Analysing the Security Properties of Object-Capability Patterns

Toby Murray

Hertford College

Oxford University Computing Laboratory

A thesis submitted for the degree of

Doctor of Philosophy

Hilary Term, 2010

This thesis is dedicated to
Bel,
who has taught me more than anyone else.

Acknowledgements

Thanks firstly to my supervisor, Gavin Lowe, whose conscientious and dedicated supervision of this work improved it, and its presentation, more than I can say, and whose wisdom and guidance were invaluable throughout this entire process. I couldn't have asked for a better supervisor. Gavin also contributed directly to the development of valuable parts of this work, as explained in the statement of authorship that follows.

Thanks to Fred Spiessens and Bill Roscoe for examining this thesis. Thanks to Andrew Simpson and Bill Roscoe for their advice and feedback on this work as it progressed during my transfer and confirmation of status.

Thanks to Bill Roscoe for valuable discussions about CSP and to Fred Spiessens for the same regarding Scoll and authority.

Thanks to the anonymous reviewers of [ML09b] and [ML09a], and those at NICTA, whose feedback helped to improve the presentation in Chapter 5.

Thanks to David Wagner for useful discussions about authority and causation that ultimately influenced the work in Chapter 7, and for indirectly bringing to my attention the need to consider vulnerabilities due to recursive invocation.

Thanks to Bill Frantz, David-Sarah Hopwood, Matej Košík, Charles Landau, Alex Murray, Mark Seaborn and David Wagner, who helped me assemble Table 2.1.

Thanks to all of the people on cap-talk and e-lang, including Mark Miller and Jonathan Shapiro, and all of my old workmates from DSTO's Annex project. You have greatly influenced my thinking about capability systems and have undoubtedly influenced the work contained in this thesis.

My greatest thanks to Duncan Grove and Chris North, and also to Dave Munro, Rob Esser, Ed Lowrey and Ken Yiu, for encouraging and facilitating me to undertake this doctorate.

Thanks so much to my family and friends, both in Oxford and back home, for their love and support. Thanks especially to Mark Larsen, whose friendship helped make Oxford home.

Thanks finally to my wife, Bel, for her unwavering support during times of crisis, and her sacrifice throughout this adventure. She knew from the beginning that Oxford was the place to be.

This work was supported by a John Crampton Travelling Scholarship, for which I am extremely grateful.

Statement of Authorship

The following parts of this thesis contain work carried out jointly with my supervisor, Gavin Lowe.

- The use and form of the **PosConjEqT** safe abstraction of the Trademarks guard object, which appears in Section 4.1.3, was conceived by Gavin Lowe.
- Gavin Lowe also conceived the technique used to model and analyse the Membrane in the concurrent context, which appears in Section 4.3.4.
- The proofs of Lemmas 5.3.6 and 5.3.7 were adapted from a proof that was co-written with Gavin Lowe and appears in [ML07].
- The basic structure of the refinement check for testing weakened refinement-closed noninterference properties, which appears in Section 5.3.3, was conceived by Gavin Lowe. This work also appears in [ML09b].
- The proof of Theorem 5.5.3 is a simple generalisation of a proof that was conceived by Gavin Lowe. This work also appears in [ML09a].
- The proof of Lemma A.0.6 is a simple extension of a proof that was co-written with Gavin Lowe and appears in [ML07].

Analysing the Security Properties of Object-Capability Patterns

Toby Murray

Hertford College
University of Oxford

A thesis submitted for the degree of Doctor of Philosophy

Hilary Term, 2010

The *object-capability model* is an increasingly popular architecture for building secure software systems. This model promotes the construction of reusable *patterns* for enforcing security properties within object-capability systems. In this thesis, we apply the process algebra CSP, and its automatic *refinement-checker* FDR, to analyse object-capability patterns and prove whether they uphold the security properties they are designed to enforce.

We show how CSP can accurately model object-capability systems and patterns, and express their wide variety of features.

We show that complex *safety properties* of object-capability patterns can be reasoned about by encoding them as CSP refinement checks for FDR. This enables one to detect vulnerabilities automatically in patterns due to concurrent and recursive invocation.

We show that CSP's theory of *data-independence* can be applied to allow one to generalise the results obtained from analysing small fixed-sized systems, to systems of arbitrary size.

We show how to reason about the *information flow properties* of object-capability patterns. We argue that in order to do so sensibly, one must make the assumption that objects can directly influence each other only through their overt interactions together. We show how traditional noninterference properties can be adapted to take this assumption into account, and how they can then be tested with FDR.

We consider how to reason about *liveness* properties of object-capability patterns under necessary *fairness* assumptions. We prove that such properties cannot always be expressed as CSP refinement checks for FDR, making them impossible for FDR to test precisely, but how FDR can be applied to reason about them by testing sufficient conditions for them instead.

To reason about *authority*, we develop a framework for expressing general *non-causation* properties and show how it can capture various kinds of authority, as well as the notions of *defensive correctness* and *defensive consistency*. We show that, for deterministic systems, non-causation of *safety* effects can be expressed as refinement checks in CSP models that FDR can support. However, for nondeterministic systems, we prove that even certain simple non-causation properties cannot be precisely captured this way.

Engineers in established fields use applied mathematics to predict the behavior and properties of their designs with great accuracy. Software engineers, despite the fact that their creations exhibit far more complexity than physical systems, do not generally do this and the practice of the discipline is still at the pre-scientific or craft stage. . . . [T]he applied mathematics of software is formal logic, and calculating the behavior of software is an exercise in theorem proving. Just as engineers in other disciplines need the speed and accuracy of computers to help them perform their engineering calculations, so software engineers can use the speed and accuracy of computers to help them prove the . . . theorems required to predict the behavior of software.

John Rushby [Rus89].

Contents

List of CSP Snippets

1 Introduction

1.1 The Problem

The *object-capability model* [Mil06] is becoming an increasingly popular architecture for the construction of secure software systems. Several current research projects, including secure programming languages like E [Mil06], Joe-E [MWC10] and Cajita [MSL+08], and microkernel operating systems like the Annex Capability Kernel [GMO+07] and seL4 [EKE08, DEE08], implement the object-capability model to provide platforms on which secure software systems may be constructed from untrusted components.

The object-capability model enables this alchemical-sounding feat in two ways. Firstly, it explicitly reifies all permissions in the form of delegable, unforgeable *capabilities* [DH66]. A subject is permitted to interact directly with only those entities for which it possesses capabilities, and subjects possess no capabilities by default. Hence, untrusted components may be granted few capabilities, in accordance with the *principle of least authority* [Mil06][1], thereby limiting the damage they can potentially cause. Secondly, the object-capability model allows the construction of trusted *security enforcing abstractions* [Mil06] that mediate the interactions of untrusted components. This enables one to enforce and express security policies of wide generality, while also allowing different stakeholders that may be present in a system to each enforce their own security requirements by deploying their own security-enforcing abstractions.

An *object-capability system* is an instance of the object-capability model and comprises a set of *objects* connected to each other by *capabilities*. Objects communicate with each other by sending messages on capabilities. Any such system may be visualised as a directed graph, whose nodes are the system's objects and each edge from an object o_1 to another o_2 represents o_1 possessing a capability that allows it to send messages to o_2.

Consider an object-capability system and some stakeholder who wishes for some security requirements to be upheld in this system. Naturally, some of the objects in the system will be trusted by the stakeholder, while others will be untrusted. The stakeholder will rely upon the objects he or she

[1] The principle of least authority is a refinement of Saltzer and Schroeder's *principle of least privilege* [SS75] that takes into account the indirect effects a subject may cause.

trusts to enforce their security requirements and may view the system as a collection of trusted islands in a sea of untrusted components. This thesis is concerned with the problem of how the stakeholder can ensure that their security requirements will be upheld in such a system, when relying only on the objects that he or she trusts.

These relied-upon objects may be viewed as one or more *security-enforcing abstractions* that together ensure that the stakeholder's security requirements are met. Like functional abstractions, a security enforcing abstraction may be reused, and be deployed whenever the security property it enforces needs to be ensured. In this sense, each security-enforcing abstraction may be viewed as a security *design pattern* [GHJV95]. For this reason, we refer to object-capability-based security-enforcing abstractions as *object-capability patterns*. The question we wish to answer, therefore, is "given an object-capability pattern, how can we analyse this pattern to prove (or disprove) that it upholds the security properties it was designed to enforce?"

1.2 Our Approach

This thesis examines the application of the process algebra Communicating Sequential Processes [Hoa85, Ros97] (CSP) and its automatic model-checker FDR [Ros94, RGG+95, GGH+05] to this problem. For a particular pattern, and the security property it is designed to enforce, our approach is to construct a CSP model of an object-capability system that contains the pattern: we explicitly model the behaviour of the relied-upon objects that implement the pattern while modelling the behaviour of the other untrusted objects as being arbitrary. We then use FDR to analyse this formal model automatically to prove whether the pattern upholds its security property. This is achieved by encoding security properties in the form of CSP *refinement* assertions, which FDR can check automatically.

This approach is largely inspired by Spiessens' previous work [Spi07, SV05], in which the Scoll language [JSV05] and its Scollar model-checker [SJV05] were conceived, implemented and applied to this same problem[2]. In adopting CSP and FDR, we aim to improve upon Spiessens' work by extending the kind of security properties one can easily reason about, as well as increasing the level of detail at which one can model the relied upon objects in an object-capability pattern and, therefore, the utility of the approach overall.

We use CSP and FDR, over alternative formalisms and tools, for the following reasons. Firstly, object-capability systems come in many flavours; we need a formalism that is capable of accurately modelling both single-threaded and concurrent systems, as well as phenomena like recursive and

[2] Unlike ours, Spiessens' work also considered how to automatically derive formal models for the behaviour of the relied-upon objects in a pattern so as to uphold a particular security property; however, that problem is beyond the scope of this thesis.

concurrent reentrancy. As we will see in this thesis, CSP's natural expressiveness means that it is well suited to these tasks. Secondly, we wish to reason about a range of security properties. This requires a formalism with a rich semantic framework within which such properties can be expressed. Perhaps CSP's greatest strengths are its incredibly rich arsenal of denotational semantic models and its supporting body of theory, in which one can express and verify a very wide range of security properties. CSP's notion of *refinement* is particularly useful for expressing certain subtle security properties (such as the *weakened refinement-closed noninterference properties* that we define in Chapter 5).

Thirdly, we require a formalism for which tools exist to automatically check that some semantic property is true of a system modelled in that formalism. The FDR tool is particularly suited to this task. It automatically tests properties framed in terms of CSP refinement assertions which, coupled with CSP's various semantic models, are able to express a wide variety of properties for automatic verification [Ros05]. It is the existence of FDR, CSP's arsenal of denotational semantic models and theory of refinement, and CSP's rich body of supporting theory (*e.g.* that of *data-independence* [Laz99]), that make CSP preferable to other process algebras and related formalisms.

Finally, CSP has an excellent track-record for being successfully applied to reason about the security properties of many security-relevant systems, including cryptographic protocols [Low96, RSG+00], access control models [KN06, Bry05], operating system kernels [KT09], intrusion-detection systems [RL05] and real-world security policies [RA03].

1.3 Contribution and Thesis Organisation

The main contribution of this thesis is to demonstrate how CSP and FDR can be applied to reason about a range of relevant security properties of object-capability patterns, to allow one to design patterns that are free of vulnerabilities and provably enforce their security properties. The thesis contents are organised as follows, mostly in accordance with the kinds of security property investigated.

In Chapter 2, we introduce and explain the main concepts that underpin the work in this thesis. We provide a brief overview of CSP, and discuss the object-capability model and the systems that embody it. Then we describe our basic approach to modelling object-capability systems in CSP and using FDR to reason about them. In particular, we describe how we model untrusted objects in CSP that can exhibit any and all behaviours permitted by the object-capability model, and how one can define an object-capability system modelled in CSP by using CSP's theory of refinement. We also discuss how to model single-threaded object-capability systems, such as object-capability languages like Joe-E and Cajita. We then describe how

CSP's theory of *data-independence* [Laz99] can be applied to allow us to prove that a property that is agnostic to data (such as one involving only capability propagation) holds for an object-capability system that is also agnostic to data, regardless of the specific data that exists within that system. This allows us, in the analyses performed in subsequent chapters, basically to ignore data when it is not relevant, confident that the results we obtain are unaffected by it.

In Chapter 3, we demonstrate how to reason about *safety properties* of object-capability patterns. We show how complicated safety properties, such as *safe coercion*, can be expressed naturally as CSP refinements and automatically checked using FDR. We demonstrate that this approach can automatically discover vulnerabilities in object-capability patterns that arise through concurrent and recursive invocation of relied-upon objects that are otherwise difficult to manually diagnose. To our knowledge, we demonstrate the first automatic detection of instances of each of these kinds of vulnerability in an object-capability pattern. This illustrates the extra expressiveness and utility of our approach over its predecessors.

In Chapter 4, we show how to generalise the results obtained from analysing a pattern deployed in a small fixed-sized system to allow us to conclude that the pattern upholds its security properties when deployed correctly in any arbitrary-sized system. We borrow Spiessens' idea of *aggregation* [Spi07] and couple this with Lazić's theory of *data-independence* [Laz99] for CSP. This allows us to treat small fixed-sized systems as *safe abstractions* of larger systems of arbitrary size, in which a pattern might be instantiated. Inspired by Spiessens [Spi07], we show how this approach can be used to reason about patterns that make use of unbounded object-creation by analysing such a pattern that implements a revocation policy that cannot be directly expressed in Scoll. We prove that this pattern upholds slightly different revocation properties when deployed in single-threaded and concurrent object-capability systems; this conclusion would be difficult to draw using previous approaches.

In Chapter 5, we show how to reason about the *information flow properties* of object-capability patterns. We discover that, in order to talk sensibly about the information flow properties of an object-capability pattern, one must make the necessary assumption that the only way for objects to directly influence each other is by sending and receiving messages. We show how traditional information flow properties can be modified to take this assumption into account, and how these modified properties can be expressed in terms of CSP refinement assertions to enable them to be automatically tested by FDR. To our knowledge we provide the first formal analysis of the information flow properties of an object-capability pattern that is able to detect covert channels present within that pattern. We also show how the approach developed in Chapter 4 can be extended to allow one to draw conclusions about the information flow properties of object-capability patterns

deployed in systems of arbitrary size with unbounded object creation.

In Chapter 6, we show how to reason about the *liveness properties* of object-capability patterns. Such properties cannot be expressed in previous formalisms like Scoll, which can reason only about *liveness possibilities* [Spi07]. We prove that liveness properties under necessary fairness assumptions cannot be directly expressed as CSP refinements in any CSP model that FDR might support, preventing FDR from being able to test them directly. Instead, we show how to analyse these properties by using FDR to test sufficient conditions for them. We demonstrate this by providing the first formal analysis of the liveness properties of an object-capability pattern. We find that this approach yields useful results for this pattern, but that it is unclear how well it would apply in general to other patterns. Also, the approach is not as straightforward to apply as one would like. We conclude that it would be worthwhile investigating whether other model-checking approaches (*e.g.* [Ros01, SLDW08, Liu09]) could be adapted to test liveness properties of object-capability patterns.

In Chapter 7, we investigate to what degree properties involving an object's *authority* can be expressed in CSP and automatically checked using FDR. An object's authority is the set of effects it can cause in a system by sending and receiving messages to and from other objects. We show how authority may be captured formally by presenting a framework for expressing *non-causation properties*, which we show can capture complex kinds of authority including delegable, non-delegable, revocable and single-use authority. We show, with the aid of Clarkson and Schneider's *hyperproperties* notion [CS10], that our framework can distinguish two primitive kinds of effect that can be caused, namely the *safety* and *liveness* effects respectively. This distinction then enables us to formalise the intuitive notions of *defensive correctness* and *defensive consistency* [Mil06] within our framework, which we show how to do. To our knowledge, this is the first time that such ideas have been formally captured. We show that non-causation of safety effects, and their opposites, can be judged using FDR for deterministic systems. However, we prove that in general non-causation properties (tested over all refinements) of nondeterministic systems cannot be directly expressed as refinement checks for FDR to carry out. We conclude that alternative model-checking approaches, besides those based on refinement checking, should be investigated to enable their automatic verification.

Finally, in Chapter 8, we conclude and consider avenues for future work. Appendix A contains some subsidiary results that are used at various points in this thesis. In this way, this thesis provides a comprehensive study on the utility and limits of refinement checking for analysing the security properties of object-capability patterns.

2 Preliminaries

In this chapter, we introduce and discuss the main concepts and ideas that underpin the work in later chapters of this thesis.

2.1 CSP

In this section we give a brief overview of the parts of CSP used in this thesis[1]. Further details may be found in [Ros97].

CSP is a process algebra for describing and reasoning about concurrent systems. CSP's syntax describes systems of concurrently executing *processes* that perform atomic *events* drawn from the set Σ; these processes communicate with each other by synchronising on the performance of common events. In this thesis, we restrict our attention to a fragment of CSP in which the set Σ of events is finite, which is necessary in order to allow systems to be checked with FDR.

CSP has a number of semantic models that give meaning to it and allow one to reason formally about systems described in CSP. We begin by discussing CSP's syntax before describing some notation and the main semantic models that we use in this thesis. We then explain how one formally analyses a system described in CSP using its automatic refinement checker, FDR.

2.1.1 Syntax

CSP has a rich syntax for describing processes. The primitive process $STOP$ can perform no activity and represents deadlock. For an event $a \in \Sigma$, the process $a \rightarrow P$ can perform the event a and then behave like the process P. For a set of events $A \subseteq \Sigma$, the process $?a : A \rightarrow P_a$ is initially willing to perform all of the events from the set A and offers its environment the choice of which should be performed. Once a particular event, $x \in A$, has been performed, it behaves like the process P_a with the identifier a bound to the value x that was chosen.

CSP allows multi-part events to be defined, where a dot is used to separate each part of an event. The "?" and "!" operators are then used to offer

[1] The fragment of CSP used in this thesis excludes termination.

specific sets of events by pattern-matching on the structure of multi-part events. Suppose we define the set of events $\{\mathsf{plot}.x.y \mid x, y \in \mathbb{N}\}$. Then the process $\mathsf{plot}?x?y \rightarrow STOP$ offers its environment all of the events from the set $\{\mathsf{plot}.x.y \mid x, y \in \mathbb{N}\}$, whilst the process $\mathsf{plot}?x : \{1, \ldots, 5\}!3 \rightarrow STOP$ offers all events from $\{\mathsf{plot}.x.3 \mid x \in \{1, \ldots, 5\}\}$.

