Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 250 (2009) 49–68
www.elsevier.com/locate/entcs

On Refinement-Closed Security Properties and Nondeterministic Compositions

Toby Murray [1] Gavin Lowe [2]

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD, UK

Abstract

Refinement-closed security properties allow the verification of systems for all possible implementations. Some systems, however, have refinements that do not represent possible implementations. In particular, real instantiations of abstract systems comprising security-critical components surrounded by maximally hostile unrefined components are often characterised only by compositions of refinements of the abstract system's components, rather than all refinements of the abstract system. In this case, refinement-closed security properties that examine multiple behaviours of a system at once can be falsely violated by the presence of inconsistent pairs of behaviour arising from different, incompatible refinements of the system's components.
We show how to weaken a class of such properties, which includes both information flow and causation properties, to allow them to be applied to these sorts of abstract systems. The weakened properties ignore all pairs of inconsistent behaviour that would have violated the original property from which they are derived. We also show how to adapt existing automated tests for these properties to allow them to be used to test for their weakened counterparts instead. This enables greater flexibility in the application of these sorts of properties to compositions of nondeterministic components.

Keywords: Refinement-closed security properties, nondeterminism, information flow, causation, CSP.

1 Introduction

1.1 Refinement-Closed Security Properties

When model checking security-critical components, it is customary to construct an abstract system that contains the security-critical components surrounded by untrusted components that exhibit any and all behaviours permitted by the threat model. For example, when modelling cryptographic protocols, a common approach composes honest protocol agents with a maximally hostile intruder, in order to verify the security properties of the protocol being executed by the honest agents [14]. If the security-critical components, such as the honest protocol agents, function

[1] Email: toby.murray@comlab.ox.ac.uk
[2] Email: gavin.lowe@comlab.ox.ac.uk

doi:10.1016/j.entcs.2009.08.017

correctly in the presence of this maximally hostile untrusted environment, we gain confidence that they uphold their security properties independently of the components with which they are composed and, hence, should uphold these properties universally.

Formalisms, such as CSP [13], that support a notion of behavioural *refinement* greatly assist in this process. In this case, one component, Q, refines another, P, precisely when all behaviours of Q are also behaviours of P. To be useful, such a formalism should support the notion that, when Q refines P, a system built by placing Q in a particular context will always refine the system built by placing P in that same context, for all possible contexts.

In this case, we can model untrusted components as the most unrefined components that exhibit any and all feasible behaviours under the threat model. Any refinement of an untrusted component thus represents the behaviour of a real component with which the security-critical components may be composed in practice. An abstract system can them be formed by composing the security-critical components with the unrefined untrusted components. We can then apply *refinement-closed* correctness properties to the abstract system in order to verify its security-critical parts. A refinement-closed property is one that, if it holds for a system will also hold for all of the system's refinements. Applying refinement-closed properties to the abstract system is sound because if a property holds for all refinements of the abstract system, it is guaranteed to hold for all possible real instantiations of the system. Hence, it will hold for all possible components with which the security-critical components might be composed in practice.

1.2 Properties in CSP

CSP processes model individual components that when combined in parallel form larger CSP processes that model whole systems. We restrict our attention to processes that are *divergence-free* since a divergent process, which can reach states in which it *diverges* by performing an infinite amount of internal activity, are almost always incorrect by definition. CSP's *stable failures* denotational semantic model is useful for reasoning about divergence-free processes. It captures the behaviour of a process P by two sets, $\mathit{traces}(P)$ and $\mathit{failures}(P)$. $\mathit{traces}(P)$ is the set containing all finite sequences of visible actions that the process P can perform. Each member of the set $\mathit{failures}(P)$ is known as a *stable failure* and is a pair, (s, X), representing that P can perform the sequence of events s and then reach a "stable" state from which no internal activity is possible and none of the events from X can be performed. The traces and failures of a divergence-free process P are related by $\mathit{traces}(P) = \{s \mid (s, X) \in \mathit{failures}(P)\}$. In this model, a process Q *refines* another P, written $P \sqsubseteq Q$, if and only if $\mathit{traces}(Q) \subseteq \mathit{traces}(P) \wedge \mathit{failures}(Q) \subseteq \mathit{failures}(P)$. Further details about the stable failures model and CSP appear in Appendix A.

Properties of CSP processes are typically expressed in terms of the process's representation in a particular model. In this paper, all properties are expressed in terms of the traces and failures of a process – *i.e.* within the stable failures model. We use the following sequence notation when expressing properties. We write se-

quences of events in between angle-brackets, $\langle \ldots \rangle$. $s \mathbin{\hat{}} t$ denotes the concatenation of sequences s and t. $s \setminus A$ is the sequence obtained by removing all events in the set A from the sequence s, while $s \upharpoonright A$ denotes the sequence obtained by removing all non-A events from s.

The property of *determinism* is an example of a property that is refinement-closed. A process P is deterministic if and only if it is divergence-free and it can never perform and refuse the same event e after some trace s. In this case we write $det(P)$. This property can be expressed in the stable failures model, for divergence-free processes P, as

$$det(P) = \nexists\, s, e \bullet s \mathbin{\hat{}} \langle e \rangle \in traces(P) \wedge (s, \{e\}) \in failures(P).$$

We can see that for any process P, $det(P) \Rightarrow (\forall\, Q \bullet P \sqsubseteq Q \Rightarrow det(Q))$. Hence, det is refinement-closed. This makes sense since deterministic processes have no proper divergence-free refinements.

1.3 Refinement-Closed Independence Properties

In the context of CSP, a number of refinement-closed properties have recently been proposed for verifying certain security properties of systems, including the absence of information flows [7] and causation [10]. These properties differ from most of their predecessors because they examine multiple behaviours of a system at once. In particular, these kinds of security properties examine pairs of behaviours and are violated by the presence of certain related pairs. We term these properties *refinement-closed independence properties*, since they measure whether the occurrence of events from some set is *independent* of some other factor. The following definition captures this idea formally.

Definition 1.1 [Refinement-Closed Independence Property] A *refinement-closed independence property*, $\mathcal{I}$, is one that can be expressed as

$$\mathcal{I}(P) = \nexists\, s_1, e, t_1, s_2 \bullet s_1 \mathbin{\hat{}} \langle e \rangle \mathbin{\hat{}} t_1 \in traces(P) \wedge (s_2, \{e\}) \in failures(P) \wedge Pred(s_1, e, s_2, t_1),$$

for some predicate $Pred$ such that $Pred(s_1, e, s_2, t_1) \Rightarrow s_1 \upharpoonright \{e\} = s_2 \upharpoonright \{e\}$.

It is trivial to see that all such properties are refinement-closed. The relationship $Pred$ captures the other factor that the occurrences of events e are supposed to be independent of. The requirement that $Pred(s_1, e, s_2, t_1) \Rightarrow s_1 \upharpoonright \{e\} = s_2 \upharpoonright \{e\}$ simply states that whatever factor that the occurrence of a specific event e is supposed to be independent of, it is not the occurrence, or not, of the event e itself. This seems entirely sensible to assume since it makes little sense to talk about whether something is independent of itself.

