*

as required.

The more interesting case is where *R*(*⟨⟩,..., ⟨⟩*) 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*,..., trn*), we will allow the specification to refuse subsets of the following *characteristic refusals*:

*charRefs*

*R*(*tr*1*,..., trn*) =

*{*(*X*1*,..., Xn*) *| R*((*tr*1*, X*1)*,...,* (*trn, Xn*)) *∧*

^

*∀ i ∈* 1 *.. n* . *Xi ⊇ impossiblei*(*tr*1*,..., trn*)*}.*

2 If *tr* is a trace then *tr ↓ c.i* represents the sequence of data transmitted over channel *c.i* during *tr*.

We define the specification as follows:

*Spec* = *Spec*(*⟨⟩,..., ⟨⟩*)*,*

*Spec*(*tr*1*,..., trn*) = *H*(*X*1*,..., Xn*) *∈ charRefsR*(*tr*1*,..., trn*) .

^

*c*?*i*?*y* : Σ *− Xi → Spec*(*tr*1*,..., tri*ˆ*⟨y⟩,..., trn*)*.*

Note that after trace *tr*, *Spec* will be in state *Spec*(*tr ↓ c.*1*,..., tr ↓ c.n*).

Note also that if *R*(*tr*1*,..., trn*) holds, then *R*((*tr*1*, impossible*1)*,...,* (*trn, impossiblen*)), 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*,..., impossiblen*).

We show that *R*(*tr*1*,..., trn*) is indeed satisfied in every reachable state, by induction on the sum of the lengths of the traces. The base case holds by as- sumption. For the inductive step, suppose *R*(*tr*1*,..., trn*), and pick (*X*1*,..., Xn*) *∈*

*charRefsR*(*tr*1*,..., trn*) and *y ∈* Σ *− Xi*. Then by definition of *charRefs*, *y ∈/ impossiblei*, so *R*(*tr*1*,..., tri*ˆ*⟨y⟩,..., trn*).

Conversely, if *R*(*tr*1*,..., trn*) and *R*(*tr*1*,..., tri*ˆ*⟨y⟩,..., trn*) hold, then

*Spec*(*tr*1*,..., trn*) allows *c.i.y* since *y ∈/ impossiblei*. Hence *Spec* allows trace *tr*

precisely when *∀ tr' ≤ tr* . *R*(*tr' ↓ c.*1*,..., tr' ↓ c.n*)*.*

It is then easy to see that *Spec*(*tr*1*,..., trn*) allows the components to refuse precisely those (*X*1*,..., Xn*) such that *R*((*tr*1*, X*1)*,...,* (*trn, Xn*)): each such tuple of refusals is included within (*X*1 *∪ impossible*1*,..., Xn ∪ impossiblen*), 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

*failures*(*Spec*) = *{*(*tr, X*) *| tr ∈ {|c|}∗ ∧ R*((*tr, X*) *↓ c.*1*,...,* (*tr, X*) *↓ c.n*)

*∧ ∀ tr' ≤ tr* . *R*(*tr' ↓ c.*1*,..., tr' ↓ c.n*)*}.*

Hence, comparing the failures of *Spec* and *Harness*(*P* ),

*PropR*(*P* ) *⇔ Spec ±F Harness*(*P* )*,* as required.

# *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 *PropR*(*P* ). Consider an *n*-copy refinement test of the form *Spec ± 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

*{*(*tr, X*) *| ∃*(*tr*1*, X*1)*,...,* (*trn, Xn*) *∈ failures*(*P* ) .

*S*((*tr, X*)*,* (*tr*1*, X*1)*,...,* (*trn, Xn*))*},*

where *S* captures the semantic equations corresponding to how the harness is con- structed.

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

*∀*(*tr*1*, X*1)*,...,* (*trn, Xn*) *∈ failures*(*P* ) .