The process $P \,\Box\, Q$ can behave like either the process P or the process Q and offers its environment the initial events of both processes, giving the environment the choice as to which process it behaves like. If the environment chooses an initial event of P, $P \,\Box\, Q$ goes on to behave like P, and similarly for Q. The process $P \sqcap Q$ can also behave like either P or Q but doesn't allow the environment to choose which; instead, it makes this choice internally. It therefore nondeterministically offers its environment the initial events of either P or Q (but not both). The "$\sqcap$" operator is often used, therefore, to model nondeterminism.

The "$" operator can also be used to model nondeterminism and, like the "?" operator, is used to pattern-match on multi-part events. For instance, the process $\mathsf{plot}\$x : \{1, \ldots, 5\}!3 \rightarrow STOP$ nondeterministically chooses the value x from the set $\{1, \ldots, 5\}$. Having done so, it is willing to perform the single event $\mathsf{plot}.x.3$.

The process $P \triangleright Q$ may behave like either P or Q. It can refuse to behave like P but cannot refuse to behave like Q. The "$\triangleright$" operator is sometimes called the *timeout* operator because $P \triangleright Q$ may be thought of as an abstract model of a process that can initially offer its environment the initial events of P. However, it makes this offer only for a limited time, after which it offers its environment only the initial events of Q.

$P \setminus A$ denotes the process obtained when P is run but all occurrences of the events in A are hidden from its environment. The environment cannot observe the occurrence of these events and is, therefore, prevented from synchronising on them. The hiding operator is often used to abstract away from the occurrence of irrelevant events.

The process **if** b **then** P **else** Q behaves like P if b is true and like Q otherwise. The notation $b \,\&\, P$ is shorthand for **if** b **then** P **else** $STOP$. The process **let** $x = y$ **within** P behaves like the process P with the identifier x bound to the value y.

The process $P[\![y_1, \ldots, y_n / x_1, \ldots, x_n]\!]$ behaves like the process P except that, for all $i \in \{1, \ldots, n\}$, it performs the event y_i whenever P performs the event x_i. The process $c.P$ behaves like the process P except that, for each event $x \in \Sigma$, $c.P$ performs the event $c.x$ whenever P performs x. Both of these operators are used, therefore, to rename certain events.

The process $P \underset{A}{\parallel} Q$ runs the processes P and Q in parallel forcing them to synchronise on all events from the set A. The process $S = \left\|_{i \in \{1,\ldots,n\}} (P_i, A_i)\right.$ is the *alphabetised parallel composition* of the n processes $P_1, \ldots, P_n$ on their corresponding *alphabets* $A_1, \ldots, A_n$. Each process P_i may perform events

only from its alphabet A_i, and each event must be synchronised on by all processes in whose alphabet it appears. The syntax $P_1 \ {}_{A_1}\|_{A_2} \ P_2$ is equivalent to $\|_{i \in \{1,\ldots,2\}}(P_i, A_i)$.

The process $P \ ||| \ Q$ runs P and Q in parallel with no synchronisation (and, hence, no communication) between the two. We say that P and Q are *interleaved* here.

A process *diverges* when it performs an infinite amount of internal activity without performing a visible event from Σ. If we take the process $P = a \rightarrow P$ that continually performs the event a and then we internalise the occurrence of a using the hiding operator, arriving at the process $P \backslash \{a\}$, we see that $P \backslash \{a\}$ diverges immediately because each occurrence of a results in $P \setminus \{a\}$ performing some internal activity and P can perform an infinite number of as. The primitive process **div** diverges immediately and, for our purposes, is equivalent to the process $P \setminus \{a\}$ above. A process that never diverges is said to be *divergence-free*.

The process $CHAOS_A$ is the most nondeterministic, divergence-free process that performs events from the non-empty set A. This process may be defined as follows[2].

$$CHAOS_A = \$a : A \rightarrow CHAOS_A \sqcap STOP.$$

If s is a sequence of events from Σ and P is a process that can perform the sequence of events s, then $P \ / \ s$ denotes the process whose behaviour is exactly the behaviour that P exhibits after P performs the sequence of events s.

2.1.2 Notation

We use the following notation. Sequences are written between angle-brackets, so the sequence that contains the first 3 natural numbers is written $\langle 0, 1, 2 \rangle$. Let s and t be sequences. Then $s \mathbin{\hat{}} t$ denotes the sequence obtained by concatenating s and t. We write $s \leq t$ to mean that s is *prefix* of t, *i.e.* $s \leq t \Leftrightarrow \exists u \bullet s \mathbin{\hat{}} u = t$. We write $s < t$ when s is a strict prefix of t, *i.e.* when $\exists u \bullet s \mathbin{\hat{}} u = t \wedge u \neq \langle\rangle$. If s is a sequence, then $\#(s)$ denotes the length of s.

Within the context of multi-part events, the notation $\{|c_1.c_2.\ldots.c_k|\}$ denotes the set of events whose first k components are respectively $c_1, c_2 \ldots, c_k$. So $\{|\mathsf{plot}|\} = \{\mathsf{plot}.x.y \mid x, y \in \mathbb{N}\}$ and $\{|\mathsf{plot}.1|\} = \{\mathsf{plot}.1.y \mid y \in \mathbb{N}\}$ using the example from the previous subsection.

If X is a set, then $\mathbf{P} X$ denotes the powerset of X. X^* denotes the set containing all finite sequences whose elements are drawn from the set X. We use "$-$" to denote set difference, so that if X and Y are sets, then $X - Y$ denotes the set obtained by removing all Y-elements from X.

[2]Note that this definition is sufficient for the standard denotational models of CSP used most commonly in this thesis. However, for more intricate models (such as the $\mathcal{FL}$ model discussed in Chapter 6) a more subtle definition is required.

2.1.3 Semantics

CSP has a number of semantic models. We describe the three most often used denotational semantic models, each of which will be used in this thesis. Others will also be used, but will be introduced later as necessary.

In any denotational semantic model $\mathcal{M}$, one process P is said to be *refined by* another Q, when every behaviour that $\mathcal{M}$ records as being able to be performed by Q, $\mathcal{M}$ also records as being able to be performed by P. In this case, we write $P \sqsubseteq_{\mathcal{M}} Q$ and say that Q *refines* P in $\mathcal{M}$. Intuitively, Q therefore refines P when Q is consistent with P while being less ambiguous, because there are fewer possibilities for how it might behave. When P is refined by Q, we sometimes call P an *anti-refinement* of Q.

Given two finite-state processes P and Q, FDR is used to automatically test whether the refinement $P \sqsubseteq_{\mathcal{M}} Q$ holds, when $\mathcal{M}$ is one of CSP's standard denotational semantic models. As explained later, by framing the security properties of a system in terms of such refinement statements, we can use FDR to check them automatically.

The Traces Model

The *traces model* is the simplest of CSP's denotational semantic models and gives meaning to a process by recording the finite sequences of events from Σ that it may perform. We write $traces(P)$ for the set of all finite sequences of visible events that can be performed by the process P. $traces(P)$ is P's representation in the traces model.

For any process P, $traces(P)$ always contains the empty sequence $\langle\rangle$ and is always *prefix-closed*, meaning that and if s is a member of $traces(P)$ then so is every trace t for which $t \leq s$.

We say that Q is a *traces-refinement* of P when Q refines P in the traces model. This occurs precisely when $traces(Q) \subseteq traces(P)$. In this case, we write $P \sqsubseteq_T Q$. When P's representation $traces(P)$ in the traces model is identical to that of Q, $traces(Q)$, we say that P and Q are *trace-equivalent* and write $P \equiv_T Q$, so that $P \equiv_T Q \Leftrightarrow traces(P) = traces(Q)$.

The traces model does not record enough information for one to reason about nondeterminism. This can be seen by noting that $traces(a \rightarrow STOP \,\Box\, b \rightarrow STOP) = traces(a \rightarrow STOP \sqcap b \rightarrow STOP) = \{\langle\rangle, \langle a\rangle, \langle b\rangle\}$. The first of these processes in deterministic while the second is nondeterministic.

The Stable-Failures Model

The *stable-failures model* is more expressive than the traces model and can be used to reason about nondeterminism. In the stable-failures model, a process P is represented by the two sets $traces(P)$ and $failures(P)$. So the stable-failures model records at least as much information about a process

as the traces model. $\mathit{failures}(P)$ is the set of P's *stable-failures*. Each stable-failure is a pair (s, X), and represents that P can perform the sequence of events $s \in \mathit{traces}(P)$ and then reach a *stable* state from which no internal activity can occur and none of the events in X can be performed. When $(s, X) \in \mathit{failures}(P)$, we say that P can *stably refuse* X after performing the trace s.

For a process P, let $T = \mathit{traces}(P)$ and $F = \mathit{failures}(P)$, so that T and F capture P's representation in the stable-failures model. Then T and F satisfy the following *healthiness conditions*, which we present in the form of axioms of the stable-failures model [Ros97].

F1. T contains the empty sequence $\langle\rangle$ and is prefix-closed.

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

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

Axiom **F2** arises because whenever P can stably refuse to perform some set X of events, P can also stably refuse to perform any subset of events Y of X. Axiom **F3** arises because if P cannot perform some event a directly after the trace s, then P must be able to stably refuse a whenever it stabilises (*i.e.* reaches a stable state) directly after s.

Note that because the process **div** never stabilises, $\mathit{failures}(\mathbf{div}) = \{\}$. This highlights the fact that the stable-failures model records only stable refusal information, avoiding recording information about events that may be refused from unstable states (*i.e.* those from which internal activity can occur). Also, when P is divergence-free, $\mathit{traces}(P)$ may be calculated from $\mathit{failures}(P)$ as $\mathit{traces}(P) = \{s \mid (s, X) \in \mathit{failures}(P)\}$. Hence, $\mathit{traces}(P)$ adds extra information above what is recorded in $\mathit{failures}(P)$ only when P is not divergence-free.

Letting $F = \mathit{failures}(a \rightarrow \mathit{STOP} \sqcap b \rightarrow \mathit{STOP})$, $(\langle\rangle, \{a\}) \in F$ since if the internal choice in this process is resolved to the right, so that it is willing initially to perform only b, then it refuses a initially. Similarly, $(\langle\rangle, \{b\}) \in F$ too. However, $(\langle\rangle, \{a, b\}) \notin F$ because this process must initially be able to perform either a or b, so it cannot initially refuse to perform both events.

Neither $(\langle\rangle, \{a\})$ nor $(\langle\rangle, \{b\})$ are present in $\mathit{failures}(a \rightarrow \mathit{STOP} \mathbin{\Box} b \rightarrow \mathit{STOP})$, however, since this process can refuse neither a nor b initially. Hence, the nondeterminism present in the process mentioned in the previous paragraph, that is absent in this one, is clearly captured in the previous one's stable-failures semantics (*i.e.* in F above) by the presence of each of these two stable-failures.

Indeed, the presence of nondeterminism in a divergence-free process is defined in terms of its stable-failures semantics. A process acts nondeterministically if at some point in time it can both perform and refuse some event. This is captured by the following definition of *determinism* (*i.e.* the absence of nondeterminism).

Definition 2.1.1 (Determinism)**.** A CSP process P is *deterministic*, written $det(P)$, iff it is divergence-free and

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

We say that some process Q is a stable-failures refinement of another P, iff $traces(Q) \subseteq traces(P) \wedge failures(Q) \subseteq failures(P)$. In this case we write $P \sqsubseteq_F Q$. We have, for instance, that $a \rightarrow STOP \sqcap b \rightarrow STOP \sqsubseteq_F a \rightarrow STOP \,\Box\, b \rightarrow STOP$.

We say that P and Q are *stable-failures equivalent* when $traces(P) = traces(Q) \wedge failures(P) = failures(Q)$. In this case, we write $P \equiv_F Q$.

The Failures-Divergences Model

The *failures-divergences* model is also more expressive than the traces model and can reason about not only nondeterminism but also divergence, which the stable-failures model cannot. It is the most commonly used denotational semantic model for CSP.

This model is *divergence-strict*, meaning that it records information about divergence but treats divergence as catastrophic by effectively assuming that once a process can diverge, it can do anything at all. In this model, a process P is represented by the sets $failures_\perp(P)$ and $divergences_\perp(P)$. $failures_\perp(P)$ and $divergences_\perp(P)$ are each obtained from the sets $failures(P)$ and $divergences(P)$ respectively in such a way that reflects the divergence-strictness of the failures-divergences model. $divergences(P)$ contains those traces $s \in traces(P)$ where directly after performing s, P can perform an infinite amount of internal activity and so diverge.

$failures_\perp(P)$ and $divergences_\perp(P)$ are each obtained from the underlying sets by applying a transformation that encodes the divergence-strictness assumption that once a process diverges it can do anything at all. For this reason, $divergences_\perp(P)$ includes every extension t of every trace $s \in divergences(P)$ and $failures_\perp(P)$ includes any failure that might be exhibited following any trace $s \in divergences_\perp(P)$.

$$divergences_\perp(P) = divergences(P) \cup \{s \mid \exists\, t \in divergences(P) \wedge t \leq s\}$$
$$failures_\perp(P) = failures(P) \cup \{(s, X) \mid s \in divergences_\perp(P) \wedge X \subseteq \Sigma\}$$

Note that when $divergences(P) = \{\}$ (*i.e.* when P is divergence-free), $failures_\perp(P) = failures(P)$. In this case, the failures-divergences model records exactly the same information about P as is recorded by the stable-failures model.

For any process P, let $F = failures_\perp(P)$, $D = divergences_\perp(P)$ and $T = \{t \mid (t, X) \in F\}$. Then T and F satisfy Axioms **F1**–**F3** above. The following two axioms are also satisfied by F and D and encode the divergence-strictness of this model.

D1. $s \in D \wedge t \in \Sigma^* \Rightarrow s \hat{} t \in D$.

D2. $s \in D \wedge X \subseteq \Sigma \Rightarrow (s, X) \in F$.

We say that one process Q is a *failures-divergences refinement* of another P when $\mathit{failures}_\perp(Q) \subseteq \mathit{failures}_\perp(P) \wedge \mathit{divergences}_\perp(Q) \subseteq \mathit{divergences}_\perp(P)$. In this case we write $P \sqsubseteq_{FD} Q$. Sometimes we will write simply $P \sqsubseteq Q$, which should be interpreted to mean failures-divergences refinement unless otherwise stated. We say that P and Q are *failures-divergences equivalent*, written $P \equiv_{FD} Q$, iff $\mathit{failures}_\perp(P) = \mathit{failures}_\perp(Q) \wedge \mathit{divergences}_\perp(P) = \mathit{divergences}_\perp(Q)$.

Observe that in the failures-divergences model, a divergence-free process is refined only by other divergence-free processes[3]. Also, **div** is failures-divergences refined by every other process, meaning that it is the least refined process in the failures-divergences model.

For most of this thesis, we restrict our attention to divergence-free processes; we often use the failures-divergences model to specify refinement assertions that include the requirement that a divergence-free process cannot be refined by one that can diverge.

2.1.4 Verifying Properties of CSP Processes

Having described CSP and its semantic models, we now describe the basic technique of how to use FDR to prove that a system modelled in CSP satisfies a particular property. This is done by expressing the property in terms of a CSP refinement statement that can be automatically checked using FDR.

Naturally, any system that we want to analyse will be represented by some CSP process *Sys*. Usually, we begin by formally stating the property that we wish *Sys* to satisfy. This property is most often stated in terms of *Sys*'s representation in one of the three denotational semantic models mentioned above, *i.e.* the traces, stable-failures and failures-divergences models.

A common example of a property that one might want to test is that none of the events from a certain set A can occur in *Sys*[4]. Let $\mathit{cannotOccur}_A(\mathit{Sys})$ denote this property which, for an arbitrary process P, is most easily expressed in the traces model, as follows.

$$\mathit{cannotOccur}_A(P) = \nexists\, s, a \bullet s \hat{} \langle a \rangle \in \mathit{traces}(P) \wedge a \in A. \tag{2.1}$$

To use FDR to decide whether $\mathit{cannotOccur}_A(\mathit{Sys})$, we express the property $\mathit{cannotOccur}_A(P)$ in terms of a CSP refinement statement that mentions P, and then test this refinement in FDR with *Sys* in place of P.

[3]This is not true for the stable-failures model, however, since $P \sqsubseteq_F P \sqcap \mathbf{div}$ for any process P because the stable-failures model doesn't record information about divergence.

[4]This is actually a very simple *safety property*. We will see more complex ones later in Chapter 3.

Consider the most general process that never performs any event from A. This process is simply $CHAOS_{\Sigma-A}$. Its traces include all traces except those that contain some A-event. Hence, for some process P, $cannotOccur_A(P)$ holds precisely when P performs no trace s that cannot be performed by $CHAOS_{\Sigma-A}$, *i.e.* precisely when $traces(P) \subseteq traces(CHAOS_{\Sigma-A})$. Hence, we have that

$$cannotOccur_A(P) \Leftrightarrow CHAOS_{\Sigma-A} \sqsubseteq_T P \tag{2.2}$$

We can use FDR to test that *Sys* can perform no A-events, then, by having FDR test whether $CHAOS_{\Sigma-A} \sqsubseteq_T Sys$.

Refinement-Closed Properties

Note that this refinement check, $CHAOS_{\Sigma-A} \sqsubseteq_T P$, for the property $cannotOccur_A(P)$ is of the form $Spec \sqsubseteq_{\mathcal{M}} G(P)$ where *Spec* is some fixed *specification* process ($CHAOS_{\Sigma-A}$), $G(\_)$ is a CSP expression involving P (the identity expression) and $\mathcal{M}$ is a CSP model (the traces model).

Consider an arbitrary property $Prop(P)$ expressed in terms of P's representation in some denotational semantic model $\mathcal{M}$. It is straightforward to show that when $Prop(P)$ can be expressed as a refinement check of the form $Spec \sqsubseteq_{\mathcal{M}} G(P)$, that *Prop* must be *refinement-closed* in $\mathcal{M}$[5]. A property is refinement-closed in a semantic model $\mathcal{M}$ precisely when, if it holds for any process P, it must also hold for all of P's refinements in $\mathcal{M}$.

Definition 2.1.2 (Refinement-Closed)**.** The property *Prop* is refinement-closed in the denotational model $\mathcal{M}$ iff

$$\forall P \bullet Prop(P) \Rightarrow \forall Q \bullet P \sqsubseteq_{\mathcal{M}} Q \Rightarrow Prop(Q).$$

One may observe that the property $cannotOccur_A(P)$ is refinement-closed in the traces model since if some process P has no trace $s^\frown\langle a\rangle$ (where $a \in A$), then neither can any of P's traces refinements Q since for each $traces(Q) \subseteq traces(P)$ by definition.