The information flow property referred to above is known as *Refinement-Closed Failures Non-Deducibility on Compositions* (RCFNDC) [7]. RCFNDC asserts that no information can flow from a system's high interface, H, to its disjoint low interface, L, where these two interfaces partition the entire external interface exposed by the system. It does so by capturing whether all occurrences of low events $e \in L$

are independent of the occurrence of high events from H. Here, the predicate *Pred* can be expressed as

$$e \in L \wedge t_1 = \langle\rangle \wedge ((s_1 \upharpoonright H \neq \langle\rangle \wedge s_2 = s_1 \setminus H) \vee (s_2 \upharpoonright H \neq \langle\rangle \wedge s_1 = s_2 \setminus H)).$$

Lazy Independence [13] is another refinement-closed independence property for a lack of information flow. Here *Pred* is $e \in L \wedge t_1 = \langle\rangle \wedge s_1 \setminus H = s_2 \setminus H$.

The refinement-closed property for a lack of causation referred to above is known simply as *Non-Causation* [10]. Non-Causation asserts that the occurrence of events from the set A cannot cause any of the events from the disjoint set B to occur. It does so by capturing whether all occurrences of events from the set B and any non-A event that precedes a B event are independent of the occurrence of events from the set A. Here, the associated *Pred* may be written

$$e \notin A \wedge last(t_1) \in B \wedge s_1 \upharpoonright A \neq \langle\rangle \wedge s_2 = s_1 \setminus A.$$

Finally, it should be noted that the property of determinism is itself a refinement-closed independence property; although, it is somewhat degenerate. Determinism asserts that all events e occur independently of all unobservable factors. Here *Pred* can be expressed simply as $t = \langle\rangle \wedge s_1 = s_2$.

1.4 The Problem with Compositions of Nondeterministic Components

Unfortunately, trying to apply one of these properties to an abstract system that is formed as the composition of unrefined components can be problematic. This is because it is possible for the property to be violated by the presence of a pair of *inconsistent* behaviours that contains behaviours from multiple incompatible refinements of the system's components. For example, in the case of abstract systems that comprise security-critical components surrounded by unrefined, untrusted components, any real system that includes the security-critical components will be formed by composing them with refinements of the untrusted ones. Pairs of behaviour that cannot arise in any such implementation are considered to be inconsistent. In situations like these in which the *only* valid implementations can be formed as the composition of refinements of the abstract system's components, the presence of inconsistent pairs of behaviour can lead to the property being falsely violated. This differs from the usual interpretation of a CSP process in which *all* refinements are considered to represent valid implementations and to which it is most appropriate to apply ordinary refinement-closed independence properties.

As an example of nondeterminism leading to a refinement-closed independence property being falsely violated, consider a system that is known to be built by interleaving two components, *High* and *Low*, that run independently of each other. We trust that *High* will only perform the single event h before refusing to perform any further events. *High* could be modelled in CSP as

$$High = h \rightarrow STOP.$$

However we are less certain about *Low*'s behaviour. *Low* may perform the single event l before refusing to perform any further events, or *Low* may immediately refuse to perform any events at all, leading to the following model of *Low* in CSP

that is the nondeterministic choice between these two possibilities.

$$Low = l \rightarrow STOP \sqcap STOP$$

The abstract system built by interleaving *High* and *Low* may be defined as

$$Sys = High \;|||\; Low.$$

It contains a high interface, $H = \{h\}$, and a disjoint low interface, $L = \{l\}$.

The abstract system includes the following behaviours.

- $\langle h, l\rangle \in traces(Sys)$: When *Low* performs l, the system can perform the sequence of events $\langle h, l\rangle$.
- $(\langle\rangle, \{l\}) \in failures(Sys)$: On the other hand, were *Low* to simply refuse all events, then the system would initially refuse to perform the event l.

Notice that these behaviours arise from *different* proper refinements of *Low*, corresponding to implementations of *Low* in which it does, and does not, perform l initially, respectively. But taken together, they violate RCFNDC.

What are we to make of the fact that this system apparently fails RCFNDC, despite the fact that we know that its two components, *High* and *Low*, are interleaved and, therefore, cannot influence each other at all, under any circumstances, for any possible implementation of either? The only answer is that the two behaviours above must be *inconsistent* with one another. Loosely, we say that they are inconsistent because in order for them to both be exhibited, *Low* must act nondeterministically.

1.5 Why Nondeterminism Implies Inconsistency

To understand why the need for nondeterminism implies inconsistency, consider the reasons for using nondeterminism when modelling in the first place. Lowe [7] argues that nondeterminism is used for two reasons. The first is when modelling a *specification* of the behaviour of some component. Nondeterminism is used in this case to capture the various choices about how the component might be implemented in practice; each way of resolving the nondeterminism in a component corresponds to a different implementation choice that can be made. Here, a pair of behaviours that can be exhibited only by a component acting nondeterministically cannot both be exhibited by a single implementation of that component and, hence, cannot arise in a real instantiation of it.

The second reason that nondeterminism is used when modelling a component is to abstract away from low-level details of the component's behaviour. Here, different resolutions of the nondeterminism in a component correspond to different behaviours occurring within the component below the level of abstraction at which it has been modelled. Because abstraction necessarily occurs before components are composed together, interactions between components will rarely be abstracted away.

An exception to the above discussion concerns scheduling. If nondeterminism in the models of components is caused by abstracting away details of the scheduling decisions, and two components are run using a common scheduler, then it is possible

that the resolutions of nondeterminism in the two components are not independent. The analysis in this paper is not appropriate when the nondeterminism is of this form.

Under the reasonable assumption that independence properties are used to detect influence only between components (and not within them), we see that considering pairs of behaviour that can arise only from different resolutions of nondeterminism within a single component is therefore normally faulty. This is because independence properties seek to reason about whether the occurrence of an event can be affected by some other factor – such as, in the case of RCFNDC, the occurrence of high events. In observing pairs of behaviour that can arise only from a component acting nondeterministically, one has not only varied the supposed affecting factor – the occurrence, or not, of high events, say – but also the component's internal choices that are independent of the supposed affecting factor.

1.6 Contribution

In this paper, we develop a technique that allows refinement-closed independence properties to be applied to detect influence between the components of systems, Sys, of the following form

$$Sys = \Big|\Big|_{1 \leq i \leq n} (C_i, A_i),$$

that comprise a set of divergence-free components[3], $\{C_1, \ldots, C_n\}$, where each component, C_i, has an associated alphabet of events, A_i, that it is able to partake in, and which the only valid implementations can be constructed by composing refinements, C_i', of each component, C_i, to form a system $Sys' = \Big|\Big|_{1 \leq i \leq n} (C_i', A_i)$. The technique takes an existing refinement-closed independence property and weakens it to ignore all inconsistent pairs of behaviour that would ordinarily violate it.

Compositions of this sort are very general and can be used to express almost arbitrary systems of interacting components, including the example from the previous section, and have been used to model and verify various kinds of security-critical components including cryptographic protocols [14], access control policies [3], least privilege systems [10] and object-capability patterns [9]. Allowing the application of refinement-closed independence properties to detect influence between the components of these sorts of systems while ignoring inconsistent pairs of behaviour enables greater flexibility when reasoning about more novel security properties based on the notion of independence, such as information flow and causation.