*∀*(*tr, X*) . *S*((*tr, X*)*,* (*tr*1*, X*1)*,...,* (*trn, Xn*)) *⇒* (*tr, X*) *∈ failures*(*Spec*)*.*

This is of the form *PropR*(*P* ) if we define *R*((*tr*1*, X*1)*,...,* (*trn, Xn*)) to be the predicate expressed by the second line of the above formula.

# Making it feasible

The construction in Section [3](#_bookmark1) 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* ) = *Sched*

^

*Spec'* = *Sched*

^



*{|c|}*



*{|c|}*

*Harness*(*P* )*, Spec.*

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

*Spec' ±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](#_bookmark8), sufficient conditions on *Sched* for the above refine- ment to hold precisely when *PropR*(*P* ) is true.

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

such that:

1. For all traces *tr*1*,..., trn* corresponding to failures where *R* does not hold, *S*

includes a corresponding trace, i.e.:

*∀ tr*1*,..., trn, X*1*,..., Xn* . *¬R*((*tr*1*, X*1)*,...,* (*trn, Xn*)) *⇒*

*∃ tr ∈ S* . *∀ i ∈ {*1 *.. n}* . *tr ↓ c.i* = *tri.*

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

1. For all failures (*tr ↓ c.*1*, X*1)*,...,* (*tr ↓ c.n, Xn*) where *R* does not hold, *Sched*

allows the harness to demonstrate the falsity of *R*; it does this by allowing

*n*

all events in some (possibly different) sets *X'*

1

*⊆ X*1*,...,X'*

*⊆ Xn* such that

*¬R*((*tr ↓ c.*1*,X'* )*,...,* (*tr ↓ c.n, X'* )).

1 *n*

*tr ∈ S ∧ ¬R*((*tr ↓ c.*1*, X*1)*,...,* (*tr ↓ c.n, Xn*)) *⇒*

*∃ X' ⊆ X*1*,...,X' ⊆ Xn* . *¬R*((*tr ↓ c.*1*,X'* )*,...,* (*tr ↓ c.n, X'* )) *∧*

1 *n* 1 *n*

*∀ i ∈* 1 *.. n* . *∀ a ∈ X'* . *tr*ˆ*⟨c.i.a⟩∈ S.*

*i*

Intuitively, it is the events in the *X'* that demonstrate the falsity of *R*: for

*i*

example, in the case of determinism, *R*((*tr*1*, X*1)*,* (*tr*2*, X*2) is false whenever

1

*tr*1 = *tr*2ˆ*⟨a⟩* and *a ∈ X*2 for some *a*; so we can take *X'*

2

= *{a}* and *X'*

= *{}*.

If *Harness'*(*P* ) refuses the events of the *X'* then that refusal can be attributed

*i*

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](#_bookmark11), we show that this is straightforward for the examples of Section [2](#_bookmark0). As *Sched* is deterministic with traces *S*,

*failures*(*Sched* (*S*)) = *{*(*tr, X*) *| tr ∈ S ∧ ∀ x ∈ X* . *tr*ˆ*⟨x⟩ ∈/ S}.*

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

**Theorem 5.2** *PropR*(*P* ) *⇔ Spec' ±F Harness'*(*P* ).

**Proof:** If *¬PropR*(*P* ) then there exist *f*1 = (*tr*1*, X*1)*,..., fn* = (*trn, Xn*) *∈*

*failures*(*P* ) such that *¬R*(*f*1*,..., fn*). By condition ([i](#_bookmark9)) of Definition [5.1](#_bookmark8), there is

some *tr ∈ S* such that *∀ i ∈* 1 *.. n* . *tr ↓ c.i* = *tri*. Let *X' ,...,X'* be as in condi-

1 *n*

tion ([ii](#_bookmark10)) of Definition [5.1](#_bookmark8), and let *f'* = (*tr ,X'*), for *i ∈* 1 *.. n*; so *¬R*(*f',...,f'* ).