Most useful security properties are refinement-closed in the model in which they are expressed, including safety and liveness properties, which are examined later in Chapters 3 and 6 respectively. An exception to this are some information flow properties [Low07]; we examine information flow properties in Chapter 5.

2.2 The Object-Capability Model

Before describing how we model object-capability systems in CSP, we first describe the object-capability model, and the kinds of systems that embody

[5]This follows because each CSP operator, and hence the context $G(\_)$, is monotonic under $\sqsubseteq_{\mathcal{M}}$ [Ros08].

it, in more detail. This allows us to better understand what needs to be included in our CSP models of object-capability systems.

The *object-capability model* [Mil06] is a model of computation and security that aims to capture the semantics of many actual object-based programming languages and capability-based systems, including all of those listed later in Table 2.1. An *object-capability system* is an instance of the model and comprises just a collection of *objects*, connected to each other by *capabilities*. An object is a protected entity comprising code and mutable state that together define its behaviour. An object's state includes both data and the capabilities it possesses. A capability c is an unforgeable object reference that allows its holder to send messages to the object it references by *invoking* c. In any object-capability system, capabilities and data are primitively distinct, meaning that each can always be distinguished from the other[6].

In an object-capability system, the only overt means for objects to interact is by sending messages to each other. Capabilities may be passed between objects only within messages. In practice, object o can pass one of its capabilities c directly to object p only by invoking a capability it possesses that refers to p, including c in the invocation. This implies that capabilities can be passed only between objects that are connected, perhaps via intermediate objects.

Each object may expose a number of interfaces, known as *facets*. A capability that refers to an object o also identifies a particular facet of o to which messages sent on that capability are directed. This allows the object to expose different functionality to different clients by handing each of them a capability that identifies a separate facet, for example.

An object may also create others. In doing so, it must supply any resources required by the newly created object, including its code and any data and capabilities it is to possess initially. Hence, a newly created object receives its first capabilities solely from its parent. When creating an object, the parent exclusively receives a capability to the child. Thus, an object's parent has complete control over those objects the child may come to interact overtly with in its lifetime. This is the basis upon which mandatory security policies can be enforced [Mil06].

2.2.1 Current Object-Capability Systems

The object-capability model is purposefully very general, and leaves room for different object-capability systems that implement it to vary in a number of different ways.

As hinted at previously, there are two main kinds of object-capability systems, namely operating systems and programming languages. In an

[6]Because of this, traditional *password capability systems* [APW86], in which capabilities are just data strings, are not object-capability systems.

object-capability operating system, each operating system process may be thought of as a separate object and inter-process communication occurs by passing messages on capabilities that refer to individual processes. In an object-capability language, objects are akin to those from object-oriented languages; capabilities are simply object references and sending a message to an object is achieved by calling one of its methods.

Examples of object-capability operating systems that are currently in use or under development include CapROS [Lan09] (which is derived from EROS [SSF99], a reimplementation of KeyKOS [Har85]), Coyotos [SDN+04, SA07], NICTA's seL4 [EKE08, DEE08] and the Annex Capability Kernel [GMO+07]. Plash [Sea07] and Capsicum [Wat09] both implement object-capability environments atop POSIX operating systems and, as such, we consider them to be virtualised object-capability operating systems. Current object-capability languages include E [Mil06], Cajita [MSL+08] (which is part of Google's Caja project), Joe-E [MWC10, MW08], Emily [Sti07], Tamed Pict [Košo8] and Class [DY08].

In an object-capability operating system, each object (*i.e.* each operating system process) usually executes concurrently to all others. In contrast, most object-capability languages are *single-threaded*, meaning that a single-thread of control is shared between all objects in any such system; as one object invokes another, the thread of control migrates from the invoker to the invokee, and then returns to the invoker when the invokee returns from the invocation. Hence, different object-capability systems can have vastly different concurrency semantics.

Table 2.1 depicts a taxonomy of current object-capability systems, categorising them across the following dimensions. A system is *concurrent* if it is not single-threaded. A system has *inter-thread blocking sends* if it is concurrent and allows one object to perform a blocking send of a message to another object that doesn't share the same thread of control. Such sending will block the sender until the recipient is ready to receive the message. A system may exhibit the following forms of *reentrancy*: *concurrent*, *recursive*, neither or both. A system exhibits concurrent reentrancy if it is concurrent and a single object can receive an invocation from one thread whilst in the middle of processing a previous invocation from some other thread. A system exhibits recursive reentrancy if a single object can be executed recursively by a single thread. A system may provide *non-blocking communications* for *sending*, *receiving* or both. It is not required for a sender to be notified when a non-blocking send completes and in many systems no such notification is provided. A system provides the equality primitive *EQ* if it provides a facility to determine whether two arbitrary capabilities refer to exactly the same object, even if this facility is not universally available.

Table 2.1 illustrates the sheer diversity of current object-capability systems. When modelling object-capability patterns, we therefore require a formalism that is flexible enough to be able to cope with this diversity, by

System	Kind	Concurrent	Inter-Thread Blocking Sends	Reentrancy	Non-Blocking Comms.	EQ
E	Language	Yes	No	Recursive	Send + Recv.	Yes
Cajita	Language	No	N/A	Recursive	Send + Recv.[8]	Yes
Joe-E	Language	No	N/A	Recursive	Send + Recv.[8]	Yes
Emily	Language	No	N/A	Recursive	No	Yes
Class	Language	No	N/A	Recursive	No	Yes
Tamed Pict	Language	Yes	No	R.+Concurrent	Send + Recv.	No
EROS / CapROS	OS	Yes	Yes	No	No	Yes
Coyotos	OS	Yes	Yes	No	Send[9]	Yes
seL4	OS	Yes	Yes	No	Send[9]	No
Annex	OS	Yes	Yes	R.+Conc.[10]	Send	No
Plash	Virt. OS	Yes	Yes	R.+Conc.[10]	Send + Recv.	Yes
Capsicum	Virt. OS	Yes	Yes	R.+Conc.[10]	Send + Recv.	Yes

Table 2.1: A taxonomy of current object-capability systems.[7]

[7] Bill Frantz, David-Sarah Hopwood, Matej Košík, Charles Landau, Alex Murray, Mark Seaborn and David Wagner all provided input to this table.

[8] The `ref_send` library [Clo08] provides non-blocking communication in Joe-E and Cajita.

[9] This facility is provided in seL4 and Coyotos through their *non-blocking Send* operations which are best-effort sending primitives that don't notify the sender if sending fails for some reason [DEE08, SA07].

[10] Annex, Plash and Capsicum have similar reentrancy. In each, a separate thread is allocated at minimum to each object. While waiting for a reply message, an object's thread is free to process other incoming messages. This creates the possibility for both concurrent and recursive reentrant invocation.

being able to express each of the different phenomena captured in Table 2.1. Fortunately, CSP is able to express them all.

2.3 Modelling Object-Capability Systems in CSP

In this section, we describe our approach to modelling object-capability systems in CSP. Note that we ignore the issue of object creation for now. This will be handled later on in Chapter 4.

2.3.1 System Model

We model an object-capability system *System* that comprises a set *Object* of objects as the alphabetised parallel composition of a set of processes $\{behaviour(o) \mid o \in Object\}$, one for each object $o \in Object$, on their corresponding alphabets $\{\alpha(o) \mid o \in Object\}$. So $System = \|_{o \in Object}(behaviour(o), \alpha(o))$. The behaviour of each object $o \in Object$ is captured by the process $behaviour(o)$, which may perform events from the object's alphabet $\alpha(o)$.

The facets of each object $o \in Object$ are denoted $facets(o)$. We restrict our attention to those well-formed systems in which $facets(o) \cap facets(o') \neq \{\} \Rightarrow o = o'$. Recall that an individual capability refers to a particular facet of a particular object. Hence, we define the set $Capability = \bigcup_{o \in Object} facets(o)$ containing all entities to which capabilities may refer.

The events that each process $behaviour(o)$ can perform represent it sending and receiving messages to and from objects in the system. We define events of the form $f_1.f_2.op.arg$ to denote the sending of a message from the object with facet f_1 to facet f_2 of the object with this facet, requesting it to perform operation op, passing the argument arg and a *reply* capability f_1, which can be used later to send back a response. Here $f_1, f_2 \in Capability$. Arguments are either capabilities, data or the special value null, so $arg \in Capability \cup Data \cup \{\mathsf{null}\}$, for some set $Data$ of data. An operation op comes from the set $\{\mathsf{Call}, \mathsf{Return}\}$. These operations model a call/response remote procedure call sequence in an object-capability operating system or a method call/return in an object-capability language.

The alphabet of each object $o \in Object$ contains just those events involving o. These include those events that represent o sending a message (*i.e.* those events $f_1.f_2.op.arg$ for which $f_1 \in facets(o)$), and those that represent o receiving a message (*i.e.* those for which $f_2 \in facets(o)$). Hence

$$\alpha(o) = \{|f_1.f_2 \mid f_1, f_2 \in Capability \wedge (f_1 \in facets(o) \vee f_2 \in facets(o))|\}.$$

We require that the process $behaviour(o)$ representing the behaviour of each object $o \in Object$ adheres to the basic rules of the object-capability model, such as not being able to use a capability it has not legitimately

acquired. We codify this by defining the most general process that includes only those behaviours that can be legitimately exhibited by an object o under the rules of the object-capability model. Letting $facets = facets(o)$ denote the set that comprises o's facets, and $caps \subseteq Capability$ and $data \subseteq Data$ denote the sets of capabilities and data that o initially possesses, the most general process that includes just those behaviours permitted by the object-capability model that o may perform is denoted $Untrusted_{OS}(facets, caps, data)$ and appears in Snippet 2.1.[11]

$$
\begin{aligned}
&Untrusted_{OS}(facets, caps, data) = \\
&\left(\begin{array}{l}
?me : facets?c : caps \cup facets?op?arg : caps \cup data \cup \{\mathsf{null}\} \rightarrow \\
\quad Untrusted_{OS}(facets, caps, data) \ \Box \\
?from : Capability - facets?me : facets?op?arg \rightarrow \\
\quad \textbf{let}\ C' = \{arg, from\} \cap Capability\ ;\ D' = \{arg\} \cap Data\ \textbf{within} \\
\quad Untrusted_{OS}(facets, caps \cup C', data \cup D')
\end{array}\right) \\
&\sqcap\ STOP.
\end{aligned}
$$

Snippet 2.1: The most general object in an object-capability system.

This process represents that an arbitrary object that exposes the set of facets *facets*, and whose capabilities and data are *caps* and *data* respectively, may invoke any capability $c \in caps \cup facets$ that it possesses, requesting any operation *op*, and including in that invocation any argument $arg \in caps \cup data \cup \{\mathsf{null}\}$ it has, along with a reply capability $me \in facets$ to one of its facets. Such an object may also receive any invocation from any other, such that the reply capability included in the invocation is $from \in Capability - facets$, to one of its facets $me \in facets$, requesting an arbitrary operation *op*, and containing an arbitrary argument *arg*. If such an invocation occurs, the object may acquire the reply capability *from*, as well as any capability or datum *arg* contained as an argument.

The "$\sqcap\ STOP$" clause allows this process to be able to deadlock at any time and ensures that it is maximally nondeterministic (and therefore as general as possible) in the stable-failures model. With this clause, this process is failures equivalent to one where each "?" symbol is replaced by a nondeterministic choice involving "$"[12].

The behaviour $behaviour(o)$ of an object $o \in Object$, whose initial capabilities and data are $caps(o)$ and $data(o)$ respectively, is then valid if and only if all behaviours it contains are present in $Untrusted_{OS}(facets(o), caps(o), data(o))$. This leads to the following definition of a valid object-capability system.

[11] The purpose of the "*OS*" subscript in this name is explained later in Section 2.3.5.

[12] We do not use "$" here (although doing so might be more intuitive) because in any use "$a : A$" the set A must be non-empty.

Definition 2.3.1 (Object-Capability System)**.** An *object-capability system* is a tuple $(Object, behaviour, facets, Data)$, where:

- $Object$ is the finite set that contains the unique names of the objects that comprise the system;
- $Data$ is the finite set of data that may exist in the system;
- $facets$ is a function that gives the finite set of names for the facets that each object $o \in Object$ exposes, such that

$$\forall\, o, o' \in Object \bullet facets(o) \cap facets(o') \neq \{\} \Rightarrow o = o';$$

and

- $behaviour$ is a function that maps object names to CSP processes, giving the behaviour of each object in the system such that, letting $Capability = \bigcup_{o \in Object} facets(o)$, there exist functions $caps : Object \to \mathbf{P}\, Capability$ and $data : Object \to \mathbf{P}\, Data$ that assign minimal initial capabilities and data to each object so that, for each $o \in Object$,

$$Untrusted_{OS}(facets(o), caps(o), data(o)) \sqsubseteq_{FD} behaviour(o).$$

In an object-capability system $(Object, behaviour, facets, Data)$, the *alphabet* of each object $o \in Object$ is denoted $\alpha(o)$ and defined as

$$\alpha(o) = \{|\, f_1.f_2 \mid f_1, f_2 \in Capability \wedge (f_1 \in facets(o) \vee f_2 \in facets(o))\,|\}.$$

An object-capability system $(Object, behaviour, facets, Data)$ is captured by the CSP process $System$ defined as[13]

$$System = \Big\|_{o \in Object} (behaviour(o), \alpha(o)).$$

Sometimes, when no ambiguity is created by doing so, we will identify an object-capability system $(Object, behaviour, facets, Data)$ by the CSP process $System = \Big\|_{o \in Object} (behaviour(o), \alpha(o))$ that captures it.

Some Remarks on the System Model

Before continuing, it is worth making a few remarks about our system model for object-capability systems, embodied in Definition 2.3.1.

Firstly, our model of an object-capability system differs from previous ones (such as Spiessens' [Spi07]) in that it explicitly includes the notion of

[13] Note that for some of the analyses performed in this thesis, when building a system $System$, we sometimes apply FDR's `normalise` *compression* function [RGG+95] to each $behaviour(o)$ before composing them in parallel to make the resulting $System$ quicker for FDR to analyse.

facets, via the *facets* function. We made this choice because many object-capability systems, like seL4, Coyotos and the Annex kernel, primitively support facets in one form or another (usually in the form of unmodifiable meta-information attached to each capability that identifies a particular facet of the target object). In other systems, like E and Joe-E for instance, facets must be implemented manually. We choose to supports facets primitively, despite them not being primitively available in all object-capability systems, to allow us to more easily model object-capability systems like seL4 and Annex. When modelling systems without direct support for facets, one can simply give each object o just a single unique facet by, for instance, setting $facets(o) = \{o\}$.

Secondly, this model was designed to be the simplest model that is adequate to reason about the sorts of systems and properties covered in this thesis. While it is sufficient for our purposes, it is by no means as complete as one might otherwise like. For instance, our model of data is very simple and effectively treats all data as unguessable. Also, we allow only a single capability to be sent in each message. Each of these limitations could be overcome by appropriately extending the definitions above, although doing so might affect the complexity of model-checking the resulting systems. The results and basic techniques obtained and employed in this thesis should translate readily to any such extended model.

Finally, note from these definitions, because the process $Untrusted_{OS}$ is divergence-free, each $behaviour(o)$ must also be divergence-free (or else the failures-divergences refinement wouldn't hold). This means that we effectively choose to restrict our attention to those systems in which all objects are divergence-free. Doing so does not prevent us from modelling untrustworthy objects that might perform an infinite amount of internal computation that does not directly effect other objects in the system, since such behaviour can be modelled by an object that repeatedly sends messages to itself for instance. However, this restriction does make much of the technical analysis involving CSP's semantic models simpler because it implies that the process *System* that represents any such object-capability system will itself be divergence-free.

2.3.2 An Example System

To illustrate our approach to modelling object-capability systems in CSP, we now present a small example in which we model the object-capability system depicted in Figure 2.1.

This system contains just two objects, Alice and Bob. Neither initially possesses a capability to the other; although each possesses a capability to itself, as is natural for each object to do in an object-capability system[14]. The

[14]In future, we will avoid drawing these self-loops in pictures of object-capability systems to reduce clutter.

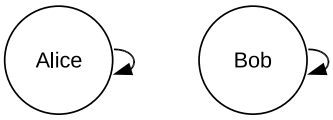

Figure 2.1: A very simple object-capability system.

behaviour of each object is unknown. We therefore model each as being completely untrusted, *i.e.* as instances of the most general process $Untrusted_{OS}$ from Snippet 2.1. We give each a capability to itself, and we allow each to initially possess all data in the system. This leads to the following definitions for the behaviour of each.

$$behaviour(\mathsf{Alice}) = Untrusted_{OS}(facets(\mathsf{Alice}), \{\mathsf{Alice}\}, Data),$$
$$behaviour(\mathsf{Bob}) = Untrusted_{OS}(facets(\mathsf{Bob}), \{\mathsf{Bob}\}, Data).$$

Let $Object = \{\mathsf{Alice}, \mathsf{Bob}\}$. Each of these objects exposes just a single facet, so we may set $facets(o) = \{o\}$ for all $o \in Object$. This object-capability system is then captured formally by the tuple $(Object, behaviour, facets, Data)$, where $Data$ denotes the set of all data in the system. For now, we let $Data$ be just a singleton set containing a fresh value, so that $Data = \{\mathsf{SomeDatum}\}$. $(Object, behaviour, facets, Data)$ is trivially an object-capability system under Definition 2.3.1 because, for any CSP process P, it is naturally the case that $P \sqsubseteq_{FD} P$. Per Definition 2.3.1, this object-capability system is captured by the CSP process $System = \|_{o \in System}(behaviour(o), \alpha(o))$.

We would expect that in this system, by the rules of the object-capability model, neither Alice nor Bob should ever be able to send a message to the other. Hence, we would expect that in $System$, no event from the set $A = \{|o.o' \mid o, o' \in \{\mathsf{Alice}, \mathsf{Bob}\} \wedge o \neq o'|\}$ should ever occur, *i.e.* that $cannotOccur_A(System)$ should hold, where $cannotOccur_A$ is the property defined by Equation 2.1. By Equation 2.2, we can test that this property holds by testing whether

$$CHAOS_{\Sigma - A} \sqsubseteq_T System.$$

Testing this assertion in FDR reveals that it holds, as one would expect[15].