This paper is organised as follows. In Section 2 we develop a formal, testable, characterisation of what it means for two behaviours to be consistent. This allows us to define the *weakened counterpart for compositions* of a refinement-closed independence property, which applies the property but rejects all inconsistent pairs of behaviour that would ordinarily violate it. In Section 3, we then show how to adapt existing tests for refinement-closed independence properties to their weakened coun-

[3] The fragment of CSP that we consider (see Appendix A) elides termination. Hence, as in common in this area, we also implicitly restrict our attention throughout this paper to ✓-free processes.

terparts, to enable their automatic verification using the CSP refinement checker FDR [4]. In Section 4, we consider related work before concluding in Section 5.

2 Our Approach

2.1 Consistency

In Section 1, we argued that pairs of behaviour that can arise only when one or more components act nondeterministically should be considered inconsistent. To be safe, we consider all other pairs of behaviour to be consistent. We now formalise that intuition.

Definition 2.1 [Consistency] Given a pair of behaviours, $s_1 \frown \langle e \rangle \frown t_1$ and $(s_2, \{e\})$, of a system $Sys = \|_{1 \leq i \leq n} (C_i, A_i)$, that violate some refinement-closed independence property, we say that these behaviours are *consistent* when there exists some process $Sys' = \|_{1 \leq i \leq n} (D_i, A_i)$ that can exhibit both behaviours, where $\forall\, 1 \leq i \leq n \bullet C_i \sqsubseteq D_i \wedge det(D_i)$. We write $Consistent(s_1 \frown \langle e \rangle \frown t_1, (s_2, \{e\}))$ in this case. We say they are *inconsistent* otherwise.

To apply a refinement-closed independence property, $\mathcal{I}$, where

$$\mathcal{I}(P) = \not\exists\, s_1, e, t_1, s_2 \bullet s_1 \frown \langle e \rangle \frown t_1 \in traces(P) \wedge (s_2, \{e\}) \in failures(P) \wedge Pred(s_1, e, s_2, t_1),$$

and $Pred(s_1, e, s_2, t_1) \Rightarrow s_1 \upharpoonright \{e\} = s_2 \upharpoonright \{e\}$, to some system $Sys = \|_{1 \leq i \leq n} (C_i, A_i)$ while rejecting inconsistent pairs of behaviour, we must therefore weaken $\mathcal{I}$ to become $\mathcal{I}_W$, where

$$\mathcal{I}_W(Sys) = \not\exists\, s_1, e, t_1, s_2 \bullet s_1 \frown \langle e \rangle \frown t_1 \in traces(Sys) \wedge (s_2, \{e\}) \in failures(Sys) \wedge Pred(s_1, e, s_2, t_1) \wedge Consistent(s_1 \frown \langle e \rangle \frown t_1, (s_2, \{e\})).$$

Unfortunately, in weakening $\mathcal{I}$ to $\mathcal{I}_W$, we have apparently made it infeasible to test, because of the implicit quantification over all refinements of the system's components that occurs in the predicate *Consistent*. In order to remedy this, we now consider an alternative, testable, characterisation of consistency and prove that it is equivalent.

2.2 Apparent Consistency

Our alternative characterisation of consistency is based on the following simple result. $Sys = \|_{1 \leq i \leq n} (C_i, A_i)$ can exhibit both $s_1 \frown \langle e \rangle \frown t_1$ and $(s_2, \{e\})$ if and only if both of the following conditions are met:

- All components involved in performing the traces $s_1 \frown \langle e \rangle \frown t_1$ and s_2 are able to

perform their events at the appropriate time, *i.e.*

$$\forall\, i \bullet (s_1 \mathbin{^\frown} \langle e \rangle \mathbin{^\frown} t_1) \upharpoonright A_i \in \mathit{traces}(C_i) \wedge s_2 \upharpoonright A_i \in \mathit{traces}(C_i),$$

- and, at least one of the components that could be involved in refusing e after s_2 is able to at the appropriate time, *i.e.*

$$\exists\, i \bullet e \in A_i \wedge (s_2 \upharpoonright A_i, \{e\}) \in \mathit{failures}(C_i).$$

The intuition, therefore, behind the following alternative characterisation of consistency is that in order for the two behaviours, $s_1 \mathbin{^\frown} \langle e \rangle \mathbin{^\frown} t_1$ and $(s_2, \{e\})$, to be consistent, there must exist some component, C_i, involved in performing and refusing e (*i.e.* for which $e \in A_i$), for whom the trace after which it must refuse e (namely $s_2 \upharpoonright A_i$) is different from that after which it must perform e (namely $s_1 \upharpoonright A_i$), in order for it to avoid having to act nondeterministically.

Definition 2.2 [Apparent Consistency] Given a pair of behaviours, $s_1 \mathbin{^\frown} \langle e \rangle \mathbin{^\frown} t_1$ and $(s_2, \{e\})$, of a system $Sys = \|_{1 \le i \le n} (C_i, A_i)$, that violate some refinement-closed independence property, we say that these behaviours are *apparently consistent* if and only if

$$\exists\, i \bullet e \in A_i \wedge s_1 \upharpoonright A_i \neq s_2 \upharpoonright A_i \wedge (s_2 \upharpoonright A_i, \{e\}) \in \mathit{failures}(C_i).$$

We will show that apparent consistency is both sufficient and necessary for consistency. We do so via a few lemmas. We begin by considering necessity.

Lemma 2.3 (Apparent consistency is necessary for consistency) *If two behaviours that violate a refinement-closed independence property are consistent, then they are apparently consistent.*

Proof. We prove the contrapositive, *i.e.* that two behaviours that are not apparently consistent are inconsistent.

Suppose we have two behaviours, $s_1 \mathbin{^\frown} \langle e \rangle \mathbin{^\frown} t_1$ and $(s_2, \{e\})$, of a system $Sys = \|_{1 \le i \le n} (C_i, A_i)$ that violate a refinement-closed independence property. Then by Definition 1.1, $s_1 \upharpoonright \{e\} = s_2 \upharpoonright \{e\}$. Suppose further that these behaviours are not apparently consistent, *i.e.* by Definition 2.2 that

$$\nexists\, i \bullet e \in A_i \wedge s_1 \upharpoonright A_i \neq s_2 \upharpoonright A_i \wedge (s_2 \upharpoonright A_i, \{e\}) \in \mathit{failures}(C_i).$$

Then, we have that

$$\forall\, i \bullet (s_1 \mathbin{^\frown} \langle e \rangle \mathbin{^\frown} t_1) \upharpoonright A_i \in \mathit{traces}(C_i)$$

and for some j, $e \in A_j \wedge (s_2 \upharpoonright A_j, \{e\}) \in \mathit{failures}(C_j)$. Then we know that for C_j, $s_1 \upharpoonright A_j = s_2 \upharpoonright A_j$. Since $e \in A_j$, we also have that $(s_1 \mathbin{^\frown} \langle e \rangle) \upharpoonright A_j = s_1 \upharpoonright A_j \mathbin{^\frown} \langle e \rangle$. Hence

$$s_1 \upharpoonright A_j \mathbin{^\frown} \langle e \rangle \in \mathit{traces}(C_j) \wedge (s_1 \upharpoonright A_j, \{e\}) \in \mathit{failures}(C_j).$$

This proves that for the two behaviours to arise, C_j must act nondeterministically. Hence, by Definition 2.1, they must be inconsistent. □

We now show that apparent consistency is not only necessary but also sufficient for inconsistency.