*f' ∈*

*P*

*i ∈*

*. .n*, by Axiom F2. Let *X*

*{c.i.x | i ∈*

*. .n, x ∈ X'}*

^

*i*

Then

*i*

*f* = (*tr, X*

and

*failures*(

), for 1

*f ∈*

*i* ^ *i i* = 1 *i n*

Now, (*tr |*`*{|c|},X*) *∈/ failures*(*Spec*), by construction. Also, for each *i ∈* 1*. .n*, for

^

). Then

*failures*(*Harness'*(

*P*

))

*.*

every *a ∈ X'*, we have *tr*ˆ*⟨c.i.a⟩ ∈ S*, by definition of the *X'*. Hence (*tr, {c.i.a}*) *∈/*

*i* *i*

*failures*(*Sched* ), and so *f ∈/ failures*(*Spec'*). Hence *Spec' /±F Harness'*(*P* ).

Conversely, if *Spec' /±F Harness'*(*P* ) then there is some *f* = (*tr, X*) *∈ failures*(*Harness'*(*P* )) such that *f ∈/ failures*(*Spec'*). Then *tr ∈ traces*(*Sched* ) and so *tr ∈ S*. We show that *¬R*(*f ↓ c.*1*,...,f ↓ c.n*). Suppose, otherwise. Then (*tr |*`

*{|c|},X*) *∈ failures*(*Spec*), by construction. Now, *tr ∈ S* so (*tr, {}*) *∈ failures*(*Sched* ) and so *f ∈ failures*(*Spec'*), giving a contradiction.

Hence *¬R*(*f ↓ c.*1*,...,f ↓ c.n*). Let *X' ,...X'* be as in condition ([ii](#_bookmark10)) of Def-

1 *n*

inition [5.1](#_bookmark8). Let *f* = (*tr ↓ c.i, X'*), for *i* = 1 *.. n*. Then *¬R*(*f ,...,f* )*.* Now,

= *i* ^

*i* 1 *n*

, so ( ) *'*

*i*

1 *. . n, x ∈ X'} ⊆ X tr, X' ∈ failures*(*Harness* (*P* )). *Sched*

*X* ^ *{c.i.x | i ∈*

is not refusing any of the events of *X'*, by definition of the *X'*; hence those events

*i*

*i*

*'*

must be refused by *c.i.P* , so *fi ∈ failures*(*P* ), for *i* = 1 *.. n*. Hence *¬PropR*(*P* ).

# 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](#_bookmark0).

* 1. *Determinism*

Recall that for determinism we have

*R*((*tr ,X* )*,* (*tr ,X* )) = *∀ a* . *tr*

= *tr* ˆ*⟨a⟩⇒ a ∈/ X .*

1 1 2 2 ^ 1 2 2

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

*Sched* = *c.*1?*x → c.*2*.x → Sched,*

^

Note that *Sched* is deterministic, and satisfies the conditions of Definition [5.1](#_bookmark8): for condition ([i](#_bookmark9)), note that *Sched* includes a trace corresponding to all cases where *R* is false (i.e., where *tr ↓ c.*1 = *tr ↓ c.*2ˆ*⟨a⟩*, for some *a*); for condition ([ii](#_bookmark10)), if

*tr ↓ c.*1 = *tr*2 *↓ c.*2ˆ*⟨a⟩*, we can take *X'*

1

2

= *{}* and *X'*

= *{a}*, and note that *Sched*

allows *c.*2*.a*.