2.3.3 Modelling Trusted Objects

The system in Figure 2.1 contains two objects that can exhibit any and all behaviours, given the capabilities that each object possesses. Both objects therefore naturally model completely untrusted objects, about whose

[15] Using the machinery presented in Chapter 4, one can generalise this result to show that in any object-capability system no two disjoint subgraphs of objects can ever become connected. We leave doing so as an exercise to the reader.

behaviour we can make no assumptions. Security-enforcing patterns are necessarily implemented by trusted objects, however. A trusted object is one whose behaviour we are prepared to make some assumptions about. To model a trusted object, we must therefore be able to faithfully model those aspects of its behaviour that we rely upon. These are precisely those aspects of its behaviour that are designed to enforce the security property implemented by the pattern that it instantiates.

A trusted object doesn't usually exhibit the full range of behaviours that can be exhibited by the process $Untrusted_{OS}$ from Snippet 2.1. Indeed, what one usually trusts about such an object is that it will not perform certain behaviours that it might otherwise be capable of performing. An example is a trusted object that possesses a capability but is trusted never to divulge that capability to any other object. Modelling a trusted object therefore involves defining a CSP process that exhibits the behaviours we expect the object to possibly perform, while not exhibiting the behaviours that we trust the object not to perform.

For instance, suppose we altered the system depicted in Figure 2.1 by giving Alice a capability to Bob and by changing Bob's behaviour so that he is trusted never to invoke any object (including itself), being willing only to be invoked by other objects. We model Bob's new (trusted) behaviour by defining the process $NeverInvokes(facets, caps, data)$ that represents an object whose facets are *facets* and that initially possesses the capabilities and data from the sets *caps* and *data* respectively, and is trusted to never invoke any object.

$$
\begin{aligned}
&NeverInvokes(facets, caps, data) = \\
&\left(\begin{array}{l}
?from : Capability - facets?me : facets?op?arg \rightarrow \\
\quad \textbf{let}\ C' = \{arg, from\} \cap Capability\ ;\ D' = \{arg\} \cap Data\ \textbf{within} \\
\quad NeverInvokes(facets, caps \cup C', data \cup D')
\end{array}\right) \\
&\sqcap STOP.
\end{aligned}
$$

Notice that to define this process, we have basically altered the definition of the process $Untrusted_{OS}$ from Snippet 2.1 by removing the clause that allows it to exhibit those behaviours that model an object calling itself or another object, *i.e.* by restricting the behaviours it may exhibit to exclude all those that we trust the object never to perform.

Note also that the restriction that *from*, the reply capability passed with the invocations of *NeverInvokes*, comes from the set $Capability - facets$ (*i.e.* that *from* is not one of the object's own facets) does *not* imply that any object that implements *NeverInvokes* must check who has invoked it and allow only those invocations to proceed that come from objects other than itself. Instead, it merely represents that an object that implements *NeverInvokes* must never be involved in invocations in which the reply capability in the invocation is one of its own facets, *i.e.* must never invoke any object including itself. This same point applies to every other object modelled in this thesis.

This new system would be instantiated by altering $behaviour(\mathsf{Alice})$ to give her a capability to Bob and by altering $behaviour(\mathsf{Bob})$ so that his behaviour is given by the process *NeverInvokes* rather than by $Untrusted_{OS}$, so that

$$behaviour(\mathsf{Alice}) = Untrusted_{OS}(facets(\mathsf{Alice}), \{\mathsf{Alice}, \mathsf{Bob}\}, Data),$$
$$behaviour(\mathsf{Bob}) = NeverInvokes(facets(\mathsf{Bob}), \{\mathsf{Bob}\}, Data).$$

While this system and the system in Figure 2.1 are each incredibly simple, our approach to modelling object-capability systems in CSP is very general. In particular, our approach can express each of the various phenomena captured in Table 2.1, thanks to CSP's high degree of expressiveness. We discuss two of these phenomena now, namely non-blocking communication and single-threaded vs. concurrent systems. Most of the others will be discussed later in this thesis, in the context of the patterns within which each becomes relevant. Recursive reentrancy, for instance, is discussed in Chapter 3, where we model trusted objects that can be recursively invoked.

2.3.4 Non-Blocking Communication

In any object-capability system $(Object, behaviour, facets, Data)$, captured by the process $System = \|_{o \in Object}(behaviour(o), \alpha(o))$, each event $f_1.f_2.op.arg$ that represents a communication between one object $o_1 \in Object$ and another $o_2 \in Object$ (where $o_1 \neq o_2 \wedge \forall i \in \{1,2\} \bullet f_i \in facets(o_i)$) will necessarily be present in just the alphabets $\alpha(o_1)$ and $\alpha(o_2)$ of o_1 and o_2 respectively. Because *System* is formed as an alphabetised parallel composition, every event that it performs must be synchronised on by all $behaviour(o)$ in whose alphabet the event appears. This means that $f_1.f_2.op.arg$ can occur only when both $behaviour(o_1)$ and $behaviour(o_2)$ are willing to perform it. A consequence is that even if the sender o_1 wants to perform this event (*i.e.* wants to send this message to facet f_2 of o_2), the communication cannot occur until o_2 is ready to perform this same event (*i.e.* is ready to receive this message). If this is the only event that o_1 is willing to perform, then o_1 will block until o_2 is ready to perform it with him.

As such, we say that all communication in our CSP models of object-capability systems is *blocking*. Some object-capability systems also include support for *non-blocking* object invocation (see Table 2.1). Non-blocking object invocations are implemented in actual object-capability systems atop blocking communication primitives either by buffering them and deferring their execution to a later time (as occurs in E, Joe-E and Cajita), by performing them in a separate thread (as occurs in Tamed Pict, Plash, Annex and Capsicum), or by discarding the invocation if the receiver is not ready to receive it (as occurs in seL4 and Coyotos).

Each of these approaches can be modelled in CSP atop our basic blocking model of communication. However, for simplicity, we mainly concern

ourselves with blocking invocation only in this thesis (meaning that, for the most part, the work in this thesis might need to be extended to fully cover those object-capability patterns in which trusted objects make use of non-blocking invocation). We do however briefly consider the third form of non-blocking invocation in Section 6.2 in the context of liveness.

2.3.5 Single-Threaded Systems

Observe from Table 2.1 that many current object-capability systems are single-threaded, meaning that in each of them all objects share a single thread of control. However, our technique for modelling object-capability systems, embodied in Definition 2.3.1, naturally gives each object $o \in \mathit{Object}$ its own thread of control by representing it as a distinct process $\mathit{behaviour}(o)$ that runs in parallel to all others.

It is important to be able to model single-threaded systems accurately, as certain object-capability patterns work correctly only when deployed in single-threaded environments [Mur08]. We show how this can be done by simply imposing extra restrictions on Definition 2.3.1.

An object is *active* iff it is currently executing and is *inactive* otherwise. In a single-threaded object-capability system, it is naturally the case that only one object is ever active at a time. The object that is currently active can invoke other objects. When it does so, the invokee becomes active and the invoker (who was the active object) becomes inactive. In a single-threaded object-capability system, we must therefore distinguish between when each object is active and inactive, and impose the restriction that only one object is ever active at a time.

We do so by refining the behaviour of the most general object, $\mathit{Untrusted}_{OS}$ from Snippet 2.1, into two processes. The first, $\mathit{UntrustedActive}_{lang}$, defines the possible behaviours of the most general object that is currently active in a single-threaded system. The second, $\mathit{Untrusted}_{lang}$, does the same for the most general object that is currently inactive in a single-threaded system. These processes appear in Snippet 2.2.

In a single-threaded system, an object that exposes the facets *facets* and possesses the capabilities *caps* and data *data* and is currently active may perform all of the behaviours that $\mathit{UntrustedActive}_{lang}(\mathit{facets}, \mathit{caps}, \mathit{data})$ may perform. Such an object may invoke any capability $c \in \mathit{caps} - \mathit{facets}$ that is possesses that designates some other object besides itself. Having done so, it naturally becomes inactive. Alternatively, it may invoke a capability $\mathit{to} \in \mathit{facets}$ that refers to itself, in which case it remains active. When active, it cannot be invoked by any other object, however, because all other objects are inactive.

When the same object is currently inactive, it may perform all of the behaviours that $\mathit{Untrusted}_{lang}(\mathit{facets}, \mathit{caps}, \mathit{data})$ may perform. Such an object can only wait to be invoked by another, for which the reply capability included in the invocation is $\mathit{from} \in \mathit{Capability} - \mathit{facets}$. Once invoked, this

$$
\begin{aligned}
&UntrustedActive_{lang}(facets, caps, data) = \\
&\quad \left(\begin{array}{l} ?from : facets?c : caps - facets?op?arg : caps \cup data \cup \{\mathsf{null}\} \rightarrow \\ \quad Untrusted_{lang}(facets, caps, data) \ \Box \\ ?from : facets?to : facets?op?arg : caps \cup data \cup \{\mathsf{null}\} \rightarrow \\ \quad UntrustedActive_{lang}(facets, caps, data) \end{array} \right) \\
&\quad \sqcap STOP,
\end{aligned}
$$

$$
\begin{aligned}
&Untrusted_{lang}(facets, caps, data) = \\
&\quad \left(\begin{array}{l} ?from : Capability - facets?to : facets?op?arg \rightarrow \\ \quad \textbf{let } C' = \{arg, from\} \cap Capability \ ; D' = \{arg\} \cap Data \textbf{ within} \\ \quad UntrustedActive_{lang}(facets, caps \cup C', data \cup D') \end{array} \right) \\
&\quad \sqcap STOP.
\end{aligned}
$$

Snippet 2.2: The most general active and inactive objects respectively.

object may obtain the reply capability *from* and whatever argument *arg* was contained in the invocation, and becomes active.

For simplicity, these definitions forbid an active object from being able to send a message and then remain active. Such a facility, as is provided in the single-threaded subset of E for instance, is useful only for non-blocking sends. Our single-threaded system model would therefore need to be extended further to cover single-threaded systems, like the single-threaded subset of E, in which non-blocking invocations can occur.

With these definitions, we can easily give a formal definition of a single-threaded object-capability system as a straightforward special case of Definition 2.3.1.

Definition 2.3.2 (Single-Threaded Object-Capability System)**.** An object-capability system, $(Object, behaviour, facets, Data)$, is *single-threaded* iff there exists a single initially active object $p \in Object$ such that

$$
\begin{aligned}
&UntrustedActive_{lang}(facets(p), caps(p), data(p)) \sqsubseteq_{FD} behaviour(p) \ \wedge \\
&\forall\, o \in Object - \{p\} \bullet \\
&\quad Untrusted_{lang}(facets(o), caps(o), data(p)) \sqsubseteq_{FD} behaviour(o).
\end{aligned}
$$

To illustrate this idea, suppose the system depicted in Figure 2.1 was a single-threaded object-capability system. One of the objects, Alice or Bob, must be initially active, with the other being initially inactive. Suppose Alice is the initially active object. Recall that the behaviour of each is completely unknown and so each must be modelled as being completely untrusted. Then this single-threaded system could be modelled by setting $behaviour(\mathsf{Alice}) = UntrustedActive_{lang}(facets(\mathsf{Alice}), \{\mathsf{Alice}\}, Data)$ and $behaviour(\mathsf{Bob}) = Untrusted_{lang}(facets(\mathsf{Bob}), \{\mathsf{Bob}\}, Data)$, keeping the other definitions unchanged.

At this point, the rationale behind the use of the subscripts "*OS*" and "*lang*" on the *Untrusted* processes is apparent, since (as shown in Table 2.1) the majority of object-capability operating systems are concurrent, while the majority of object-capability languages are single-threaded.

One can see by inspecting the syntax of $\mathit{UntrustedActive}_{lang}$ and $\mathit{Untrusted}_{lang}$, that all behaviours of each can be exhibited by $\mathit{Untrusted}_{OS}$, as one would expect. This leads to the following result.

Theorem 2.3.3. Given any three sets F, C and D,

$$\mathit{Untrusted}_{OS}(F, C, D) \sqsubseteq_{FD} \mathit{UntrustedActive}_{lang}(F, C, D)$$

and

$$\mathit{Untrusted}_{OS}(F, C, D) \sqsubseteq_{FD} \mathit{Untrusted}_{lang}(F, C, D).$$

The following result states formally that in any single-threaded object-capability system, as defined by Definition 2.3.2, only one object is ever active at a time. It is straightforward to prove and is used later on in this thesis.

Lemma 2.3.4. Let $(\mathit{Object}, \mathit{behaviour}, \mathit{facets})$ be a single-threaded object-capability system captured by the CSP process $\mathit{System} = \|_{o \in \mathit{Object}} (\mathit{behaviour}(o), \alpha(o))$. Then for all traces $s \in \mathit{traces}(\mathit{System})$, there exists a unique object that is active after System has performed s.

Proof. This result is proved straightforwardly by induction on the length of s; its proof has been omitted for brevity. □

2.3.6 Data Independence

Recall the system depicted in Figure 2.1, and our proof that in it neither Alice nor Bob can ever send a message to the other. Recall that we proved this result with the set *Data* of all data in the system being just a singleton set. However, one intuitively expects that this result should hold no matter what data exists in this system, *i.e.* that it should hold for all choices for the set *Data*.

There are two reasons for this intuitive expectation. Firstly, the behaviour of each of the objects in this system is largely unconcerned with data; varying the choice for the set *Data* should not affect how these objects behave other than to affect the data that may be contained in messages they send and receive, of course. Secondly, the property we are testing is also unconcerned with data, so varying the set *Data* should not affect whether it holds or not.

These intuitions are captured formally by the concept of *data-independence* [Wol86]. Informally, a system is data-independent in a data type when it handles members of that type uniformly, not distinguishing one member of that type from another. The theory of data-independence

for CSP is due mostly to Lazić [Laz99]. A summary of the main results appears in [Ros97, Chapter 12]. We may apply these results to show that the property $cannotOccur_A(System)$ that we proved about the system *System* depicted in Figure 2.1 will hold for any choice of the set *Data*.

Informally, a CSP process that is *data-independent* in some type (equivalently some set) T may be written so as to be parameterised by T such that the only things it does with members of T are to [Laz99, p. 55]:

- use the "?" or "$" operators to offer or choose between members of T, as in $?x : T$ or $\$x : T$;
- store them, perhaps in its local variables;
- use the "!" operator to perform events whose components contain values of T that were previously stored in a local variable; and
- perform equality tests between members of T, either explicitly (*e.g.* as in **if** $x = y$ **then** P **else** Q) or implicitly (*e.g.* through the synchronisation of two processes, as in $c!x \to STOP \underset{\{|c|\}}{\parallel} c!y \to STOP$ which will proceed only if $x = y$).

A number of further syntactic requirements are also imposed on a CSP process that is to be data-independent in T. In particular, its program text must not use or mention [Ros97, Section 12.2.2]:

- concrete members of T;
- operations on values of T, other than those which treat values of T as opaque tokens such as tuple and list formation and other polymorphic operations;
- predicates on values of T, other than equality tests;
- operations such as $|T|$ that reveal information about T (in this case its size); and
- replicated constructs (*e.g.* $\Big\|_{t \in T}$) whose indexing set depends in any way on T other than nondeterministic selection (*e.g.* $\$x : T$).

A range of data-independence theorems exist, each of which applies in different circumstances. Each of these theorems is used to show that a process P_T, which is data-independent in some type T, satisfies a certain property *Prop* for all choices of T if $Prop(P_T)$ holds for just a few concrete choices for the set T whose size is less than or equal to some *threshold*.

The simplest of these theorems is Theorem 2.3.5 below, which is a restatement of [Ros97, Theorem 15.2.1]. It requires that the process P_T satisfies the semantic condition $\mathbf{NoEqT}_T$ and that the property *Prop* being tested doesn't depend on the value of T. A process P_T satisfies $\mathbf{NoEqT}_T$

iff it never needs to test two values of T for equality. Theorem 2.3.5 is used to show that some property encoded as a refinement assertion $Spec_T \sqsubseteq P_T$ holds for all choices of T if $Spec_T \sqsubseteq P_T$ holds when T is a singleton set, *i.e.* it gives a data-independence threshold for T of 1 for the refinement $Spec_T \sqsubseteq P_T$. The theorem requires that the specification $Spec_T$ also satisfies $\mathbf{NoEqT}_T$, plus the extra restriction that the only operation that $Spec_T$ can perform on members of T is to use "$" to nondeterministically choose members of T, as in $\$x : T$.

Theorem 2.3.5. Suppose $Spec_T$ and $System_T$ are processes that are data-independent in T and both satisfy $\mathbf{NoEqT}_T$. Suppose further that the only operation that $Spec_T$ performs on members of T is to choose between them using the nondeterministic selection operator, as in $\$x : T$. Then, letting $\sqsubseteq$ mean either $\sqsubseteq_T$, $\sqsubseteq_F$ or $\sqsubseteq_{FD}$,

$$Spec_T \sqsubseteq System_T$$

holds for for all non-empty choices for T if it holds for any non-empty choice for T, including a singleton set.

For the case of traces-refinement, $Spec_T$ may also use the query operator to choose between members of T, as in $?x : T$.

Consider the CSP process *System* from Section 2.3.2 that models the object-capability system depicted in Figure 2.1. Treating the set *Data* as a type, *System* is data-independent in the type *Data*. Since it never tests two values of *Data* for equality, *System* also satisfies $\mathbf{NoEqT}_{Data}$. The specification $CHAOS_{\Sigma - A}$ against which this system was tested in Section 2.3.2 is totally unconcerned with *Data* and satisfies the conditions of Theorem 2.3.5 with respect to the type *Data*. Hence, we may apply Theorem 2.3.5 to the refinement check $CHAOS_{\Sigma - A} \sqsubseteq_T System$. Recall that this check passed when *Data* was instantiated as a singleton set. Hence, by Theorem 2.3.5, we conclude that this same property holds for any choice of the set *Data*, *i.e.* irrespective of the data present in the object-capability system.

Most of the security properties and patterns that we consider in this thesis are fairly unconcerned with data. In fact, almost all systems analysed in this thesis are data-independent in the set *Data* and satisfy $\mathbf{NoEqT}_{Data}$. Because most of the security properties that we test in this thesis are unconcerned with data, most of the specifications used in the refinement checks that embody those security properties satisfy the conditions of Theorem 2.3.5 with respect to the set *Data*. Therefore, unless stated otherwise, for all systems in this thesis we instantiate the set *Data* to be the singleton set $Data = \{\mathsf{SomeDatum}\}$. Using Theorem 2.3.5, each result that we obtain then naturally generalises to any choice for the set *Data* in each of these systems.