Lemma 2.4 (Apparent consistency is sufficient for consistency) *If two behaviours that violate a refinement-closed independence property are apparently consistent, then they are consistent.*

Proof. We prove this by showing how to construct compositions of deterministic refinements of a system's components that exhibit both behaviours.

Suppose we have two behaviours, $s_1 \frown \langle e \rangle \frown t_1$ and $(s_2, \{e\})$, of a system $Sys = \|_{1 \leq i \leq n} (C_i, A_i)$ that violate a refinement-closed independence property. Then by Definition 1.1, $s_1 \upharpoonright \{e\} = s_2 \upharpoonright \{e\}$. Suppose further that these behaviours are apparently consistent, *i.e.* by Definition 2.2 that for some j, $e \in A_j \wedge (s_2 \upharpoonright A_j, \{e\}) \in failures(C_j) \wedge s_1 \upharpoonright A_j \neq s_2 \upharpoonright A_j$.

Now, we seek to build a set of deterministic refinements, $\{D_i\}$ where $\forall\, i \bullet C_i \sqsubseteq D_i \wedge det(D_i)$, that when composed in parallel to form a system, $Sys' = \|_i (D_i, A_i)$, result in both behaviours being present in Sys', *i.e.* $s_1 \frown \langle e \rangle \frown t_1 \in traces(Sys')$ and $(s_2, \{e\}) \in failures(Sys')$. This will hold if

$$\forall\, i \bullet (s_1 \frown \langle e \rangle \frown t_1) \upharpoonright A_i \in traces(D_i) \wedge s_2 \upharpoonright A_i \in traces(D_i),$$

and $(s_2 \upharpoonright A_j, \{e\}) \in failures(D_j)$, for j above.

For the remaining D_i, $i \neq j$, we simply require each to exhibit the appropriate traces. We can achieve this simply by taking the deterministic trace-equivalent refinement of each C_i. That is, for each $i \neq j$, we define D_i to be the process that has the same stable failures as C_i, except that whenever C_i can perform $s \frown \langle e \rangle$, D_i cannot have any refusal (s, X) where $e \in X$. Lemma B.1 (in Appendix B), proves that such a process exists for all divergence-free C_i.

We construct D_j in two steps. We first remove any traces that would prevent a deterministic process from refusing e after $s_2 \upharpoonright A_j$, *i.e.* we remove all failures associated with the trace $s_2 \upharpoonright A_j \frown \langle e \rangle$ and any extension of it. Lemma B.3 (in Appendix B), shows that for any divergence-free C_j, one can always do this to arrive at a process, R_j, that refines C_j and for which $s \upharpoonright A_2 \frown \langle e \rangle \notin traces(R_j)$. Hence, all refinements of R_j (including R_j itself) must have the failure $(s_2 \upharpoonright A_j, \{e\})$. Also, because $s_1 \upharpoonright A_j \neq s_2 \upharpoonright A_j$, it must be the case that $s_1 \frown \langle e \rangle \frown t_1 \in traces(R_j)$, since it wasn't removed when forming R_j.

We then simply take D_j to be the deterministic trace-equivalent refinement of R_j. D_j is thus guaranteed to have the stable failure $(s_2 \upharpoonright A_j, \{e\})$. It will also have the trace $s_1 \frown \langle e \rangle \frown t_1$. □

The following theorem follows directly from Lemmas 2.3 and 2.4.

Theorem 2.5 *Apparent consistency is equivalent to consistency.*

2.3 Weakened Refinement-Closed Independence Properties for Compositions

We now have a testable characterisation for consistency in the form of apparent consistency, given by Definition 2.2. We can use this to appropriately weaken refinement-closed independence properties to avoid false positives created by inconsistent pairs of behaviour when applied to compositions of nondeterministic

components. This idea is formalised by the following definition.

Definition 2.6 [Weakened Independence Property] Given a refinement-closed independence property, $\mathcal{I}$, its *weakened counterpart for compositions* is denoted $\mathcal{I}_W$. Given a system of the form $Sys = \|_{1 \leq i \leq n} (C_i, A_i)$, $\mathcal{I}_W$ is defined as

$$\begin{aligned} \mathcal{I}_W(Sys) = \not\exists\, s_1, e, t_1, s_2 \bullet\ & s_1 \mathbin{\hat{}} \langle e \rangle \mathbin{\hat{}} t_1 \in traces(Sys) \wedge \\ & (s_2, \{e\}) \in failures(Sys) \wedge Pred(s_1, e, s_2, t_1) \wedge \\ & \exists\, i \bullet e \in A_i \wedge s_1 \upharpoonright A_i \neq s_2 \upharpoonright A_i \wedge \\ & (s_2 \upharpoonright A_i, \{e\}) \in failures(C_i). \end{aligned}$$

We say that a $\mathcal{I}_W$ is a *weakened refinement-closed independence property for compositions*, hereafter abbreviated to *weakened independence property.*

Notice that in doing so, some independence properties will be weakened sufficiently to become equivalent to *true*. This occurs, for example, when $Pred(s_1, e, s_2, t_1) \Rightarrow (\forall\, i \bullet e \in A_i \Rightarrow s_1 \upharpoonright A_i = s_2 \upharpoonright A_i)$. Recall that the property for determinism, *det*, can be written as a refinement-closed independence property where $Pred(s_1, e, s_2, t_1) \mathrel{\widehat{=}} t_1 = \langle\rangle \wedge s_1 = s_2$. Hence, the weakened counterpart of determinism is equivalent to *true*. This makes sense, of course, since it is precisely those behaviours that require the presence of nondeterminism that are being weeded out when an independence property is weakened.

More interestingly, perhaps, is the case of RCFNDC when applied to a system of two components with disjoint alphabets, H and L respectively. The system *Sys* from Section 1.4 is an obvious example. Recall that for RCFNDC, *Pred* can be defined as

$$e \in L \wedge t_1 = \langle\rangle \wedge ((s_1 \upharpoonright H \neq \langle\rangle \wedge s_2 = s_1 \setminus H) \vee (s_2 \upharpoonright H \neq \langle\rangle \wedge s_1 = s_2 \setminus H)).$$

Under these circumstances, for systems comprising just two components with alphabets H and L respectively, we see that *Pred* satisfies the condition above and, hence, the weakened counterpart of RCFNDC will be satisfied for all such systems. This concurs with our earlier intuition that no information flow should exist from *High* to *Low* in *Sys* from Section 1.4. By symmetry, we see that this argument applies in the other direction also, for information flows from *Low* to *High*.

This further realisation agrees with the intuition that systems built as an interleaving of components should contain no information flows between those components. In [7], Lowe notes that such systems can fail RCFNDC, if they comprise nondeterministic components, because the system can contain refinements that are insecure. Here, we have shown how to weaken RCFNDC to allow it to properly handle processes that model systems built as an interleaving of components, by rejecting all such refinements. We revisit this idea later in Section 4.