Note that

*charRefs*(*tr, tr*)= (P Σ)2*,*

*charRefs*(*tr*ˆ*⟨a⟩, tr*)= *{*(*X*1*, X*2) *| a ∈/ X*2*}.*

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

*Spec*(*tr, tr*)  *Sched*

= step laws: *Spec* chooses over all pairs of refusals

*HX ∈* P Σ . *c.*1?*y* : Σ *− X →* (*Spec*(*tr*ˆ*⟨y⟩, tr*)  *c.*2*.y → Sched* )

= by consideration of initial refusals;

step laws: *Spec* does not allow *c.*2*.y* to be refused

*c.*1?*y → c.*2*.y →* (*Spec*(*tr*ˆ*⟨y⟩, tr*ˆ*⟨y⟩*)  *Sched* ) *H STOP.*

Hence for all *tr*, *Spec*(*tr, tr*)  *Sched* is equivalent to

*Spec'* = *c.*1?*y → c.*2*.y → Spec' H STOP.*

(Formally, this is by the Unique Fixed Point rule [[8](#_bookmark23), Section 1.3].) To summarise,

*P* is deterministic if

*Spec' ±F Sched * (*c.*1*.P ||| c.*2*.P* )*.*

This is Lazi´c’s test from [[4](#_bookmark19)].

* 1. *RCFNDC*

Recall that for *RCFNDC* we have

*R*((*tr ,X* )*,* (*tr ,X* )) = *tr*

*|*`*H /*= *⟨⟩⇒ ∀ l* . *tr*

= *tr |*`*L*ˆ*⟨l⟩⇒ l ∈/ X*

1 1 2 2 ^ 1

2 1 1

*∧ ∀ l* . *tr*2ˆ*⟨l⟩* = *tr*1 *|*`*L ⇒ l ∈/ X*2*.*

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:

*Sched* = *c.*1?*h* : *H → Sched*

^

*c.*1?*l* : *L → Sched* 1(*l*)

*c.*2?*l* : *L → Sched* 2(*l*)*,*

*Sched* 1(*l*) = *c.*2*.l → Sched,*

^

*Sched* 2(*l*) = *c.*1*.l → Sched.*

^

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 *|*`*L*ˆ*⟨l⟩* or *tr*2ˆ*⟨l⟩* = *tr*1 *|*`*L* for some *l*; it is easy to construct a corresponding trace of *Sched* where the equal events of *tr*1 *|*`*L* and *tr*2 are performed consecutively, in either order. Also, condition ([ii](#_bookmark10)) of Definition [5.1](#_bookmark8) 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 * *Spec*(*⟨⟩, ⟨⟩*). Throughout the following, suppose *tr*1 *|*`*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.

*Sched * *Spec*(*tr*1*, tr*2) =

⎛*c.*1?*h* : *H →* (*Sched * *Spec*(*tr*1ˆ*⟨h⟩, tr*2)) ⎞

⎜⎝ ⎟⎠

If *tr*1 *|*`*H* = *⟨⟩*:

*c.*1?*l* : *L →* (*Sched* 1(*l*)  *Spec*(*tr*1ˆ*⟨l⟩, tr*2)) *H STOP.*

*c.*2?*l* : *L →* (*Sched* 2(*l*)  *Spec*(*tr*1*, tr*2ˆ*⟨l⟩*))

*Sched* 1(*l*)  *Spec*(*tr*1ˆ*⟨l⟩, tr*2) =

*c.*2*.l →* (*Sched * *Spec*(*tr*1ˆ*⟨l⟩, tr*2ˆ*⟨l⟩*)) *H STOP,*

*Sched* 2(*l*)  *Spec*(*tr*1*, tr*2ˆ*⟨l⟩*) =

*c.*1*.l →* (*Sched * *Spec*(*tr*1ˆ*⟨l⟩, tr*2ˆ*⟨l⟩*)) *H STOP.*

If *tr*1 *|*`*H /*= *⟨⟩* then *Spec* can’t refuse respectively *c.*2*.l* and *c.*1*.l* in the following cases:

|  |  |  |
| --- | --- | --- |
| *Sched* 1(*l*) | *Spec*(*tr*1ˆ*⟨l⟩, tr*2) = *c.*2*.l →* (*Sched* | *Spec*(*tr*1ˆ*⟨l⟩, tr*2ˆ*⟨l⟩*))*,* |
| *Sched* 2(*l*) | *Spec*(*tr*1*, tr*2ˆ*⟨l⟩*) = *c.*1*.l →* (*Sched* | *Spec*(*tr*1ˆ*⟨l⟩, tr*2ˆ*⟨l⟩*))*.* |

So we can define *Spec'* as follows; the two clauses correspond to *Sched * *Spec*(*tr*1*, tr*2) with *tr*1 *|*` *H /*= *⟨⟩* and *tr*1 *|*` *H /*= *⟨⟩*, respectively; the other cases have been absorbed into these clauses.