Some patterns analysed in this thesis (such as the Data Diode pattern considered in Chapter 5) do make use of data, and when analysing these we

will have to take data into account explicitly. This is why we have included data in our framework for modelling object-capability systems. However, for the majority of patterns and properties that we consider, Theorem 2.3.5 allows us to quietly ignore data where it is not relevant, confident in the knowledge that it doesn't affect the results we obtain. The reader should assume, therefore, that each result obtained in this thesis is unaffected by the choice for *Data* unless we specifically instantiate the set *Data* to be something other than the singleton set $\{\mathsf{SomeDatum}\}$.

3 Safety

In this chapter, we show how safety properties of object-capability patterns can be examined in CSP. These are the simplest of the security properties considered in this thesis and serve as an ideal starting point before we look at more complicated properties in later chapters.

Following Lamport's now universal definition [Lam77] a *safety* property asserts that something (bad) *cannot* happen. Safety properties are usually expressed as refinement tests in the traces model.

We show how to reason about safety properties with reference to two related object-capability patterns designed to help an object securely handle arbitrary, possibly malicious, capabilities that it might be given. These patterns are the low-level building blocks from which larger security-enforcing object-capability patterns (such as the IOU protocol [Clo06] on which the DonutLab system relies [SMS05] and patterns [Mil06, Chapter 11] for *confinement* [Lam73] to name just a few) are composed. In this sense, they play a role similar to low-level cryptographic protocols[1], on which secure application-level protocols are constructed, that have long been targeted for formal verification.

In Sections 3.1 and 3.2, we model and reason about the safety properties of implementations of the *Trademarks* and *Sealer-Unsealer* patterns, first described in [Mor73], respectively. We show how the safety properties of each pattern can be expressed in terms of CSP traces refinement checks, for FDR to carry out, by defining appropriate specification processes. This allows us to define, and automatically verify, complicated safety properties, such as *safe coercion*, that would otherwise be difficult to state and prove.

We use FDR to help us iteratively derive safe implementations of each pattern. In doing so, we illustrate how this approach allows one to diagnose and correct vulnerabilities in patterns that arise in the presence of concurrency, as well as those that arise from recursive invocation.

3.1 Safe Authenticating Trademarks

In this section, we use FDR to help derive a safe implementation of the *Trademarks* [Mor73] pattern. This pattern is designed to allow an object o

[1] Thanks to David Wagner for this analogy.

to determine whether the object p referred to by an arbitrary capability that it has been given is of a particular kind. Knowing that p is of a certain kind might allow o to make assumptions about p's behaviour, for instance.

Instantiating the pattern constructs two objects, a *stamp* and a *guard*. The stamp can be used to *mark* objects with the stamp's unique trademark. The guard is then used to determine whether the object designated by an arbitrary capability *carries* the trademark of the corresponding stamp. In this case, we say that the object *passes* the guard. A guard therefore authenticates objects that have been stamped by the corresponding stamp.

The Trademarks pattern can be implemented trivially using EQ, which recall is an equality primitive available in some object-capability systems that, given two capabilities, returns true if and only if it can be determined that they both refer to exactly the same object and false if and only if it can be determined that they don't. In this trivial Trademarks implementation, corresponding stamps and guards each have access to a common *capability set* object. Stamping an object adds (the capability that designates) it to the set. The guard authenticates a capability, c, by simply testing for inclusion in the set by enumerating each capability, d, it contains and testing for each whether $c\ EQ\ d = \mathsf{true}$.

In many cases, however, it will be impractical to maintain such a capability set for each stamp-guard pair. In many object-capability systems that rely on automatic garbage collection mechanisms to reclaim objects that are no longer in use, keeping such a set may prevent defunct objects from being reclaimed. Generally, this solution imposes a storage cost on the stamp-guard pair that is proportional to the number of stamped capabilities. Authenticating a capability might also take time proportional to the number of stamped capabilities. A better solution would impose a constant storage and runtime cost regardless of the number of stamped capabilities.

3.1.1 Deriving a Safe Implementation

We can derive such a solution by adapting the implementation of an object-capability authentication pattern, due to Stiegler, known as the *Nontransferable Claim Check* [Sti06]. The basic idea is that each stamp-guard pair has access to a common *slot* object that is unique for each stamp-guard pair. A slot is able to store a single capability. Stamping an object involves passing it a capability to the stamp's slot.

To check whether the object that an arbitrary capability c designates has been stamped, the guard first clears the slot and then invokes c with a message, m, that instructs the object to store a capability to itself into its slot. Note that m does not contain any capabilities since we expect that if c has been stamped, it will already have been given a capability to the slot. We assume that each stamped object has access to a capability that designates itself. In many object-capability systems, objects automatically have access to such a capability. In systems without this feature, a stamped

object could be given a capability to itself at the same time that it is given its slot capability (*i.e.* while it is being stamped).

The guard's slot will be modified in response to message m if c has been stamped or c forwards m to a stamped object. When the invocation of c returns, any capability, d, contained in the slot can be compared against c. If $c\,EQ\,d = \mathsf{true}$, we assume that the original capability that was invoked, c, has been stamped since only stamped objects have a capability that allows them to store a copy of themselves in the slot. If the slot contains no such capability d, then we assume that c has not been stamped. If $c\,EQ\,d \neq \mathsf{true}$ we assume that c might have forwarded the invocation to an authentic object, d, and therefore might not have been stamped.

A Draft Implementation

We model this implementation in CSP as follows. We begin with a slot, which is modelled as an object with two facets, *readme* and *writeme*, for reading and modifying its contents respectively. The process $ASlot(readme, writeme, val)$ models a slot object with facets *readme* and *writeme* that initially contains the value *val*, and appears in Snippet 3.1.

$$
\begin{aligned}
&ASlot(readme, writeme, val) = \\
&\quad ?from : Capability - \{readme, writeme\}!readme!\mathsf{Call}!\mathsf{null} \rightarrow \\
&\qquad readme!from!\mathsf{Return}!val \rightarrow ASlot(readme, writeme, val) \;\Box \\
&\quad ?from : Capability - \{readme, writeme\}!writeme!\mathsf{Call}?newVal \rightarrow \\
&\qquad writeme!from!\mathsf{Return}!\mathsf{null} \rightarrow ASlot(readme, writeme, newVal)
\end{aligned}
$$

Snippet 3.1: The behaviour of a slot object.

Calling its read facet, *readme*, causes it to Return its current contents, *val*. Calling its write facet, *writeme*, passing a value, *newVal*, causes it to replace its current contents with *newVal*, Returning null in response.

A guard, *me*, whose slot's read- and write-facets are *slotR* and *slotW* respectively, is modelled as the process $AGuard(me, slotR, slotW)$, which is defined in Snippet 3.2.

A guard waits to be invoked with a Call message containing a capability, *specimen*, that designates the object to be authenticated. The guard then clears its slot by Calling its write facet, *slotW*, with the value null, waiting for it to Return. Once the slot has been cleared, the guard Calls the *specimen* to instruct it to store a capability to itself in its slot, waiting for it to Return. The guard then Calls the slot's read facet, *slotR*, to query its new value, *val*. We use equality between object names to simulate a reliable *EQ* primitive. If $val = specimen$ (*i.e.* the two are *EQ*) then the guard indicates that *specimen* is authentic by Returning a capability to itself to its invoker, *from*. It then reverts to its initial state. If $val \neq specimen$, the guard indicates that

$$
\begin{array}{l}
AGuard(me, slotR, slotW) = \\
\quad ?from : Capability - \{me\}!me!\mathsf{Call}?specimen : Capability \rightarrow \\
\quad me!slotW!\mathsf{Call}!\mathsf{null} \rightarrow slotW!me!\mathsf{Return}!\mathsf{null} \rightarrow \\
\quad me!specimen!\mathsf{Call}!\mathsf{null} \rightarrow specimen!me!\mathsf{Return}!\mathsf{null} \rightarrow \\
\quad me!slotR!\mathsf{Call}!\mathsf{null} \rightarrow slotR!me!\mathsf{Return}?val \rightarrow \\
\quad \textbf{if } val = specimen \textbf{ then} \\
\qquad me!from!\mathsf{Return}!me \rightarrow AGuard(me, slotR, slotW) \\
\quad \textbf{else } me!from!\mathsf{Return}!\mathsf{null} \rightarrow AGuard(me, slotR, slotW)
\end{array}
$$

Snippet 3.2: The behaviour of a guard.

specimen is not authentic by Returning null to its invoker before reverting to its initial state.

We model a stamped object, *me*, whose slot's write-facet is $slotW$, that possesses the capabilities *caps* and data *data*, as the process $AStamped(me, slotW, caps, data)$. The implementation of this pattern appears to place a number of unstated assumptions on the behaviour of a stamped object. Rather than trying to discern all of these assumptions *a priori*, we will use FDR to uncover them for us. To do so, we can begin with a relatively unconstrained model of a stamped object and successively refine it until we find one that meets the assumptions of the implementation, *i.e.* one in which the implementation is safe.

We start by making the obvious assumption that a stamped object should not pass its capability that designates its slot's write-facet, $slotW$, to any other object. We also choose to model the more general case in which a stamped object executes with its own thread of control, as occurs in an object-capability operating system for example. Any implementation that is safe in this context will also be safe in the more restrictive single-threaded context, in which all objects share a single thread of control[2].

$$
\begin{array}{l}
AStamped(me, slotW, caps, data) = \\
\left(\begin{array}{l}
me?to : caps?op?arg : (caps - \{slotW\}) \cup data \cup \{\mathsf{null}\} \rightarrow \\
\quad AStamped(me, slotW, caps, data) \ \Box \\
me!me?op?arg : caps \cup data \cup \{\mathsf{null}\} \rightarrow \\
\quad AStamped(me, slotW, caps, data) \ \Box \\
?from : Capability - \{me\}!me?op?arg \rightarrow \\
\quad \textbf{let } C' = \{arg, from\} \cap Capability \ ; D' = \{arg\} \cap Data \textbf{ within} \\
\quad AStamped(me, slotW, caps \cup C', data \cup D')
\end{array}\right) \\
\quad \sqcap STOP
\end{array}
$$

Therefore, $AStamped(me, slotW, caps, data)$ is similar to the model of

[2]This follows from Theorem 2.3.3 and the fact that safety being refinement-closed over the traces model implies it is also refinement-closed over the failures-divergences model, since any failures-divergences refinement of a process is also a traces-refinement.

a completely untrusted object executing with its own thread of control, $Untrusted_{OS}(\{me\}, caps, data)$, from Snippet 2.1, except that it never passes $slotW$ as an argument to an invocation. Indeed it represents the most general object that adheres to this restriction.

We analyse this implementation by instantiating the system depicted in Figure 3.1. Recall that our basic approach to analysing object-capability patterns involves coupling an instance of each pattern with untrusted objects that represent arbitrary objects that may exist in a system in which the pattern is instantiated. These untrusted objects should exhibit any and all behaviours permitted by the context in which the pattern is being analysed.

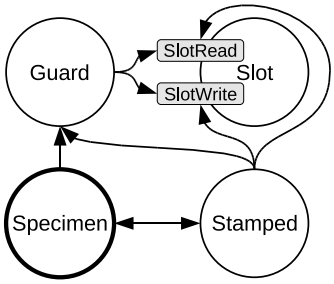

Figure 3.1: Instantiating the draft Trademarks implementation.

The instantiation depicted in Figure 3.1 comprises a stamped object, Stamped, whose corresponding slot and guard are Slot and Guard respectively. Notice that Stamped is given a capability to every facet of every object in the system. This is done because we don't want to place any constraints on the capabilities that may be possessed by a stamped object. The slot's read- and write-facets are SlotRead and SlotWrite respectively. The system also contains an untrusted object, Specimen, that represents an arbitrary unstamped object that may exist in a system in which this pattern is instantiated.

If this pattern is instantiated correctly, Specimen should not possess a capability to any of Slot's facets, since by definition Specimen does not carry Guard's trademark. Hence, we give Specimen all capabilities except those from $facets(\mathsf{Slot})$. Setting $Object = \{\mathsf{Guard}, \mathsf{Slot}, \mathsf{Stamped}, \mathsf{Specimen}\}$, $facets(\mathsf{Slot}) = \{\mathsf{SlotRead}, \mathsf{SlotWrite}\}$ and $facets(other) = \{other\}$ for $other \neq \mathsf{Slot}$, we have:

$$behaviour(\mathsf{Guard}) = AGuard(\mathsf{Guard}, \mathsf{SlotRead}, \mathsf{SlotWrite}),$$
$$behaviour(\mathsf{Stamped}) = AStamped(\mathsf{Stamped}, \mathsf{SlotWrite}, Capability, Data),$$
$$behaviour(\mathsf{Slot}) = ASlot(\mathsf{SlotRead}, \mathsf{SlotWrite}, \mathsf{null}),$$
$$behaviour(\mathsf{Specimen}) = Untrusted_{OS}(facets(\mathsf{Specimen}), Capability - facets(\mathsf{Slot}), Data).$$

The object-capability system $(Object, behaviour, facets, Data)$ is then instantiated as described in Section 2.3.1 yielding the CSP process $System$.

A safe guard is one that authenticates only stamped objects. Hence, we can define the process $SafeGuard(me, sObjs)$ that models just the behaviour of a guard with identity me that authenticates only those objects in the set $sObjs$. In doing so, we ignore the internal behaviour of the guard as well as the behaviour of other objects in the system, modelling only the guard's behaviour at its interface to its clients. $SafeGuard(me, sObjs)$ is an idealised model of a safe guard – *i.e.* one that never authenticates anything but stamped objects. It is defined in Snippet 3.3.

$$
\begin{array}{l}
SafeGuard(me, sObjs) = \\
\quad ?from : Capability - \{me\}!me!\mathsf{Call}?specimen : Capability \rightarrow \\
\quad \textbf{if } specimen \in sObjs \textbf{ then} \\
\qquad \left(\begin{array}{l} me!from!\mathsf{Return}!me \rightarrow SafeGuard(me, sObjs) \sqcap \\ me!from!\mathsf{Return}!\mathsf{null} \rightarrow SafeGuard(me, sObjs) \end{array}\right) \\
\quad \textbf{else } \; me!from!\mathsf{Return}!\mathsf{null} \rightarrow SafeGuard(me, sObjs)
\end{array}
$$

Snippet 3.3: The specification of a safe guard.

Notice that we use internal choice to model whether an authentication of a stamped object is successful or not. This is done because here we are interested in safety only; a safe guard doesn't always have to authenticate stamped objects, but merely never authenticate non-stamped objects.

We can test whether Guard is safe then by testing whether, in *System*, it can ever behave differently to its idealised representation in *SafeGuard*. We do so by testing whether

$$
\begin{array}{l}
SafeGuard(\mathsf{Guard}, \{\mathsf{Stamped}\}) \sqsubseteq_T \\
\quad System \setminus (\Sigma - SafeGuardEvents(\mathsf{Guard}))
\end{array}
$$

where $SafeGuardEvents(me)$ gives (a slight superset of) the events that can be performed at the client interface of a safe guard with identity me. It is defined simply as

$$
\begin{array}{l}
SafeGuardEvents(me) = \\
\quad \{|from.me.\mathsf{Call}, me.from.\mathsf{Return} \mid from \in Capability - \{me\}|\}.
\end{array}
$$

FDR reveals that this refinement doesn't hold. Exploring the counter-example reveals that *System* can perform the following trace.

$$
\begin{array}{l}
\langle \mathsf{Stamped.Guard.Call.Guard}, \; \mathsf{Guard.SlotWrite.Call.null}, \\
\mathsf{SlotWrite.Guard.Return.null}, \; \underline{\mathsf{Stamped.SlotWrite.Call.Guard}}, \\
\mathsf{Guard.Guard.Call.null}, \; \mathsf{Guard.Guard.Return.null}, \\
\mathsf{SlotWrite.Stamped.Return.null}, \; \mathsf{Guard.SlotRead.Call.null}, \\
\mathsf{SlotRead.Guard.Return.Guard}, \; \mathsf{Guard.Stamped.Return.Guard} \rangle
\end{array}
$$

The problem is evident from the trace above, in which we have underlined the cause of the error. We see that Stamped places Guard in the slot in between when Guard clears and subsequently checks Slot's contents.

Refining the Implementation

We can see from the counter-example above that a stamped object must never try to store anything in its slot other than a capability to itself. We can therefore refine the behaviour of a stamped object to impose just this further restriction. We redefine the process *AStamped* as appears in Snippet 3.4.

$$
\begin{array}{l}
AStamped(me, slotW, caps, data) = \\
\quad \textbf{let}\ \ caps' = caps - \{slotW\}\ \textbf{within} \\
\quad \left(\begin{array}{l}
me?to : caps'?op?arg : caps' \cup data \cup \{\mathsf{null}\} \rightarrow \\
\quad AStamped(me, slotW, caps, data)\ \Box \\
me!me?op?arg : caps \cup data \cup \{\mathsf{null}\} \rightarrow \\
\quad AStamped(me, slotW, caps, data)\ \Box \\
me!slotW!\mathsf{Call}!me \rightarrow AStamped(me, slotW, caps, data)\ \Box \\
?from : Capability - \{me\}!me?op?arg \rightarrow \\
\quad \textbf{let}\ \ C' = \{arg, from\} \cap Capability\ ;\ D' = \{arg\} \cap Data\ \textbf{within} \\
\quad AStamped(me, slotW, caps \cup C', data \cup D')
\end{array} \right) \\
\quad \sqcap\ STOP
\end{array}
$$

Snippet 3.4: The behaviour of a safe stamped object.

Reinstantiating the system using this new definition of the behaviour of a stamped object and repeating the refinement-test in FDR reveals that it does indeed now hold. The extra restriction ensures that, in this small system, Guard can authenticate only Stamped and nothing else. However, it should be noted that this analysis is far from exhaustive. The limitations of this analysis are discussed further in Section 3.1.2.

Recursive Reentrancy

In all object-capability languages, including E, Joe-E and Cajita, objects may be recursively invoked. It is worth considering the implications of this if our Trademarks implementation were to be deployed in an object-capability language. Suppose it were, producing a guard object, `guard`. Suppose `guard` is invoked with a specimen, `specimen`. At the point that `guard` calls `specimen` to tell it to place a capability to itself in its slot, `specimen` may be able to recursively call `guard`.

This kind of synchronous recursive invocation creates concurrency that might cause `guard` to behave unpredictably. We would like to know that our guard implementation remains safe in this kind of situation, otherwise deploying it in an object-capability language would require the programmer to implement extra functionality to prevent it from being recursively invoked while invoking a specimen[3].