3 Automatic Testing

In this section, we consider how to adapt existing automatic tests for independence properties to produce tests for their weakened counterparts. We provide a fairly general method that can be applied to an existing test for a refinement-closed independence property like RCFNDC to produce a new test for its weakened counterpart.

3.1 Testing Independence Properties

Refinement-closed independence properties come from a larger class of properties known as *binary failures properties*. This is the class of properties that examine pairs of stable failures and are violated by the presence of certain related pairs. Lowe [6] has shown that all binary failures properties can be expressed equivalently in the form of CSP refinement tests; most of these tests are finite state, enabling them to be automatically tested by CSP's refinement checker FDR [4]. Any such property, $\phi(P)$, can be expressed in terms of a CSP refinement test that runs two copies of the process P in a test harness, $Harness(P)$, looking for pairs of failures (one from the first copy, the other from the second) that violate the property. The harness can be defined as

$$Harness(P) = (left.P \;|||\; right.P) \underset{\{|left,right|\}}{\|} Sched$$

for some deterministic *scheduler* process, *Sched*, that allows the two copies of P to exhibit all behaviours that could lead to violations of the property in question. Each copy of P performs its events on separate fresh channels, *left* and *right*, in order to allow them to be distinguished.

A specification process, *Spec*, is constructed that is the most general process that mimics the behaviour of the test harness, except that it exhibits none of the pairs of failures that violate the property. One can test whether the property holds for some process P then by testing whether

$$Spec \sqsubseteq Harness(P).$$

Lowe provides a method of deriving both *Spec* and *Harness* to ensure they are correct by construction that, although not complete, works for all cases considered to date.

Indeed, when RCFNDC and Non-Causation were first proposed (in [7] and [10] respectively, before [6]), refinement tests of this form were presented in order to facilitate the automatic verification of these properties. We now consider how such tests can be modified in order to allow weakened independence properties to be automatically tested using FDR.

We use the test for RCFNDC as an example, but the general process that we follow here can be applied with equal success to other independence properties, such as Non-Causation, in order to express their weakened counterparts in terms of refinement tests. The following is a slight adaptation of the *Spec* process derived in [6] for RCFNDC.

$$Spec = \left(\begin{array}{l} left?h : H \to Spec' \\ \Box\ left?l : L \to (right.l \to Spec \sqcap STOP) \\ \Box\ right?l : L \to (left.l \to Spec \sqcap STOP) \end{array}\right) \sqcap STOP$$

$$Spec' = \left(\begin{array}{l} left?h : H \to Spec' \\ \Box\ left?l : L \to right.l \to Spec' \\ \Box\ right?l : L \to left.l \to Spec' \end{array}\right) \sqcap STOP$$

The test works as follows. The scheduler (not shown), *Sched*, is the deterministic trace-equivalent refinement (see Lemma B.1 of Appendix B) of *Spec* obtained by removing all of *Spec*'s "$\sqcap\ STOP$" clauses. *Sched* allows the *left* copy of P to perform events in H and L. The *right* copy of P is allowed to perform only L events. Both copies must perform the same L events. The first specification process, *Spec*, corresponds to states in which no H event has yet been performed by the *left* copy of P. Once an H event is performed, the specification evolves to *Spec*′. *Spec*′ is constructed to ensure that **(0)** assuming both copies perform the same L events, that once some H event has been performed by the *left* copy, **(1)** whenever an L event is performed by either copy of P, **(2)** the other copy cannot refuse it. This corresponds exactly to the definition of RCFNDC, which is equivalent to

$$\nexists\, s_1, e, s_2 \bullet (\mathbf{1})\ s_1 \mathbin{^\frown} \langle e \rangle \in traces(P) \wedge e \in L \wedge (\mathbf{2})\ (s_2, \{e\}) \in failures(P) \wedge (\mathbf{0})\ ((s_1 \upharpoonright H \neq \langle\rangle \wedge s_2 = s_1 \setminus H) \vee (s_2 \upharpoonright H \neq \langle\rangle \wedge s_1 = s_2 \setminus H)).$$

3.2 Testing Weakened Independence Properties

We now show a fairly general method that can be applied to adapt a refinement test for an independence property, like that above for RCFNDC, to instead test for its weakened counterpart.

We begin by observing that we can rewrite a weakened independence property $\mathcal{I}_W$ (see Definition 2.6) equivalently as

$$\begin{aligned} \mathcal{I}_W(Sys) = \nexists\, s_1, e, t_1, s_2 \bullet\ & s_1 \mathbin{^\frown} \langle e \rangle \mathbin{^\frown} t_1 \in traces(Sys) \wedge s_2 \in traces(Sys) \wedge \\ & Pred(s_1, e, s_2, t_1) \wedge \\ & \exists\, i \bullet e \in A_i \wedge s_1 \upharpoonright A_i \neq s_2 \upharpoonright A_i \wedge \\ & (s_2 \upharpoonright A_i, \{e\}) \in failures(C_i). \end{aligned}$$

Hence, weakened refinement-closed independence properties are violated by the presence of three behaviours: two traces of the system *Sys* and one stable failure of one of the system's components.

At first glance, this suggests that in order to express such a property as a refinement test, we would need to create a test harness that runs two copies of the process *Sys* (in order to test if the two traces can be exhibited), as well as a copy of each component of *Sys* (in order to test if the stable failure can be exhibited).

However, we can achieve the same result by simply running two copies of each component of *Sys*. We run two copies of a modified system, *WSys*, that allows us to observe both system-level traces as well as the behaviour of individual components by applying a renaming to each component before composing them. Given a system, $Sys = \left|\left|\right|_{1 \leq i \leq n} (C_i, A_i)$, we can create a process, *WSys*, that allows us to observe both system-level traces and individual component behaviours as follows.

$$WSys = \left|\left|\right|_{1 \leq i \leq n} (C_i[\![^{sys.x,\, cmp.C_i.x}/_{x,\, x}]\!], \{sys.x, cmp.C_i.x \mid x \in A_i\}).$$

Here *sys* and *cmp* are fresh channels over which *WSys* performs system-level and individual component events respectively. $C_i[\![^{sys.x,\, cmp.C_i.x}/_{x,\, x}]\!]$ is the process that can perform either of the events $sys.x$ or $cmp.C_i.x$, whenever C_i can perform the event x. The alphabetised parallel composition forces all of the transformed C_i to synchronise on all events performed on the channel *sys*, while events performed on any of the *cmp* channels occur without any synchronisation. This means that we can observe system-level traces by observing the *sys* channel. At some point in time, we can then observe the behaviour of individual components by observing the various *cmp* channels.

This allows us to build a modified test harness, *WHarness*, that can be applied to a system $Sys = \left|\left|\right|_{1 \leq i \leq n} (C_i, A_i)$ as follows.

$$WHarness(Sys) = (left.WSys \;|||\; right.WSys) \underset{\{|left, right|\}}{\|} WSched.$$

WSched is explained shortly, as it is derived from the specification process, *WSpec*, against which *WHarness* is tested for refinement. *WSpec* is constructed by adapting *Spec* as follows. In general, the specification process, *Spec*, for a (non-weakened) independence property applied to some process P, is constructed so that whenever the *left* (respectively *right*) copy of P performs a trace $s_1 \,\hat{}\, \langle e \rangle \,\hat{}\, t_1$ and the *right* (respectively *left*) copy of P performs the related trace s_2, *Spec* never refuses the event $right.e$ (respectively $left.e$). Hence, it will be refined by $Harness(P)$ if and only if the *right* (respectively *left*) copy of P cannot refuse the event e after performing the trace s_2. We call the states in which *Spec* never refuses the event $right.e$ (respectively $left.e$) above, the *critical* states.