⎛*c.*1?*h* : *H → Spec''* ⎞

^ ⎜⎝ ⎟⎠

*Spec'* =  *c.*1?*l* : *L →* (*c.*2*.l → Spec' H STOP* ) *H STOP,*

*c.*2?*l* : *L →* (*c.*1*.l → Spec' H STOP* )

⎛*c.*1?*h* : *H → Spec''* ⎞

^ ⎜⎝ ⎟⎠

*Spec''* =  *c.*1?*l* : *L → c.*2*.l → Spec'' H STOP.*

*c.*2?*l* : *L → c.*1*.l → Spec''*)

This is equivalent to, but defined rather differently from, the test in [[5](#_bookmark20)] [3](#_bookmark13) .

* 1. *Causation*

Recall that for causation we have

*R*((*tr ,X* )*,* (*tr ,X* )) =

1 1 2 2 ^

*∀ s, t* . *tr*1 = *s*ˆ*t*ˆ*⟨e⟩∧ s |*`*A /*= *⟨⟩ ∧ tr*2 = *s \ A ⇒ ﬁrst* (*t \ A*ˆ*⟨e⟩*) *∈/ X*2*.*

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* = *αP − 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 *ﬁrst* (*t\A*ˆ*⟨e⟩*). We then (state *Sched''*(*b*)) allow arbitrary events on *c.*1, corresponding to the rest of *t*, and also allow *c.*2*.b*.

*Sched* = *c.*1?*a* : *A → Sched' c.*1?*b* : *B → c.*2*.b → Sched,*

*Sched'* = *c.*1?*a* : *A → Sched'*

^

*c.*1?*b* : *B →* (*c.*2*.b → Sched' ping → Sched''*(*b*))

*Sched''*(*b*) = *c.*1?*d* : *αP → Sched''*(*b*) *c.*2*.b → STOP.*

^

Note that *R*((*tr*1*, X*1)*,* (*tr*2*, X*2)) is false only when, for some *s* and *t*, *tr*1 = *s*ˆ*t*ˆ*⟨e⟩ ∧ s |*`*A /*= *⟨⟩ ∧ tr*2 = *s \ A*. Such a trace is reflected by the traces of *Sched* that end in state *Sched''*(*b*). Note also that condition ([ii](#_bookmark10)) of Definition [5.1](#_bookmark8) is satisfied.

The only non-trivial case to check is after a trace as above; taking *X'* = *{b}*, we

2

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 * *Spec* is equivalent to the following pro- cess.

*Spec*

= (*c.*1?*a* : *A → Spec c.*1?*b* : *B →* (*c.*2*.b → Spec H STOP* )) *H STOP,*

1 ^ ⎛ 2 ⎞

1

*Spec*

⎟

else 2 2 3 )⎠

2

⎜⎝

then *c.*2*.b → Spec d ping → Spec* (*b, b*)

*c. .e → Spec*2 *ping → Spec*3(*e, e*

*c.*1?*a* : *A → Spec*2

*c.*1?*b* : *B →* if *b /*= *e*

=^

⎜

*H STOP,*

*Spec* (*b, c*) = if *c /*= *e*

3 ^ then ( 1? :

*c.*

*d*

*.b →*

*H*

( ) 2

*αP → Spec*3 *b, d*

*c.*

*STOP* )

*STOP*

else *c.*1?*d* : *αP → Spec*3(*b, d*) *d c.*2*.b → STOP.*

This is similar to the test from [[6](#_bookmark21)].