[3]This can be achieved for example by adding a boolean flag, `inuse`, that is set before

Our model of a guard, $AGuard(me, slotR, slotW)$, with identity me and slot capabilities $slotR$ and $slotW$, does not allow for this kind of recursive invocation. We can extend it, however, to allow for it, giving the process $AGuardR(me, slotR, slotW, rtstack, maxsz)$ that includes a model of a call-stack, $rtstack$, whose maximum size is $maxsz$. This process appears in Snippet 3.5.

$$
\begin{array}{l}
AGuardR(me, slot, rtstack, maxsz) = \\
\quad \left(\begin{array}{l}
\#(rtstack) < maxsz \ \& \\
?from : Capability - \{me\}!me!\mathsf{Call}?specimen : Capability \rightarrow \\
me!slotW!\mathsf{Call}!\mathsf{null} \rightarrow slotW!me!\mathsf{Return}?!\mathsf{null} \rightarrow \\
me!specimen!\mathsf{Call}!\mathsf{null} \rightarrow \\
AGuardR(me, slot, \langle\langle from, specimen\rangle\rangle \hat{} \, rtstack, maxsz)
\end{array} \right) \\
\quad \Box \\
\quad \left(\begin{array}{l}
\#(rtstack) > 0 \ \& \\
\textbf{let } \langle\langle from, specimen\rangle\rangle \hat{} \, st = rtstack \textbf{ within} \\
specimen!me!\mathsf{Return}!\mathsf{null} \rightarrow \\
me!slotR!\mathsf{Call}!\mathsf{null} \rightarrow slotR!me!\mathsf{Return}?val \rightarrow \\
\textbf{if } val = specimen \textbf{ then} \\
\ me!from!\mathsf{Return}!me \rightarrow AGuardR(me, slot, st, maxsz) \\
\textbf{else} \\
\ me!from!\mathsf{Return}!\mathsf{null} \rightarrow AGuardR(me, slot, st, maxsz)
\end{array} \right)
\end{array}
$$

Snippet 3.5: The behaviour of a recursively invocable guard.

If there is room on the stack to store another invocation, then the object is willing to be Called. When it invokes a *specimen* it places a record of the Call that it is currently processing on the stack in the form of a 2-element sequence, $\langle from, specimen\rangle$, that contains the object that Called it, *from*, as well as the specimen it is invoking, *specimen*. The object is willing to be Returned to only if there is a record on the stack. In this case, it is willing to accept a Return only from the last specimen, *specimen*, it Called whose name will be on the top of the stack.

Note that all invocations other than to specimens are modelled as before, not allowing for recursive invocation while they complete. This is sound because the usual context in which recursive invocation can occur is single-threaded. Hence, the only objects that can invoke the guard whilst these invocations are completing are those that have been invoked themselves. These other invocations are performed on objects that can be relied upon not to recursively invoke the guard, and this can be confirmed by inspecting their code. Hence, the guard cannot be invoked during these other invocations in

the invocation of the specimen and cleared after the invocation returns. The object must then check whether it is not already `inuse` before servicing invocations.

$$
\begin{array}{l}
SafeGuardR(me, sObjs, rtstack, maxsz) = \\
\quad \left(\begin{array}{l}
\#(rtstack) < maxsz \;\&\; \\
?from : Capability - \{me\}!me!\mathsf{Call}?specimen : Capability \rightarrow \\
SafeGuardR(me, sObjs, \langle\langle from, specimen\rangle\rangle \hat{} \, rtstack, maxsz)
\end{array}\right) \\
\quad \Box \\
\quad \left(\begin{array}{l}
\#(rtstack) > 0 \;\&\; \\
\textbf{let } \langle\langle from, specimen\rangle\rangle \hat{} \, st = rtstack \textbf{ within} \\
\textbf{if } specimen \in sObjs \textbf{ then} \\
\quad \left(\begin{array}{l}
me!from!\mathsf{Return}!me \rightarrow SafeGuardR(me, sObjs, st, maxsz) \\
\sqcap \\
me!from!\mathsf{Return}!\mathsf{null} \rightarrow SafeGuardR(me, sObjs, st, maxsz)
\end{array}\right) \\
\textbf{else} \\
\quad me!from!\mathsf{Return}!\mathsf{null} \rightarrow SafeGuardR(me, sObjs, st, maxsz)
\end{array}\right)
\end{array}
$$

Snippet 3.6: The specification of a safe recursively invocable guard.

the context in which we are analysing this pattern.

All vulnerabilities in object-capability patterns, of which the author is aware (see *e.g.* [Wag08b, Wag08a]), that have arisen due to recursive invocation require the vulnerable object to be recursively invoked only once. We can test whether a single recursive invocation of a guard can make it unsafe by reinstantiating the first safe guard implementation in the context of the system depicted in Figure 3.1, setting the behaviour of the guard, Guard, to $AGuardR(\mathsf{Guard}, \mathsf{SlotRead}, \mathsf{SlotWrite}, \langle\rangle, 2)$. Doing so gives us the new process $System$.

A recursive guard with a maximum call-stack size of 1 should be equivalent to a non-recursive guard. Hence, $AGuard(\mathsf{Guard}, \mathsf{SlotRead}, \mathsf{SlotWrite})$ should be equivalent to $AGuardR(\mathsf{Guard}, \mathsf{SlotRead}, \mathsf{SlotWrite}, \langle\rangle, 1)$. Testing this equivalence in FDR, by testing whether each process failures-divergences refines the other, reveals that it does indeed hold.

To test whether Guard is still safe now that it can be recursively invoked, we need to extend the definition of a working guard, $SafeGuard(me, sObjs)$ with identity me that authenticates only the objects in the set $sObjs$, to also be recursively invocable. Doing so yields the process $SafeGuardR(me, sObjs, rtstack, maxsz)$, which appears in Snippet 3.6.

Again, we expect that $SafeGuard(\mathsf{Guard}, \{\mathsf{Stamped}\})$ should be (failures-divergences) equivalent to $SafeGuardR(\mathsf{Guard}, \{\mathsf{Stamped}\}, \langle\rangle, 1)$. Testing this assertion in FDR reveals that it holds.

We can then test whether single recursive invocations to Guard can make it unsafe by testing whether

$$
\begin{array}{l}
SafeGuardR(\mathsf{Guard}, \{\mathsf{Stamped}\}, \langle\rangle, 2) \sqsubseteq_T \\
\quad System \setminus (\Sigma - SafeGuardEvents(\mathsf{Guard})).
\end{array}
$$

FDR reveals that this refinement does hold. Hence, we have some confidence that our Trademarks implementation can be applied in an object-capability language without fearing that recursive invocations will make it unsafe.

3.1.2 Summary

We have derived a safe Trademarks implementation, built using *EQ*, that functions in the presence of concurrently executing objects and recursive invocation. Subject to the caveats regarding exhaustiveness below, our analysis indicates that a guard will never authenticate a non-stamped object so long as each of its stamped objects never (1) divulges any of its slot capabilities, and (2) never tries to place anything in any of its slots other than a capability to itself.

In analysing this pattern, we have considered it in just a small context involving only a handful of objects. This means that our analysis is far from exhaustive. We have considered, for example, systems in which a guard has stamped only a single object, and that contain only one unstamped object (besides the guard and its slot). Also, when considering recursive invocation, our model allowed only a single recursive call to occur.

These analyses, therefore, give us confidence that our pattern is safe. However, on their own, they do not prove that it will be safe in all possible deployments. Later, in Chapter 4, we will show how the results from these small analyses can be generalised to systems of arbitrary size. In doing so we obtain results that are more exhaustive than those obtained here, allowing us to assert our safety results with greater confidence.

3.2 Safe Coercing Sealer-Unsealers

The Trademarks implementation of the previous section is an example of an *authentication* pattern; a guard authenticates capabilities that carry its trademark. We now consider a similar pattern that is based instead on the idea of *coercion* [TMHK95].

We explain coercion in the context of the Trademarks pattern, since this is familiar; however, it can be applied in other cases too. We will analyse one such case that extends the basic idea slightly. Coercion works as follows. Suppose an object o is given a capability c, purportedly carrying a particular trademark. o can perform some operation with c but is willing to do so only if c carries the purported trademark. o sends a message that contains c to a *coercer* object that is associated with the trademark that c purportedly carries. The coercer object will respond with one of two messages indicating success or failure respectively. If the message indicates success, it will contain a capability d that o can be sure carries the purported trademark. In this case, we say that c has been *coerced* to d. Otherwise a message indicating failure contains no capabilities and o cannot assume that c has the purported

trademark.

Note that even if c successfully coerces, o cannot infer that c carries the purported trademark. c might, for instance, be a proxy that forwards invocations to d, thereby allowing c to be coerced to d. The disadvantages of coercion over authentication are clear. No matter whether coercion is successful or not, o cannot determine if c was authentic. In many cases, this information could be vital.

Coercion does have the advantage that it doesn't require a discrimination primitive like EQ in order to implement. As we will see later in Chapter 4 and the ones that follow it, patterns that use EQ make the task of generalising analysis results for them more difficult than those that do not. Hence, avoiding EQ can have its advantages.

We consider the implementation of a pattern that uses coercion. The pattern[4] is known as the *Sealer-Unsealer* pattern [Mor73]. Instantiating the Sealer-Unsealer pattern creates two objects, a *sealer* and a corresponding *unsealer*. Invoking the sealer, passing it a capability c, causes it to return a capability b to an opaque *box* object. We say that c is the *contents* of the box b, or that c has been *sealed* to produce b. b conveys no authority on its own, except when used with the unsealer that corresponds to the sealer that produced b. The corresponding unsealer is used to coerce a capability b', that purportedly refers to a box b, to the box's contents c.

3.2.1 Deriving a Safe Implementation

We use FDR to help us analyse the safety of an implementation of this pattern, whose basic structure is due to Stiegler [Sti04]. Here, each sealer-unsealer pair has access to a unique slot object. When creating a new box, b, the sealer hands b a capability to its slot object. When invoked with a purported box, b', the unsealer first clears its slot before invoking b' with a message m instructing it to place its contents in its slot. When this invocation returns, the unsealer simply checks its slot and returns whatever capability the slot contains, if any. The intuition is that only valid boxes have access to the unsealer's slot object and that each box is trusted to place only its contents in its slot.

If b' is a valid box, then upon receiving m, it will place c in its slot, thereby allowing the unsealer to coerce b' to c successfully. Alternatively, b' might be some kind of proxy that forwards messages to a valid underlying box b. If b' chooses to forward m, the coercion will be successful; otherwise, if this pattern functions correctly, the coercion should fail and the unsealer should not return c.

[4]Note that this pattern can also be implemented using authentication rather than coercion [Yee99, VH04] (although with naturally different security properties). Languages that provide support for unforgeable record field names allow a particularly elegant implementation [VH04, p. 204].

Notice that this implementation is very similar to the Trademarks implementation presented earlier. One notable difference is that an unsealer, unlike a guard, does not apply any primitive test, such as EQ, to the slot's contents before deciding what to return to its invoker. Despite this similarity, the security properties of these patterns are very different.

A Draft Implementation

Slots have the same behaviour as before. We model an unsealer with identity me and slot capabilities $slotR$ and $slotW$ as the process $AnUnsealer(me, slotR, slotW)$, defined as follows[5].

$$
\begin{aligned}
&AnUnsealer(me, slotR, slotW) = \\
&\quad ?from : Capability - \{me\}!me!\mathsf{Call}?specimen : Capability \rightarrow \\
&\quad me!slotW!\mathsf{Call}!\mathsf{null} \rightarrow slotW!me!\mathsf{Return}!\mathsf{null} \rightarrow \\
&\quad me!specimen!\mathsf{Call}!\mathsf{null} \rightarrow specimen!me!\mathsf{Return}!\mathsf{null} \rightarrow \\
&\quad me!slotR!\mathsf{Call}!\mathsf{null} \rightarrow slotR!me!\mathsf{Return}?val \rightarrow \\
&\quad me!from!\mathsf{Return}!val \rightarrow AnUnsealer(me, slotR, slotW)
\end{aligned}
$$

When Called with a capability, *specimen*, that potentially refers to a box sealed with the corresponding sealer, our unsealer implementation first clears its slot before Calling the specimen, telling it to place its contents in its slot. The unsealer then reads the contents, *val*, of the slot and Returns *val* to its invoker, *from*.

A box, *me*, whose capability to the write-facet of its slot is *slotW* and whose contents is *contents*, is modelled by the process $ABox(me, slotW, contents)$, defined in Snippet 3.7. When Called, a box simply places its *contents* in its slot, Returning null.

$$
\begin{aligned}
&ABox(me, slotW, contents) = \\
&\quad ?from : Capability - \{me\}!me!\mathsf{Call}!\mathsf{null} \rightarrow \\
&\quad me!slotW!\mathsf{Call}!contents \rightarrow slotW!me!\mathsf{Return}!\mathsf{null} \rightarrow \\
&\quad me!from!\mathsf{Return}!\mathsf{null} \rightarrow ABox(me, slotW, contents)
\end{aligned}
$$

Snippet 3.7: The behaviour of a box.

At this point, we are ready to perform a safety analysis on this pattern. Before doing so, it is worth considering what safety properties we should test. Consider, for example, the following safety property that is analogous to the one applied earlier to the Trademarks pattern: "an unsealer should never successfully unseal anything other than a valid box". This property is unsuitable for this pattern because the pattern is based on coercion rather than authentication. This means that if we invoke an unsealer, passing a

[5] This implementation is derived from [MSL+07, Figure 13].

specimen to it that is really a proxy to a valid box, that the specimen should rightly be unsealed, for the same reason that a proxy should rightly coerce to the valid underlying object for which it is a proxy.

Therefore, the safety property that we want to verify for this pattern is that an unsealer should Return the contents of one of its boxes b, to an object that has Called it, only if the Call included a valid capability that proxies to b. An object proxies to b in response to being invoked if it is b or invokes an object that proxies to b. An unsealer never proxies by definition.

This property is captured by the specification process *SafeUnsealer* which appears in Snippet 3.8. Given a system captured by a process *System*, that contains an unsealer *unsealer* and corresponding box bx, whose contents is *conts* and slot is *slot*, we can test whether this unsealer is safe in *System* by testing whether

$$SafeUnsealer(unsealer, bx, conts) \sqsubseteq_T System \setminus \alpha(slot).$$

The property captured by $SafeUnsealer(unsealer, bx, conts)$ is that, once *unsealer* has been Called by some object *from* passing some specimen *specimen*, and once *unsealer* has Called *specimen* and received *specimen*'s subsequent Return message, that *unsealer* will Return *conts* to *from* only if *specimen* proxied to bx when *unsealer* Called it.

This is the reason that we cannot hide the events of objects other than *slot*, when performing the refinement test above, since their behaviour may need to be tracked in order to determine whether *specimen* proxies to bx.

Initially, *unsealer* should be waiting to be Called by some object *from*, with some argument capability *specimen*. Other objects may invoke each other while *unsealer* is waiting. This is captured by the two sides of the external choice "$\Box$" in *SafeUnsealer*. Once *unsealer* is Called, the specification transitions to the process $SafeUnsealer'(unsealer, bx, conts, from, specimen)$, which remembers which object, *from*, Called *unsealer* and what argument, *specimen*, they passed. The processes $SafeUnsealer''$ and $SafeUnsealer'''$ represent subsequent specification states and can be ignored for now. When in the state captured by $SafeUnsealer'$, *unsealer* Calls *specimen*. However, while this invocation is taking place, other invocations may also occur in the system. This is captured by the external choice in $SafeUnsealer'$.

Once *specimen* has been Called, the specification transitions to the process $SafeUnsealer''(objectof(specimen), specimen = bx)$.[6] The first argument here is the name of the object that has the facet *specimen*. The second indicates whether *specimen* has proxied to bx, which is true at this point if and only if $specimen = bx$.

The job of $SafeUnsealer''(current, hasProxied)$ is to monitor the invocations that occur in *System* following *unsealer*'s Call to *specimen*, and update *hasProxied* accordingly as to whether *specimen* has proxied to bx in response to *unsealer*'s Call. It therefore keeps track, via the parameter *current*, of

[6]The function *objectof* maps a facet name to the unique object that has this facet.

$$otherthan(o) = Capability - facets(o)$$

$$\begin{array}{l}
SafeUnsealer(unsealer, bx, conts) = \\
\quad ?from : otherthan(unsealer)!unsealer!\mathsf{Call}?specimen : Capability \rightarrow \\
\quad\quad SafeUnsealer'(unsealer, bx, conts, from, specimen)\ \Box \\
\quad ?other : otherthan(unsealer)?to : otherthan(unsealer)?op?arg \rightarrow \\
\quad\quad SafeUnsealer(unsealer, bx, conts)
\end{array}$$

$$\begin{array}{l}
SafeUnsealer'(unsealer, bx, conts, from, specimen) = \\
\quad \textbf{let} \\
\quad\quad SafeUnsealer''(current, hasProxied) = \\
\quad\quad\quad ?c : facets(current)?to : otherthan(unsealer)?op?arg \rightarrow \\
\quad\quad\quad\quad SafeUnsealer''(objectof(to), hasProxied \vee (to = bx))\ \Box \\
\quad\quad\quad ?other : otherthan(current)?to : otherthan(unsealer)?op?arg \rightarrow \\
\quad\quad\quad\quad SafeUnsealer''(current, hasProxied)\ \Box \\
\quad\quad\quad specimen!unsealer!\mathsf{Return}!\mathsf{null} \rightarrow SafeUnsealer'''(hasProxied) \\
\quad\quad SafeUnsealer'''(hasProxied) = \\
\quad\quad\quad ?other : otherthan(unsealer)?to : otherthan(unsealer)?op?arg \rightarrow \\
\quad\quad\quad\quad SafeUnsealer'''(hasProxied)\ \Box \\
\quad\quad\quad \textbf{if}\ hasProxied\ \textbf{then} \\
\quad\quad\quad\quad unsealer!from!\mathsf{Return}\$c : \{conts, \mathsf{null}\} \rightarrow \\
\quad\quad\quad\quad\quad SafeUnsealer(unsealer, bx, conts) \\
\quad\quad\quad \textbf{else}\ \ unsealer!from!\mathsf{Return}!\mathsf{null} \rightarrow SafeUnsealer(unsealer, bx, conts) \\
\quad \textbf{within} \\
\quad\quad unsealer!specimen!\mathsf{Call}!\mathsf{null} \rightarrow \\
\quad\quad\quad SafeUnsealer''(objectof(specimen), specimen = bx)\ \ \Box \\
\quad\quad ?other : otherthan(unsealer)?to : otherthan(unsealer)?op?arg \rightarrow \\
\quad\quad\quad SafeUnsealer'(unsealer, bx, conts, from, specimen)
\end{array}$$

Snippet 3.8: The specification of a safe coercing unsealer.

the most recent object that has been invoked in the chain of invocations beginning at *unsealer*'s Call to *specimen*. Originally, this object is, of course, $objectof(specimen)$. If *current* invokes a capability *to*, $objectof(to)$ becomes the new *current* and *hasProxied* is updated accordingly. *hasProxied* is set from false to true precisely when the object that *current* invokes is bx. Once it becomes true, it remains that way.