For a weakened independence property applied to some composition, $Sys = \left|\left|\right|_{1 \leq i \leq n} (C_i, A_i)$, we require that, once in a critical state, rather than the other copy of *Sys* not being able to refuse the event e, that instead none of the individual *components*, C_i, that have e in their alphabets ($e \in A_i$) and have performed different traces ($s_1 \upharpoonright A_i \neq s_2 \upharpoonright A_i$) can refuse e. This gives us a recipe for adapting *Spec*, to make *WSpec*, as follows.

- Change all references to the channels *left* and *right* to *left.sys* and *right.sys* respectively.
- Maintain a set, S, of components that have performed different traces so far, *i.e.* for whom $s_1 \upharpoonright A_i \neq s_2 \upharpoonright A_i$; S is initially empty.
- For each system-level event that is performed, determine which components have now performed different traces, and add those to the set S.

- The critical states in which *Spec* could not refuse the event $right.e$ (respectively $left.e$) are expressed as $right.e \rightarrow Q$ for some process Q. Have *WSpec* instead do $right.sys.e \rightarrow Q \triangleright NR(right, S, e)$ (respectively $left.sys.e \rightarrow Q \triangleright NR(left, S, e)$), where

$$NR(chan, S, e) = ?a : \{chan.cmp.C_i.e \mid C_i \in S \wedge e \in A_i\} \rightarrow STOP.$$

  This instead allows *WSpec* to perform (but also refuse) $right.sys.e$ (respectively $left.sys.e$) while preventing it from refusing any of the events $right.cmp.C_i.e$ (respectively $left.cmp.C_i.e$) for all C_i that have performed different traces so far and that have e in their alphabets. We have *WSpec* become $STOP$ after refusing none of the $right.cmp.C_i.e$ (respectively $left.cmp.C_i.e$) since at this point only a subset of the components that have e in their alphabets may have performed e and, hence, the composition might have lost synchronisation. Note that the scheduler, *WSched*, explained directly, also becomes $STOP$ in the corresponding state to ensure the test remains sound.

WSched is formed by simply taking the deterministic trace-equivalent refinement of *WSpec*, which can usually be derived syntactically as shown below. The test is carried out, then, by testing the refinement

$$WSpec \sqsubseteq WHarness(P).$$

In order to apply the recipe to develop *WSpec* from *Spec*, we simply need to determine how to update the set S after each system-level event has been performed. We use the test for RCFNDC as an example. Observe that with RCFNDC, it is only the events from H that can add differences between s_1 and s_2, since both copies of the system perform the same L events. Hence, for each $h \in H$ that is performed, for all i, $s_1 \upharpoonright A_i \neq s_2 \upharpoonright A_i$, if and only if $h \in A_i$. Therefore, S needs to be updated only for each H event, h, that is performed by simply adding all of the C_i for whom $h \in A_i$; we denote this set by $cmpsWith(h)$:

$$cmpsWith(h) = \{C_i \mid 1 \leq i \leq n \wedge h \in A_i\}.$$

This leads to the following definition of *WSpec* for RCFNDC:

$$WSpec = \begin{pmatrix} left.sys?h : H \rightarrow WSpec'(cmpsWith(h)) \\ \Box\ left.sys?l : L \rightarrow (right.sys.l \rightarrow WSpec \sqcap STOP) \\ \Box\ right.sys?l : L \rightarrow (left.sys.l \rightarrow WSpec \sqcap STOP) \end{pmatrix} \sqcap STOP,$$

$$WSpec'(S) = \begin{pmatrix} left.sys?h : H \rightarrow WSpec'(S \cup cmpsWith(h)) \ \Box \\ left.sys?l : L \rightarrow (right.sys.l \rightarrow WSpec'(S) \triangleright NR(right, S, l)) \\ \Box\ right.sys?l : L \rightarrow (left.sys.l \rightarrow WSpec'(S) \triangleright NR(left, S, l)) \end{pmatrix} \sqcap STOP.$$

As stated earlier, *WSched* is the deterministic trace-equivalent refinement of *WSpec*. It can be derived syntactically from *WSpec* by removing all "$\sqcap\ STOP$" clauses and replacing all "$\triangleright$" symbols with "$\Box$" instead.

4 Related Work

In Section 3.1, we said that refinement-closed independence properties form part of a class of properties known as *binary failures properties* [7]. These are properties that are violated by pairs of stable failures. Our weakened independence properties are not binary failures properties because, in general, one cannot determine from two system-level behaviours alone whether they are inconsistent. Whether two behaviours are inconsistent depends on how the system has been constructed. This is why we need to consider the behaviours of individual components in order to determine whether two system-level behaviours are inconsistent.

In Section 2.3, we showed that the weakened counterpart of RCFNDC does not detect information flows between components *High* and *Low* of a system $Sys = High \ {}_H\|_L \ Low$ where H and L are disjoint. In [12], Roscoe and Wulf consider whether systems that can be expressed as the composition of two components with disjoint alphabets can contain information flows. They note, as did Lowe in [7], that systems containing nondeterministic components can be refined by others that do contain information flows. When all refinements represent valid implementations, then the original refinement-closed independence property should be applied. Its weakened counterpart, on the other hand, should be applied only when all real implementations can be constructed by composing refinements of the system's components.

Our approach could be considered to be akin to defining an alternate refinement relation $\preccurlyeq$, that for a process $Sys = \|_{1 \leq i \leq n}(C_i, A_i)$, is defined as

$$Sys \preccurlyeq Sys' \Leftrightarrow Sys' = \|_{1 \leq i \leq n}(D_i, A_i) \wedge \forall\, i \bullet C_i \sqsubseteq D_i.$$

A system is then secure if and only if all of its maximal refinements under $\preccurlyeq$ are secure. The idea of defining alternative refinement relations in the context of security properties is not new. In particular, Mantel [8] has shown how to define alternative refinement relations over trace sets that preserve information flow properties that are ordinarily not refinement-closed. Under these alternative traces refinement relations, a secure system can only be refined by other secure systems. Others (*e.g.* [2] and [1]) have also considered the problem of constructing security-preserving refinements. Here, however, the refinement relation is defined between terms of an operational semantics (namely labelled transition systems), rather than between terms of a denotational semantics.

These approaches subordinate the refinement-relation to the security property, while employing a refinement-closed property subordinates the security property to the refinement-relation. Our approach differs from both of these in that we consider the form of the original composition to be superior and, depending on one's point of view, we subordinate either the refinement relation or the security property to it.

Refinement-closed independence properties naturally test whether certain events occur independently of some potential affecting factor. In this sense, they can be viewed as non-causation properties that assert that the potential affecting factor cannot cause certain events to, or to not, occur. It could be argued that their

complexity comes about from trying to reconstruct causal information solely from a system's traces and failures. Other denotational semantics, however, more readily capture a notion of causation. In particular, models that represent concurrency without resorting to interleaved execution traces – so-called "true" concurrency semantics – usually incorporate information about the causal relationship between events. *Event structures* [16] are a notable example with many variations.