3 The test in [[5](#_bookmark20)] 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.

* 1. *Responsiveness*

Recall that for responsiveness we have

*R*((*tr ,X* )*,* (*tr ,X* )) =

1 1 2 2 ^

*∀*(*tr*3*, X*3) *∈ failures*(*Q*) . *∀ x ∈ J* .

*tr*1 = *tr*2ˆ*⟨a⟩∧ tr*2 = *tr*3 *⇒* (*∃ y ∈ J* . *y ∈/ X*2 *∪ X*3) *∨ x ∈/ X*3*.*

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

*∀ f*1 *∈ failures*(*P*1)*,..., fn ∈ failures*(*Pn*) . *R*(*f*1*,..., fn*)*,*

where some of the *Pi* may be the same, by considering a harness of the form [4](#_bookmark15)

(*c.*1*.P*1 *||| ... ||| c.n.Pn*)  *Sched.*

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

*R'*((*tr ,X* )*,* (*tr ,X* )*,* (*tr ,X* )) =

1 1 2 2 3 3 ^

*∀ x ∈ J* . *tr*1 = *tr*2ˆ*⟨a⟩∧ tr*2 = *tr*3 *⇒* (*∃ y ∈ J* . *y ∈/ X*2 *∪ X*3) *∨ x ∈/ X*3*.*

We can then consider a harness of the form:

*Harness*(*P, Q*) = (*c.*1*.P ||| c.*2*.P ||| c.*3*.Q*)  *Sched.*

Note that *R'* is false only when *tr*1 = *tr*2ˆ*⟨a⟩* 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](#_bookmark10)) of Definition [5.1](#_bookmark8), we allow *all* events on *c.*2 and *c.*3, but allow the processes to proceed only if the events match:

*Sched* = *c.*1?*x → Sched'*(*x*)*,*

*Sched'*(*x*) = *c.*2?*y → x* = *y* & *c.*3*.x → Sched c.*3?*y → x* = *y* & *c.*2*.x → Sched.* 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 * *Spec* is equivalent to *Spec*0, below:

*Spec*

= *c.*1?*x → Spec* (*x*) *H STOP,*

0 ^ 1

*Spec* (*x*) = (*c.*2?*y → Spec* (*x, y*) *c.*3?*y → Spec* (*x, y*))

1 ^ 2 3

*d*

*Hy ∈ J* . (*c.*2*.y → Spec*2(*x, y*) *c.*3*.y → Spec*3(*x, y*))

*H c.*3*.x →* (*c.*2*.x → Spec*0 *H STOP* )

,

*Spec* (*x, y*) = *x* = *y* & *c.*3*.x → Spec H STOP,*

2 ^ 0

*Spec* (*x, y*) = *x* = *y* & *c.*2*.x → Spec*

*H STOP.*

3 ^ 0

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

*AnyP* = *choose*?*i* :1 *.. n → Pi,*

*Sched'* = *c.*1*.choose.*1*.P*1 *→ ... → c.n.choose.n.Pn → Sched, Harness'* = (*c.*1*.AnyP ||| ... ||| c.n.AnyP* )  *Sched'.*

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.

The refinement test from [[10](#_bookmark25)] is somewhat different from this: there *Q* is syn- chronised directly with the second copy of *P* so as to combine their refusals; further, the clause “*x ∈/ 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).

# Discussion

In this paper we have shown that refinement checks of the form *Spec ± Harness*(*P* ), where *Harness*(*P* ) runs *n* copies of *P* , capture precisely those predicates of the form

*∀ f*1*,..., fn ∈ failures*(*P* ) . *R*(*f*1*,..., fn*)*.* 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”:

*∀*(*tr*1*, X*1)*,* (*tr*2*, X*2)*,* (*tr*3*, X*3) *∈ failures*(*P* ) . *∀ tr, a, b,c* .