While *current*'s invocation is taking place, other invocations may of course occur in the system. Alternatively, *specimen* may Return from *unsealer*'s original Call, in which case the specification transitions to $SafeUnsealer'''(hasProxied)$. This process simply allows *unsealer* to Return *conts* to *from* (*unsealer*'s original Caller), only if *hasProxied* is true. Of course, invocations between other objects may take place while *unsealer* is Returning to *from*, hence this process also allows these to occur.

To analyse this pattern, we instantiate it in the context of the system depicted in Figure 3.2. Here, we see an unsealer Unsealer, and associated box Box and slot Slot, as well as two untrusted objects, Contents and Alice. Contents is the contents of Box while Alice is an arbitrary object that interacts with this pattern. Each has as many capabilities as possible, meaning that each has all capabilities other than those to Slot. This is done because we don't want to place any restrictions on the objects that may be the contents of a box or may interact with this pattern respectively, other than those imposed naturally by the pattern itself.

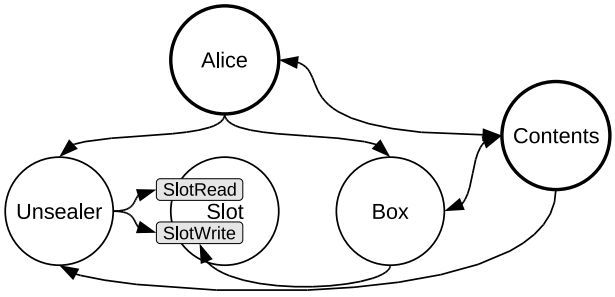

Figure 3.2: Instantiating the Sealer-Unsealer implementation.

The set *Object* and function *facets* are set as one would expect from Figure 3.2. The behaviour of each object is also as one would expect, namely:

$$
\begin{aligned}
&behaviour(\mathsf{Box}) = ABox(\mathsf{Box}, \mathsf{SlotWrite}, \mathsf{Contents}),\\
&behaviour(\mathsf{Slot}) = ASlot(\mathsf{SlotRead}, \mathsf{SlotWrite}, \mathsf{null}),\\
&behaviour(\mathsf{Unsealer}) = AnUnsealer(\mathsf{Unsealer}, \mathsf{SlotRead}, \mathsf{SlotWrite}),\\
&behaviour(\mathsf{Alice}) =\\
&\quad Untrusted_{OS}(facets(\mathsf{Alice}), Capability - facets(\mathsf{Slot}), Data),\\
&behaviour(\mathsf{Contents}) =\\
&\quad Untrusted_{OS}(facets(\mathsf{Contents}), Capability - facets(\mathsf{Slot}), Data).
\end{aligned}
$$

Notice that the system $(Object, facets, behaviour, Data)$ is not a single-threaded object-capability system, since it contains two objects, Alice and Contents, that are initially active. This pattern was developed in the context of the single-threaded subset of the object-capability language E. We begin by considering it in this more concurrent context, however, in order to see whether this pattern can be deployed in concurrent object-capability systems, such as operating systems like seL4 or Coyotos.

To test the safety of this pattern in this context, we simply test whether $SafeUnsealer(\mathsf{Unsealer}, \mathsf{Box}, \mathsf{Contents}) \sqsubseteq_T System \setminus \alpha(\mathsf{Slot})$. This test completes in less than 10 seconds in FDR, which reveals that it does not hold. Examining the counter-example, we see that *System* can perform the following trace.

⟨Alice.Unsealer.Call.Contents, Unsealer.SlotWrite.Call.null,
Alice.Box.Call.null, SlotWrite.Unsealer.Return.null,
Box.SlotWrite.Call.Contents, Unsealer.Contents.Call.null,
SlotWrite.Box.Return.null, Contents.Unsealer.Return.null,
Unsealer.SlotRead.Call.null, SlotRead.Unsealer.Return.Contents,
Unsealer.Alice.Return.Contents⟩

Again, we have underlined the cause of the safety violation in the trace. We see Alice Calls the Unsealer, passing Contents as the specimen. Unsealer then clears its slot. At this point, Alice (who recall is executing with her own thread of control, independently of all other objects) Calls Box. Box responds as it should by placing Contents in Slot. Meanwhile, the Unsealer Calls the specimen it has been passed, namely Contents, who Returns immediately. Once this has occurred, the Unsealer then reads the Slot's current value, which is now Contents and subsequently Returns this to Alice as it should.

This violates the safety property because Alice's specimen, Contents, doesn't proxy to Box when Unsealer Calls Contents, but Unsealer still Returns Contents to Alice.

A Less Concurrent Setting

The problem here is readily apparent. The unsealer expects that the objects that it invokes, and the objects that those objects invoke in response to these invocations and so on, have exclusive potential access to its slot and, therefore, that something will be placed in its slot after it has cleared it only in response to its invocation of the specimen it has been passed. This assumption is perfectly valid in the original single-threaded context in which this pattern was developed. However, it is clearly violated in a more concurrent setting, such as an object-capability operating system.

The problem here arises because of concurrent access to shared mutable state, in the form of the slot that is shared between the unsealer and the box. Such vulnerabilities can arise in any concurrent system with shared mutable state, and are certainly not unique to concurrent object-capability

systems. We have shown that FDR can easily detect the vulnerability here as a simple safety violation, which is perhaps unsurprising since FDR has a long history of detecting bugs that arise from concurrent access to shared mutable state. This shows the dangers inherent in taking a pattern from one context and applying it directly to another that is very different [Mur08], but how these dangers can be avoided by first detecting them using FDR.

Having determined that this pattern is not safe in the more concurrent setting, we now consider it in the context in which it was originally designed: a single-threaded object-capability language that allows recursive invocation of objects.

We extend the model, $AnUnsealer(me, slotR, slotW)$, of an unsealer, me, with slot capabilities $slotR$ and $slotW$, to allow for recursive invocation when it invokes a specimen. Doing so gives us the process $AnUnsealerR(me, slotR, slotW, rtstack, maxsz)$ that has the same form as the recursive guard, $AGuardR$, from Snippet 3.5.

$$
\begin{array}{l}
AnUnsealerR(me, slotR, slotW, rtstack, maxsz) = \\
\left(\begin{array}{l}
\#(rtstack) < maxsz \;\&\; \\
\quad ?from : Capability - \{me\}!me!\mathsf{Call}?specimen : Capability \rightarrow \\
\quad me!slotW!\mathsf{Call}!\mathsf{null} \rightarrow slotW!me!\mathsf{Return}!\mathsf{null} \rightarrow \\
\quad (specimen \neq me \;\&\; \\
\quad\; me!specimen!\mathsf{Call}!\mathsf{null} \rightarrow \\
\quad\; AnUnsealerR(me, slotR, slotW, \langle\langle from, specimen\rangle\rangle \frown rtstack, maxsz))
\end{array}\right) \\
\Box \\
\left(\begin{array}{l}
\#(rtstack) > 0 \;\&\; \\
\quad \textbf{let } \langle\langle from, specimen\rangle\rangle \frown st = rtstack \textbf{ within} \\
\quad specimen!me!\mathsf{Return}!\mathsf{null} \rightarrow \\
\quad me!slotR!\mathsf{Call}!\mathsf{null} \rightarrow slotR!me!\mathsf{Return}?val \rightarrow \\
\quad me!from!\mathsf{Return}!val \rightarrow \\
\quad AnUnsealerR(me, slotR, slotW, st, maxsz)
\end{array}\right)
\end{array}
$$

Notice here that the unsealer will Call a given specimen only if that specimen is not the unsealer itself. We impose this restriction to model the natural phenomenon that, in any implementation of this pattern, the message sent to a specimen telling it to place its contents in its slot (often called a "divulge" message) will be recognisably different to the message sent to invoke an unsealer when passing it a specimen (often called an "unseal" message). Any unsealer implementation will therefore typically reject divulge messages, preventing any unsealer from (recursively) Calling itself when passed a specimen capability that refers to itself.

The specification process, $SafeUnsealer(unsealer, bx, conts)$, must also be extended to allow recursive invocation of $unsealer$. Doing so yields the process $SafeUnsealerR(unsealer, bx, conts, rtstack, maxsz)$, in Snippet 3.9.

Despite its apparent complexity, this process is in some ways much simpler than its non-recursively invocable counterpart, $SafeUnsealer$ from Snippet 3.8. In particular, because we're applying it only to single-threaded

$$
\begin{array}{l}
SafeUnsealerR(unsealer, bx, conts, rtstack, maxsz) = \\
\quad \left(\begin{array}{l}
\#(rtstack) < maxsz \ \& \\
\ ?from : otherthan(unsealer)!unsealer!\mathsf{Call}?specimen : Capability \rightarrow \\
\ unsealer!specimen!\mathsf{Call}!\mathsf{null} \rightarrow \\
\ \textbf{let}\ \ rtstack' = \langle\langle from, specimen, specimen = bx\rangle\rangle \hat{}\ rtstack\ \textbf{within} \\
\ SafeUnsealerR(unsealer, bx, conts, rtstack', maxsz)
\end{array}\right) \\
\quad \Box \\
\quad \left(\begin{array}{l}
rtstack = \langle\rangle \ \& \\
\ ?c!bx!\mathsf{Call}?arg \rightarrow SafeUnsealerR(unsealer, bx, conts, rtstack, maxsz)
\end{array}\right) \\
\quad \Box \\
\quad \left(\begin{array}{l}
\#(rtstack) > 0 \ \& \\
\ \textbf{let}\ \langle\langle from, specimen, hasProxied\rangle\rangle \hat{}\ st = rtstack\ \textbf{within} \\
\ ?c!bx!\mathsf{Call}?arg \rightarrow \\
\ \ SafeUnsealerR(unsealer, bx, conts, \langle\langle from, specimen, \mathsf{true}\rangle\rangle \hat{}\ st, maxsz) \\
\ \Box \\
\ specimen!unsealer!\mathsf{Return}!\mathsf{null} \rightarrow \\
\ \textbf{if}\ hasProxied\ \textbf{then} \\
\ \ unsealer!from!\mathsf{Return}\$c : \{conts, \mathsf{null}\} \rightarrow \\
\ \ SafeUnsealerR(unsealer, bx, conts, st, maxsz) \\
\ \textbf{else} \\
\ \ unsealer!from!\mathsf{Return}!\mathsf{null} \rightarrow \\
\ \ SafeUnsealerR(unsealer, bx, conts, st, maxsz)
\end{array}\right)
\end{array}
$$

Snippet 3.9: The specification of a safe recursively invocable unsealer in the single-threaded context.

systems, it doesn't need to keep track of all object invocations in order to determine when *specimen* proxies to *bx*. This is because in a single-threaded system, if *bx* is Called at any time in between when *unsealer* Calls *specimen* and *specimen* subsequently Returns, then *specimen* must have proxied to *bx*. Hence, besides the behaviour of *unsealer*, this specification needs to track only the invocations of *bx*.

This process is otherwise similar to the specification of a recursively invocable guard, *SafeGuardR* from Snippet 3.6. Consider the first of the three main clauses, separated by external choice "□" symbols. If there is room on the stack for another invocation of *unsealer*, it is willing to accept such an invocation from an object *from*, passing a valid capability *specimen*. If such an invocation is received, *unsealer* then of course Calls *specimen* and a new entry is pushed onto the stack which contains *from*, *specimen* and a flag, called *hasProxied*, indicating whether *specimen* has proxied to *bx*. Initially this is true if and only if *specimen* is *bx* as before.

The second main clause is applicable to all states of the system before

unsealer has received any invocations, or at any other time when the stack is empty. In this state, *bx* may be Called by other objects in the system. Hence, the specification allows this.

The third main clause is applicable in all states in which the stack is non-empty. Here, if *bx* is Called, then the current *specimen*, which is sitting on top of the stack, has proxied to *bx* and hence the boolean value *hasProxied* is set to true. Alternatively, the current *specimen* may Return to *unsealer*. In this case, the specification allows *unsealer* to Return *conts* to its Caller, *from*, only if *specimen* has proxied to *bx*, *i.e.* only if *hasProxied* is true.

We re-instantiate the system depicted in Figure 3.2 as a single-threaded object-capability system using the recursively invocable unsealer behaviour for Unsealer. Setting *Object* and *facets* as before, and keeping the same behaviour for Box and Slot, this gives the following new behaviours for the remaining objects in this system.

$$
\begin{aligned}
&\mathit{behaviour}(\mathsf{Unsealer}) = \mathit{AnUnsealerR}(\mathsf{Unsealer}, \mathsf{SlotRead}, \mathsf{SlotWrite}, \langle\rangle, 2),\\
&\mathit{behaviour}(\mathsf{Alice}) =\\
&\quad \mathit{UntrustedActive}_{\mathit{lang}}(\mathit{facets}(\mathsf{Alice}), \mathit{Capability} - \mathit{facets}(\mathsf{Slot}), \mathit{Data}),\\
&\mathit{behaviour}(\mathsf{Contents}) =\\
&\quad \mathit{Untrusted}_{\mathit{lang}}(\mathit{facets}(\mathsf{Contents}), \mathit{Capability} - \mathit{facets}(\mathsf{Slot}), \mathit{Data}).
\end{aligned}
$$

As before with the recursively invocable guard, we allow only a single recursive invocation of Unsealer to occur in order to keep the analysis tractable. We also set Alice to be the object that is initially active. This choice is arbitrary since the object, Alice or Contents, that is initially active can immediately invoke the other. If this occurs, because Alice and Contents possess identical capabilities and data, the system will transition to a state that is identical to the initial state of the same system in which the other object is initially active.

To test whether Unsealer remains safe in this system, we simply test whether

$$
\begin{aligned}
&\mathit{SafeUnsealerR}(\mathsf{Unsealer}, \mathsf{Box}, \mathsf{Contents}, \langle\rangle, 2) \sqsubseteq_T\\
&\quad \mathit{System} \setminus (\Sigma - (\alpha(\mathsf{Unsealer}) - \alpha(\mathit{slot})) \cup \{\!|x.\mathsf{Box.Call} \mid x \in \mathit{Capability}|\!\}).
\end{aligned}
$$

Notice that we hide all events in *System* other than those (1) in the alphabet of Unsealer that do not involve Slot, and (2) those that represent Box being Called, since only these events are relevant to the specification.

Performing this test in FDR reveals that it does *not* hold. FDR returns a counter-example in which *System* performs the following trace.

⟨Alice.Unsealer.Call.Contents, Unsealer.SlotWrite.Call.null,
SlotWrite.Unsealer.Return.null, Unsealer.Contents.Call.null,
Contents.Unsealer.Call.Box, Unsealer.SlotWrite.Call.null,
SlotWrite.Unsealer.Return.null, Unsealer.Box.Call.null,
Box.SlotWrite.Call.Contents, SlotWrite.Box.Return.null,
Box.Unsealer.Return.null, Unsealer.SlotRead.Call.null,
SlotRead.Unsealer.Return.Contents, Unsealer.Contents.Return.Contents,
Contents.Unsealer.Return.null, Unsealer.SlotRead.Call.null,
SlotRead.Unsealer.Return.Contents, Unsealer.Alice.Return.Contents⟩

Let us examine what is going on here. Alice begins by invoking the Unsealer, passing it Contents. Unsealer clears its slot before invoking Contents as it would any other specimen. Contents then recursively Calls Unsealer, to have it unseal Box. Box is subsequently unsealed successfully, as one would expect, with Unsealer then Returning Contents to Contents (the final event on the third-to-last line). Contents then Returns to Unsealer who invoked it originally. At this point, Slot still contains Contents, *i.e.* Contents is left sitting in Slot after the recursive invocation of Unsealer completes. Hence, when Unsealer subsequently reads Slot's contents, it finds Contents and dutifully Returns it to Alice.

This violates the pattern's safety property because Unsealer has Returned Contents to Alice, but Alice's specimen, namely Contents, never proxied to Box. Indeed, while there is a chain of invocations from Contents to Box, this chain involves Unsealer and, as we said earlier, an unsealer cannot proxy by definition. Hence, this counter-example represents a valid vulnerability in this pattern that can arise due to recursive invocation of an unsealer.

This behaviour is certainly possible in a real implementation of this pattern. Indeed, it was first described by David Wagner [Wag08b] in the context of a Cajita implementation of this pattern [MSL+07] on which we have based our model. We purposefully chose this implementation that we knew should exhibit this behaviour to see whether it would be detected by our approach. Despite our purposeful choice, this result nevertheless represents the first time that a vulnerability due to recursive invocation has been automatically detected in an object-capability pattern.

This is significant precisely because vulnerabilities due to recursive invocation are, by their very nature, hard to manually diagnose. Indeed, the implementation that we have modelled here, although created for pedagogical purposes rather than actual deployment, was written by an object-capability languages expert. The vulnerability in that implementation, which was published openly, went undetected for months by all who read the paper [MSL+07] in which it was contained.