Unfortunately, using typical event structure encodings of CSP-like languages yields causal information that is less useful for our purposes than one would like. For example, under an encoding using *bundle event structures* (as in [15]), event h causes event l in system trace s when h comes before l in all system traces containing the same events as s [5]. This means that, under such an encoding, in the process $h \rightarrow l \rightarrow STOP \;\Box\; l \rightarrow STOP$, h would be identified as a unique cause of l in the trace $\langle h, l \rangle$. This conflicts with the usual intuition that no information can flow from High to Low in this process, assuming natural alphabets for both. The problem with the notion of causation that is typically employed in these sorts of semantics is that it ignores whether l can occur without h having first occurred. In contrast, this is at the heart of the notion of independence considered in this paper. Overcoming this limitation appears to be no less complicated than extracting causal information from sets of traces and failures (see *e.g.* [11]).

5 Conclusion

In this paper we have considered the problem of applying refinement-closed security properties that examine pairs of behaviours of systems composed from nondeterministic components. In particular, we have considered *refinement-closed independence properties* that detect influence between components by determining whether all occurrences of some events are independent of some other factor. These include properties to detect information flows and causation. In cases in which these systems are implemented only by composing refinements of their components, standard refinement-closed independence properties will detect *inconsistent* pairs of behaviour that could not arise in a real implementation and, hence, lead to false positives. We have shown how to weaken such properties to allow them to be applied to these sorts of systems, rejecting all pairs of inconsistent behaviour that would have ordinarily violated the property. We have also shown how to adapt existing refinement tests for refinement-closed independence properties to allow them to be used to test for their weakened counterparts. This enables greater flexibility in the application of these sorts of properties to compositions of nondeterministic components.

Acknowledgement

We would like to thank the anonymous reviewers for their comments and feedback.

References

[1] Alur, R., P. Cerný and S. A. Zdancewic, *Preserving secrecy under refinement*, in: *Proceedings of the 33rd International Colloquium on Automata, Languages and Programming, LNCS 4052*, 2006, pp. 107–118.

[2] Bossi, A., R. Focardi, C. Piazza and S. Rossi, *Refinement operators and information flow security*, in: *Proceedings of the IEEE International Conference on Software Engineering and Formal Methods (SEFM'03)*, 2003, pp. 44–53.

[3] Bryans, J., *Reasoning about XACML policies using CSP*, in: *SWS '05: Proceedings of the 2005 workshop on Secure web services* (2005), pp. 28–35.

[4] Formal Systems (Europe) Limited, "Failures Divergences Refinement: FDR2 User Manual," (2005), available at: http://www.fsel.com/documentation/fdr2/fdr2manual.ps.

[5] Langerak, R., E. Brinksma and J.-P. Katoen, *Causal ambiguity and partial orders in event structures*, in: *Proceedings of the 8th International Conference on Concurrency Theory (CONCUR '97)* (1997), pp. 317–331.

[6] Lowe, G., *On CSP refinement tests that run multiple copies of a process*, in: *Proceedings of the Seventh International Workshop on Automated Verification of Critical Systems, AVoCS '07*, 2007.

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

[8] Mantel, H., *Preserving information flow properties under refinement*, in: *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, 2001, p. 78.

[9] Murray, T., *Analysing object-capability security*, in: *Proceedings of the Joint Workshop on Foundations of Computer Security, Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security (FCS-ARSPA-WITS'08)*, 2008, pp. 177–194.

[10] Murray, T. and G. Lowe, *Authority analysis for least privilege environments*, in: *Proceedings of the Joint Workshop on Foundations of Computer Security and Automated Reasoning for Security Protocol Analysis (FCS-ARSPA'07)*, 2007, pp. 113–130.

[11] MyThS, *Models of information flow based on non-interleaving, causal models of concurrency* (2005), available at: http://www.informatics.sussex.ac.uk/users/vs/myths/deliver/mythsD1.3.pdf.

[12] Roscoe, A. and L. Wulf, *Composing and decomposing systems under security properties*, in: *Proceedings of the Eighth IEEE Computer Security Foundations Workshop*, 1995, pp. 9–15.

[13] Roscoe, A. W., "The Theory and Practice of Concurrency," Prentice Hall, Upper Saddle River, NJ, USA, 1997.

[14] Ryan, P., S. Schneider, M. Goldsmith, G. Lowe and B. Roscoe, "Modelling and Analysis of Security Protocols: the CSP Approach," Addison Wesley, 2000.

[15] van Glabbeek, R. and F. Vaandrager, *Bundle event structures and CCSP*, in: *Proceedings of the 14th International Conference on Concurrency Theory (CONCUR '03)* (2003), pp. 57–71.

[16] Winskel, G., *An introduction to event structures*, in: *REX School of Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency* (1989), pp. 364–397.

A A Brief Overview of CSP

In this appendix, we present a brief overview of the fragment of CSP that is used in this paper. In particular, this fragment elides termination. Further details about CSP can be found in [13].

CSP is used to describe processes that, when composed in parallel, can communicate with each other by synchronising on common events. The occurrence of an event can be thought of as an atomic communication between the processes that synchronise on it. An event that is synchronised on by a set of processes can be performed only when all of the processes in the set agree to perform it and is refused otherwise.

We say that a process *offers* the set of events A, if it can perform all of the events in A and can refuse none of them. The *alphabet* of a process is the set of events that it can perform and, hence, in which it can partake.

A.1 Syntax

The process $STOP$ performs no events; it refuses everything. The process $a \rightarrow P$ offers the event a. If a is performed, it acts like P. The process $?a : A \rightarrow P_a$ offers the set A. If a specific event a from A is performed, it then acts like P_a. The "$\rightarrow$" operator binds tighter than all others.

CSP allows multi-part events where each part is separated by an infix dot ".", such as the event $\mathsf{up}.3$. The process $\mathsf{up}?a : A \rightarrow P_a$ initially offers the set of events $\{\mathsf{up}.a \mid a \in A\}$. The output operator "!" is used to offer specific events. The process $\mathsf{move}?x{:}X!3 \rightarrow P$ initially offers the set of events $\{\mathsf{move}.x.3 \mid x \in X\}$. The notation $\{|\mathsf{move}|\}$ denotes the set of events whose first part is move. The first part of a multi-part event is sometimes called a *channel*.

The process $P \,\Box\, Q$ represents an external choice between P and Q; the initial events of both processes are offered; when an event is performed, the choice is resolved. $P \sqcap Q$ represents an internal or nondeterministic choice between P and Q; the process can act like either P or Q, with the choice being made according to some criteria that we do not model. If P offers A and Q offers B, then, unlike $P \,\Box\, Q$, $P \sqcap Q$ can offer either A or B but not both. $P \triangleright Q$ is the process that can act like either P or Q, except that it offers only the initial events of Q; it can refuse the initial events of P.