*tr*1 = *tr*ˆ*⟨a⟩∧ tr*2 = *tr*ˆ*⟨b⟩∧ tr*3 = *tr*ˆ*⟨c⟩⇒ a* = *b ∨ a* = *c ∨ b* = *c.*

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 circum- stances: 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*; responsive- ness 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

*∀ f ∈ failures*(*P* ) . *∃ f' ∈ failures*(*P* ) . *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.

# Acknowledgement

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

# References

1. Riccardo Focardi and Roberto Gorrieri. A classification of security properties. *Journal of Computer Security*, 1995.
2. Riccardo Focardi. Comparing two information flow security properties. In *Proceedings of 9th IEEE Computer Security Foundations Workshop*, pages 116–122, 1996.
3. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
4. Ranko Lazi´c. *A Semantic Study of Data Independence with Applications to Model Checking*. D.Phil., Oxford University, 1999.
5. Gavin Lowe. On information flow and refinement-closure. In *Proceedings of the Workshop on Issues in the Theory of Security (WITS ’07)*, 2007.
6. 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.
7. A. W. Roscoe. Model-checking CSP. In *A Classical Mind, Essays in Honour of C. A. R. Hoare*. Prentice-Hall, 1994.
8. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1997.
9. A. W. Roscoe. On the expressive power of CSP refinement. *Formal Aspects of Computing*, 17(2):93– 112, 2005.
10. J. N. Reed, J. E. Sinclair, and A. W. Roscoe. Responsiveness of interoperating components. *Formal* *Aspects of Computing*, 16:394–411, 2004.

# Details of derivations

In this appendix we give details of the derivations of a couple of the specification processes from Section [6](#_bookmark11).

* 1. *Causation*

We consider all the reachable states of *Sched * *Spec* from Section [6.3](#_bookmark12). Firstly, suppose *tr*2 = *tr*1 *\ A*, *tr*1 *|*` *A* = *⟨⟩* and *b ∈ B*. Then in both the following cases, *Spec* allows all refusals.

*Sched * *Spec*(*tr*1*, tr*2) =

*c.*1?*a* : *A →* (*Sched' * *Spec*(*tr*1ˆ*⟨a⟩, tr*2))

*H STOP,*

*c.*1?*b* : *B →* (*c.*2*.b → Sched Spec*(*tr*1ˆ*⟨b⟩, tr*2))

*c.*2*.b → Sched * *Spec*(*tr*1ˆ*⟨b⟩, tr*2) =

*c.*2*.b →* (*Sched * *Spec*(*tr*1ˆ*⟨b⟩, tr*2ˆ*⟨b⟩*)) *H STOP.*

Now suppose *tr*2 = *tr*1 *\ A*, *tr*1 *|*` *A /*= *⟨⟩*, *b ∈ B* and *b /*= *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' Spec*(*tr*1*, tr*2) =

*c.*1?*a* : *A →* (*Sched' * *Spec*(*tr*1ˆ*⟨a⟩, tr*2))

*c.*1?*b* : *B →* (*c.*2*.b → Sched' ping → Sched''*(*b*)  *Spec*(*tr*1ˆ*⟨b⟩, tr*2))

*H STOP,*

*c.*2*.b → Sched' ping → Sched''*(*b*)  *Spec*(*tr*1ˆ*⟨b⟩, tr*2) =

*c.*2*.b →* (*Sched' * *Spec*(*tr*1ˆ*⟨b⟩, tr*2ˆ*⟨b⟩*))

*d ping →* (*Sched''*(*b*)  *Spec*(*tr*1ˆ*⟨b⟩, tr*2))*,*

*c.*2*.e → Sched' ping → Sched''*(*b*)  *Spec*(*tr*1ˆ*⟨e⟩, tr*2) =

*c.*2*.e →* (*Sched' * *Spec*(*tr*1ˆ*⟨e⟩, tr*2ˆ*⟨e⟩*))