If c is a channel, then $c.P$ represents the process that acts like P except every event x is renamed to $c.x$. The process $P[\![a_1, \ldots, a_n / b_1, \ldots, b_n]\!]$ represents the process that acts like P except that for all $1 \leq i \leq n$, whenever P offers b_i, it instead offers a_i.

$P \underset{A}{\parallel} Q$ represents the parallel composition of P and Q, synchronising on events from A. For a set of processes, $\{P_1, \ldots, P_n\}$, and a set of alphabets $\{A_1, \ldots, A_n\}$, $\Big\|_{1 \leq i \leq n} (P_i, A_i)$ represents the parallel composition of the P_i, where each P_i is allowed to perform events only from the set A_i, and all processes that share a common event must synchronise on it. $P_1 \;{}_{A_1}\|_{A_2}\; P_2$ is equivalent to $\Big\|_{1 \leq i \leq 2} (P_i, A_i)$. $P \,|||\, Q$ represents the interleaving of P and Q, i.e., parallel composition with no synchronisation.

A.2 Stable Failures Semantics

Recall that the stable failures denotational semantic model of CSP represents a process P by its traces, $traces(P)$ and stable failures, $failures(P)$. We write Σ for the set of visible events. For any CSP process P, $traces(P) \subseteq \Sigma^*$ and $failures(P) \subseteq \Sigma^* \times \mathbf{P}(\Sigma)$. For a divergence-free process P, $traces(P) = \{s \mid (s, X) \in failures(P)\}$. Hence, within the stable failures model, a divergence-free process is completely characterised by its stable failures. In this case, its stable failures F and traces,

$T = \{s \mid (s, X) \in F\}$, satisfy the following axioms.

F1. T is non-empty and prefix closed.

F2. $(v, X) \in F \wedge Y \subseteq X \Rightarrow (v, Y) \in F$.

F3. $(v, X) \in F \wedge v \,\hat{}\, \langle a \rangle \notin T \Rightarrow (v, X \cup \{a\}) \in F$.

B Subsidiary Proofs

Here, we prove two results about how one can refine a divergence-free process. We first show (in Lemma B.1) that one can refine a divergence-free process P to reach a deterministic process Q that has the same traces as P. We then show (in Lemma B.3) that if a process has a failure $(s, \{e\})$, it can be refined to a process that does not have the trace $s \,\hat{}\, \langle e \rangle$. This proof is a simple generalisation of one from earlier work with Gavin Lowe [10].

Lemma B.1 (Trace-equivalent deterministic refinements) *Every process P that is both ✓-free and divergence-free has a deterministic refinement, Q, that is trace equivalent to it.*

Proof. Suppose we have a ✓- and divergence-free process, P. We mechanically define the set of stable-failures of Q, F_Q, as follows, with Q's traces, $T_Q = \{s \mid (s, X) \in F_Q\}$.

$$F_Q = \mathit{failures}(P) - \{(s, X) \mid \exists\, e \bullet s \,\hat{}\, \langle e \rangle \in \mathit{traces}(P) \wedge e \in X\}$$

Clearly, the F_Q is contained in $\mathit{failures}(P)$. Also, since we never remove any stable failure $(s, \{\})$, $T_Q = \mathit{traces}(P)$. Also, there exists no stable-failure $(s, \{e\}) \in F_Q$, for which $s \,\hat{}\, \langle e \rangle \in T_Q$. Hence, if there exists some process Q that has these traces and failures, then this Q will be a deterministic, trace-equivalent refinement of P.

In order to show that such a Q exists, we need the following result.

Theorem B.2 *Assuming the alphabet Σ is finite, for any choice of F and $T = \{s \mid (s, X) \in F\}$ that satisfies the axioms (see Section A) of the stable failures model, there is a CSP process whose traces and stable failures are T and F respectively.*

Hence it will be enough to show that T_Q and F_Q satisfy the axioms **F1**–**F3** of the stable failures model.

Axiom F1. Clearly T_Q is non-empty and prefix-closed since it is equal to $\mathit{traces}(P)$, which satisfies these conditions.

Axiom F2. Q satisfies **F2** since P does, and whenever we remove a failure, we remove all failures with larger refusal sets.

Axiom F3. We prove this by contradiction. Suppose $(v, X) \in F_Q$ and $v \,\hat{}\, \langle a \rangle \notin T_Q$ but $(v, X \cup \{a\}) \notin F_Q$. Since the traces and failures of P satisfy this axiom and we remove none of P's traces when forming T_Q, it must be the case that $(v, X \cup \{a\})$ was one of the failures removed from $\mathit{failures}(P)$. Hence, there must exist some e, where $s \,\hat{}\, \langle e \rangle \in \mathit{traces}(P) \wedge e \in X \cup \{a\}$. Since $(v, X) \in F_Q$ by assumption, we have

that $\forall\, e' \in X \bullet v \mathbin{\hat{}} \langle e' \rangle \notin \mathit{traces}(P)$ or else this failure would have been removed. Hence, it must be the case that $e = a$, *i.e.* that $v \mathbin{\hat{}} \langle a \rangle \in \mathit{traces}(P)$, equivalently $v \mathbin{\hat{}} \langle a \rangle \in T_Q$, which clearly contradicts our assumptions. Hence this axiom must be satisfied. □

Lemma B.3 *Given a divergence-free process P, and some failure $(s, \{e\}) \in \mathit{failures}(P)$, there exists a refinement, Q, of P that does not have the trace $s \mathbin{\hat{}} \langle e \rangle$ and whose failures are defined as follows:*

$$\begin{aligned} F_Q = \mathit{failures}(P) - \\ \{(s \mathbin{\hat{}} \langle e \rangle \mathbin{\hat{}} t, X) \mid t \in \Sigma^*, X \subseteq \Sigma\} - \\ \{(s, X) \mid (s, X \cup \{e\}) \notin \mathit{failures}(P)\}. \end{aligned}$$

The traces of Q are simply defined as $T_Q = \{s \mid (s, X) \in F_Q\}$.

Proof. We must show that T_Q and F_Q satisfy the axioms **F1**–**F3** of the stable failures model.

Axiom F1. Clearly T_Q is non-empty: it contains, at least, the empty trace. It is prefix-closed since $\mathit{traces}(P)$ is, and we remove an extensions-closed set of traces.

Axiom F2. Q satisfies **F2** since P does, and whenever we remove a failure, we remove all failures with larger refusal sets.

Axiom F3. Suppose $(v, X) \in F_Q$ and $v \mathbin{\hat{}} \langle a \rangle \notin T_Q$. Then $(v, X) \in \mathit{failures}(P)$. We perform a case analysis.

- Case $v \mathbin{\hat{}} \langle a \rangle \neq s \mathbin{\hat{}} \langle e \rangle$. Then $v \mathbin{\hat{}} \langle a \rangle \notin \mathit{traces}(P)$, and so $(v, X \cup \{a\}) \in \mathit{failures}(P)$, since P satisfies **F3**. And hence $(v, X \cup \{a\}) \in F_Q$, by construction.
- Case $v = s \wedge a = e$. Recall that $(s, \{e\}) \in \mathit{failures}(P)$. Hence $(s, X \cup \{e\}) \in \mathit{failures}(P)$. And hence $(v, X \cup \{a\}) = (s, X \cup \{e\}) \in F_Q$, by construction.

□