*ping →* (*Sched''*(*e*)  *Spec*(*tr*1ˆ*⟨e⟩, tr*2))*.*

Finally, suppose *tr*2 *< tr*1 *\ A*, *b* is the first event in *tr*1 *\ A* after *tr*2, *tr*1 *|*` *A /*= *⟨⟩*; 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*)  *Spec*(*tr*1*, tr*2) = if *last tr*1 */*= *e*

then *c.*1?*d* : *αP →* (*Sched''*(*b*)  *Spec*(*tr*1ˆ*⟨d⟩, tr*2))

*c.*2*.b → STOP*

*H STOP*

else *c.*1?*d* : *αP →* (*Sched''*(*b*)  *Spec*(*tr*1ˆ*⟨d⟩, tr*2)) *d c.*2*.b → STOP.*

We can therefore use the specification *Spec*1 from Section [6.3](#_bookmark12) (the processes *Spec*1, *Spec*2 and *Spec*3(*b, c*) correspond, respectively, to the first, third, and sixth processes above, with *c* = last *tr*1 in the final case).

* 1. *Responsiveness*

We consider all reachable states of *Sched * *Spec*(*⟨⟩, ⟨⟩, ⟨⟩*) from Section [6.4](#_bookmark14). In the following state, *Spec* allows all refusals.

*Sched * *Spec*(*tr, tr, tr*) = *c.*1?*x →* (*Sched'*(*x*)  *Spec*(*tr*ˆ*⟨x⟩, tr, tr*)) *H STOP.*

In the following state, *Spec* cannot allow refusals *X*2 and *X*3 such that *x ∈ X*3 and

*X*2 *∪ X*3 *⊇ J* .

*Sched'*(*x*)  *Spec*(*tr*ˆ*⟨x⟩, tr, tr*)

= step laws

*HX*2*, X*3 *∈* P *J, x ∈/ X*3 *∨ X*2 *∪ X*3 *⊂ J* .

*c.*2?*y* : *J − X*2 *→* (*x* = *y* & *c.*3*.x → Sched * *Spec*(*tr*ˆ*⟨x⟩, tr*ˆ*⟨y⟩, tr*))

 *c.*3?*y* : *J − X*3 *→* (*x* = *y* & *c.*2*.x → Sched Spec*(*tr*ˆ*⟨x⟩, tr, tr*ˆ*⟨y⟩*))

= by consideration of initial refusals

*c.*2?*y →* (*x* = *y* & *c.*3*.x → Sched * *Spec*(*tr*ˆ*⟨x⟩, tr*ˆ*⟨y⟩, tr*))

*c.*3?*y →* (*x* = *y* & *c.*2*.x → Sched * *Spec*(*tr*ˆ*⟨x⟩, tr, tr*ˆ*⟨y⟩*))

*d*

⎛⎜*H y ∈ J* . *c.*2*.y →* (*x* = *y* & *c.*3*.x → Sched * *Spec*(*tr*ˆ*⟨x⟩, tr*ˆ*⟨y⟩, tr*)) ⎞⎟

 *c.*3*.y →* (*x* = *y* & *c.*2*.x → Sched Spec*(*tr*ˆ*⟨x⟩, tr, tr*ˆ*⟨y⟩*))

⎝

⎠

*H c.*3*.x →* (*c.*2*.x → Sched Spec*(*tr*ˆ*⟨x⟩, tr, tr*ˆ*⟨x⟩*))

In the following case, *Spec* allows all refusals

*x* = *y* & *c.*3*.x → Sched * *Spec*(*tr*ˆ*⟨x⟩, tr*ˆ*⟨y⟩, tr*) =

*x* = *y* & *c.*3*.x →* (*Sched Spec*(*tr*ˆ*⟨x⟩, tr*ˆ*⟨x⟩, tr*ˆ*⟨x⟩*)) *H STOP,*

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](#_bookmark14).