Fixing the pattern is straightforward; one just modifies the behaviour of an unsealer to clear its slot once it has read its value. Indeed, had we

$$
\begin{array}{l}
AnUnsealerR(me, slotR, slotW, rtstack, maxsz) = \\
\quad \left(\begin{array}{l}
(\#(rtstack) < maxsz)\ \& \\
\quad ?from : Capability - \{me\}!me!\mathsf{Call}?specimen : Capability \rightarrow \\
\quad me!slotW!\mathsf{Call}!\mathsf{null} \rightarrow slotW!me!\mathsf{Return}!\mathsf{null} \rightarrow \\
\quad (specimen \neq me\ \& \\
\quad\ me!specimen!\mathsf{Call}!\mathsf{null} \rightarrow \\
\quad\ AnUnsealerR(me, slotR, slotW, \langle\langle from, specimen\rangle\rangle \hat{}\ rtstack, maxsz))
\end{array} \right) \\
\quad \Box \\
\quad \left(\begin{array}{l}
(\#(rtstack) > 0)\ \& \\
\quad \textbf{let}\ \langle\langle from, specimen\rangle\rangle \hat{}\ st = rtstack\ \textbf{within} \\
\quad specimen!me!\mathsf{Return}!\mathsf{null} \rightarrow \\
\quad me!slotR!\mathsf{Call}!\mathsf{null} \rightarrow slotR!me!\mathsf{Return}?val \rightarrow \\
\quad me!slotW!\mathsf{Call}!\mathsf{null} \rightarrow slotW!me!\mathsf{Return}!\mathsf{null} \rightarrow \\
\quad me!from!\mathsf{Return}!val \rightarrow \\
\quad AnUnsealerR(me, slotR, slotW, st, maxsz)
\end{array} \right)
\end{array}
$$

Snippet 3.10: The behaviour of a safe recursively invocable unsealer.

modelled the implementation of this pattern from the revised Cajita specification [MSL+08] rather than the outdated original document, this attack would not have been found.

Fixing the Implementation

We redefine the behaviour of an unsealer so that it clears its slot after reading from it, as shown in Snippet 3.10. Reinstantiating the system with this new implementation and repeating the test reveals that it now holds. Having an unsealer clear its slot after reading from it ensures that this pattern remains safe when up to a single recursive invocation of an unsealer can be performed. While we strongly suspect that allowing more recursive invocations would not give rise to any other safety violations, proving this statement is left as future work.

3.3 Related Work

Safety Properties of Object-Capability Systems Safety properties for access control systems (of which object-capability systems are a specific example) are known to be undecidable in the general case [HRU76]. However, for many classes of system, they are known to be both decidable and feasible to compute.

It should be noted that all safety properties (indeed all refinement tests) for our CSP models are decidable in FDR precisely when the CSP processes that represent the specification being tested and the system being analysed

are finite-state. All specifications and systems used in this thesis are finite-state by construction.

The analysis of safety properties in capability systems has a long history beginning with the work of Lipton and Snyder [LS77] on the Take-Grant protection model, in which safety properties are decidable in polynomial time. Variants of this model have been used often since then to analyse how capabilities can propagate (or, perhaps more precisely, be prevented from doing so) in various kinds of object-capability system [SW00, EKE08, Boy09].

Despite their utility for reasoning about the limits of capability propagation in different object-capability *systems*, these models, as argued by Spiessens and Van Roy [SV05], are not well suited for reasoning about the safety properties of different object-capability *patterns*. This is because they include no notion of object behaviour, assuming all objects to be fully untrusted. Objects that implement a pattern (like Guard or Unsealer) are, by their nature, not untrusted. Their behaviour is, by definition, absolutely relevant to the safety properties of the patterns they implement.

Analysing Object-Capability Patterns Spiessens' Scoll language [Spi07] is, to our knowledge, the first formalism for analysing the security properties of object-capability patterns. Scoll allows the behaviour of objects to be explicitly specified and can reason about safety properties. Scoll can also reason about what Spiessens calls *liveness possibilities*, which are discussed further in Section 6.4.

Scoll has the advantage over our approach that the behaviour of objects that implement a pattern can be automatically calculated to satisfy a set of safety properties that the pattern is to enforce. This calculation must be performed manually in our approach, as we have done in this chapter, by manually refining an initial (possibly) unsafe implementation to a safe one with the help of the counter-examples returned from FDR.

Scoll has the disadvantage compared to our approach that it cannot directly model single-threaded systems but must instead conservatively over-approximate them with more concurrent models. This means that it would be difficult to determine the safety of the Sealer-Unsealer implementation that we analysed, whose safety we showed relies on being deployed in a single-threaded system.

To explain, the behaviour of each object in a Scoll system increases *monotonically* over time, meaning that once an action becomes possible it remains possible forever. Hence, an initially inactive object modelled naturally in Scoll would stay active forever once it is first invoked. This would lead to spurious counter-examples being returned, from any Scoll analysis of this pattern in a single-threaded context, that would be difficult or impossible to remove.

Also, expressing the safety property of the Sealer-Unsealer pattern,

which is based on coercion and thus involves detecting proxying, in Scoll would require one to augment the model of the pattern such that the model itself performs the necessary bookkeeping in order to detect when proxying has occurred. This is because Scoll safety properties can assert only that certain individual facts do not become true. One cannot express that certain sequences of actions are forbidden, without augmenting the system model with extra bookkeeping to explicitly track when such sequences occur and to trigger a special fact in this case. In contrast, when using our approach this bookkeeping can be performed by a separate specification process. This has the advantage that the specification can be easily adapted and applied to other coercion patterns.

Unlike our approach, Scoll has yet to be applied to detect vulnerabilities due to the presence of concurrency and recursive invocation.

Analysing Object-Capability Patterns in CSP Ours [Mur08] is the only prior work of which we are aware that has formally analysed a Sealer-Unsealer pattern based on coercion. The analysis performed in [Mur08] detects the vulnerability in this pattern that arises when deployed in the concurrent context. However, the analysis performed in [Mur08] is very limited and ad hoc, considering the pattern only in a very restricted scenario and without recursive invocation. That analysis also failed to formally capture the idea of safe coercion, which we have formalised here. In contrast to [Mur08], the analysis here is far more exhaustive and systematic.

3.4 Conclusion

We have shown how the safety properties of some related object-capability patterns, based on the ideas of authentication and coercion, can be analysed in CSP and automatically checked by expressing them as traces refinement checks that can be carried out in FDR. As argued earlier, these patterns are somewhat akin the low-level cryptographic protocols, since they are the building blocks from which larger security-enforcing patterns are composed. In this sense, they are ideal targets for formal verification.

We saw that one can express complicated safety properties, such as safe coercion which requires detecting when one object has proxied to another, in the form of traces refinement checks. This is done by encoding those properties as specification processes, making use of CSP's natural expressiveness. It is unclear to what degree these kinds of properties could be stated and checked using previous formalisms for analysing object-capability patterns. Because we could accurately encode these safety properties in CSP, we were able to automatically detect subtle vulnerabilities in these patterns that arise due to concurrency and recursive invocation. Both kinds of vulnerability are difficult to diagnose manually; hence, our technique has undoubted value.

We saw that CSP's expressiveness also allows us to consider the same pattern in different contexts, such as single-threaded vs. concurrent systems, and to compare how its security properties differ in each case. The Sealer-Unsealer example from Section 3.2, in which we saw that this pattern cannot be deployed safely in the concurrent context, demonstrates that this kind of comparison is vital in order to avoid deploying patterns in unsafe contexts.

There are three primary limitations to the work presented in this chapter. Firstly, we have analysed only small instantiations of each pattern that comprise no more than a handful of objects. Hence, it is unclear (and we certainly have no formal grounds yet on which to argue) whether the results we have obtained here can generalise to arbitrary sized systems in which these patterns might be deployed. Secondly, our system models do not account for the ability of objects, in real object-capability systems, to create new objects. We have completely ignored the issue of object creation. Thirdly, our analyses of patterns involving recursive invocation allow the objects that implement a pattern to be recursively invoked only once. In the following chapter, we show how to overcome the first two limitations. Overcoming the third is left as future work.

4 Analysing Systems of Arbitrary Size

The work presented in the previous chapter has two primary limitations. The first is that the results obtained there apply only to the small systems that were analysed, which contain no more than a handful of objects. The second is that this work completely ignored the issue of object creation. In this chapter, we show in turn how each of these limitations can be overcome.

In Section 4.1 we show how the results obtained in the previous chapter can be generalised to systems of arbitrary size. Roughly, we show that any arbitrary-sized system can be safely abstracted by a small fixed-size system by *aggregating* [Spi07] multiple objects in the arbitrary-sized system together and representing them as a single object in the small fixed-sized system. We then apply the theory of *data-independence* [Laz99] to generalise the analysis of the small fixed-sized system to all systems of arbitrary size that it safely abstracts.

Then, in Section 4.2, we argue that these same techniques can be used to model patterns and systems in which objects can create arbitrary numbers of other objects. The basic idea is to model a parent object and all children it might create as a single object that aggregates the parent and all of its children together [Spi07]. We demonstrate this technique in Section 4.3 by analysing an implementation of the revocable Membrane pattern, which uses object creation as part of its normal functioning. We find that this implementation upholds a slightly weaker revocation property in the concurrent context when compared with the single-threaded context[1].

The ideas behind much of this chapter's contents were heavily influenced by similar work of Spiessens [Spi07].

4.1 Generalising Previous Results

Recall Section 3.1, in which we derived a safe Trademarks implementation. These results were obtained be analysing the pattern instantiated in the system depicted in Figure 3.1. This system contains just one untrusted object, Specimen, that has capabilities to a stamped object, Stamped, and its

[1] While similar analyses have been performed previously (*e.g.* [Spi07, Section 8.3.1]), ours is the first to explicitly model and reason about the membrane's revocation property, which cannot be expressed directly using these previous formalisms.

corresponding guard, Guard. Our approach to analysing object-capability patterns in CSP has limited value unless we can *generalise* the results obtained in that section to the larger systems in which this pattern might be deployed.

Intuitively, one might hope to be able to generalise the results obtained for the Trademarks pattern to any system that includes (implementations of) all of the non-*Untrusted* objects in Figure 3.1 (*i.e.* Guard, Slot and Stamped) composed with arbitrary objects, each of which has no capability to a non-*Untrusted* object that Specimen doesn't have. In other words, one might hope that these results can be generalised to any system of the form depicted in Figure 4.1.

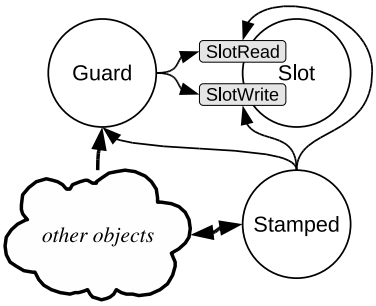

Figure 4.1: Generalising the results of the Trademarks safety analysis.

In Figure 4.1, the bold arrows from the cloud labelled "*other objects*" indicate that there could be multiple objects in this cloud that have capabilities to Guard and/or Stamped. The bold arrow from Stamped to the cloud represents Stamped possessing capabilities to some subset of objects in the cloud. The objects within the cloud can have any behaviour permitted in our model and can be interconnected in any way at all. We will generalise the results obtained earlier to all such systems.

Consider the (infinite) set of finite systems represented by the depiction in Figure 4.1. Let $\mathcal{I}$ denote this set of systems. In order to generalise our analysis results to all systems in $\mathcal{I}$, we must find a finite collection, $\mathcal{C}$, of finite systems, each of which FDR can analyse, such that if the analysis holds for all systems in $\mathcal{C}$, it holds for all systems in $\mathcal{I}$.

We use a two-step approach to achieve this. We begin by showing that any particular finite system, $System \in \mathcal{I}$, represented by the depiction in Figure 4.1, can be safely captured by a finite abstraction, $System'$, that has the form of the system depicted in Figure 3.1, but where the cloud is replaced by a single untrusted object, Specimen, which is given all of the facets and all of the capabilities of all of the objects it replaces. Borrowing from Spiessens [Spi07], we call this idea, by which a collection of objects is replaced by a single one, *aggregation*. We also say that $System'$ is a *safe abstraction* of *System*. We then use the theory of data-independence to show that it suffices to analyse just a finite set $\mathcal{C}$ of finite $System'$ in order

to obtain results that apply to all possible finite $System \in \mathcal{I}$.

Recall that the basic ideas of data-independence were introduced in Section 2.3.6 and, as argued there, Theorem 2.3.5 has been (implicitly) applied throughout this thesis so far to allow our results, for patterns and properties that are indifferent to data, to generalise over all choices for the set *Data*.

4.1.1 Safe Abstraction and Aggregation

We begin by formalising the first step of this process, which is using aggregation to build safe abstractions of systems. Given some finite system, *System*, that we wish our results to generalise to, we define what it means for some other finite system, $System'$, to be a *safe abstraction* of *System*. This occurs precisely when any analysis result in which we are interested that holds for $System'$ also holds for *System*.

Recall, from Section 2.1.4, that the safety properties examined in the previous chapter are *refinement-closed* (see Definition 2.1.2) in the traces model, meaning that each holds for *System* if and only if it holds for all of *System*'s traces refinements. Recall that one system is a traces refinement of another, when all sequences of visible events that can occur in the first can occur in the second. (In our case modelling object-capability systems, one system is a traces refinement of another when every sequence of message exchanges that can occur in the first can occur in the second.) We therefore insist that in order for $System'$ to be a safe abstraction of *System* with respect to these properties, that the traces refinements of $System'$ include all traces refinements of *System*. This occurs when $System'$ is an *anti-traces-refinement* of *System* (since refinement is transitive), *i.e.* when $System' \sqsubseteq_T System$. This argument can be generalised from the traces model to properties that are refinement-closed in any CSP model $\mathcal{M}$. Letting $\mathcal{M}$ denote an arbitrary CSP model, recall that we write $P \sqsubseteq_{\mathcal{M}} Q$ to mean that Q is a refinement of P in the model $\mathcal{M}$.

Definition 4.1.1 (Safe Abstraction wrt Refinement-Closed Properties)**.** One process P is a *safe abstraction* of another process Q with respect to properties that are refinement-closed in some model $\mathcal{M}$ iff $P \sqsubseteq_{\mathcal{M}} Q$.

We now formally define the process by which a system, $System \in \mathcal{I}$, might be abstracted by a system, $System'$, in which a collection K of objects in *System* is replaced by a single object o in $System'$. Borrowing Spiessens' terminology, we say that o *aggregates* all of the objects in K. o must be able to exhibit every behaviour, when run in $System'$, that can be exhibited by the collection K of objects it aggregates in *System*. This implies that o must expose every facet, and possess every capability and datum, of every object in K.

We formalise this idea by defining a surjective function, *Abs*, that maps object names in *System* to corresponding object names in $System'$. Each

object o' in $System'$ aggregates the set of objects in $System$ that are mapped to it under Abs, *i.e.* $Abs^{-1}(o')$. Aggregation is formally defined as follows.

Definition 4.1.2 (Aggregation)**.** Let $(Object, behaviour, facets, Data)$ be an object-capability system captured by the CSP process $System = \|_{o \in Object}(behaviour(o), \alpha(o))$. Then another object-capability system, $(Object', behaviour', facets', Data)$, with identical data and captured by the CSP process $System' = \|_{o \in Object'}(behaviour'(o), \alpha'(o))$, is an *aggregation* of the first when there exists some surjection $Abs : Object \to Object'$ such that for all $o' \in Object'$, $facets'(o') = \bigcup\{facets(o) \mid o \in Abs^{-1}(o')\}$ and

$$\begin{aligned} &\forall\, s \in traces(System) \bullet \forall\, X \in \mathbf{P}\,\Sigma \bullet \\ &\quad (s \upharpoonright \alpha'(o'), X) \in failures(\|_{o \in Abs^{-1}(o')}(behaviour(o), \alpha(o))) \Rightarrow \\ &\qquad (s \upharpoonright \alpha'(o'), X) \in failures(behaviour'(o')). \end{aligned} \tag{4.1}$$

Note that from Definition 2.3.1, this definition implies that for each $o' \in Object'$, $\alpha'(o') = \bigcup\{\alpha(o) \mid o \in Abs^{-1}(o')\}$.

It is easily shown that all aggregations are safe abstractions with respect to refinement-closed properties. We show this with respect to properties that are refinement-closed in the failures-divergences model.

Theorem 4.1.3. Let $(Object, behaviour, facets, Data)$ and $(Object', behaviour', facets', Data)$ be two object-capability systems with identical data captured by the CSP processes $System = \|_{o \in Object}(behaviour(o), \alpha(o))$ and $System' = \|_{o \in Object'}(behaviour'(o), \alpha'(o))$ respectively, such that $System'$ is an aggregation of $System$. Then $System' \sqsubseteq System$.

Proof. Suppose the conditions of the lemma. From Definition 2.3.1, both $System$ and $System'$ must be divergence-free. Hence, we need to show only that $failures(System) \subseteq failures(System')$. Consider some failure $(s, X) \in failures(System)$ then. We show that it is also present in $failures(System')$.

$$\begin{aligned} &(s, X) \in failures(System) \\ \Leftrightarrow\ &\{\text{associativity}\} \\ &(s, X) \in failures(\|_{o' \in Object'} \left(\|_{o \in Abs^{-1}(o')}(behaviour(o), \alpha(o)), \alpha'(o')\right)) \\ \Leftrightarrow\ &\{\text{alphabetised parallel composition}\} \\ &\exists\{X_{o'} \mid o' \in Object'\} \bullet \forall\, o' \in Object' \bullet \\ &\quad (s \upharpoonright \alpha'(o'), X_{o'}) \in failures(\|_{o \in Abs^{-1}(o')}(behaviour(o), \alpha(o))) \wedge \\ &\quad \bigcup_{o' \in Object'} X_{o'} \cap \alpha'(o') = X \cap \bigcup_{o' \in Object'} \alpha'(o') \\ \Rightarrow\ &\{\text{assumption}\} \\ &\exists\{X_{o'} \mid o' \in Object'\} \bullet \forall\, o' \in Object' \bullet \\ &\quad (s \upharpoonright \alpha'(o'), X_{o'}) \in failures(behaviour'(o')) \wedge \\ &\quad \bigcup_{o' \in Object'} X_{o'} \cap \alpha'(o') = X \cap \bigcup_{o' \in Object'} \alpha'(o') \\ \Leftrightarrow\ &\{\text{alphabetised parallel composition}\} \\ &(s, X) \in failures(System') \end{aligned}$$

□

Note that this result implies that all aggregations are also safe abstractions with respect properties that are refinement-closed in the traces or stable-failures models as well since, in the proof above, *System* is necessarily also a traces and stable-failures refinement of *System′* because *System′* is divergence-free.

This result implies that all behaviours (recorded in the failures-divergences model) that can be observed of a system, can also be observed in any valid aggregation of that system. However, this does not prevent extra behaviours from being exhibited in the aggregation that cannot be exhibited in the original system. In this sense, an aggregation is usually (what Spiessens calls) an *over approximation* of the system that it aggregates, because it may exhibit behaviours that cannot be observed of the original system and so properties might fail to hold for the aggregation that otherwise hold for the system being aggregated. However, we can be sure that no property that is refinement-closed in the traces, stable-failures or failures-divergences model can ever hold for an aggregation but fail to hold for a system that it aggregates.

4.1.2 Aggregation via Untrusted Objects

We now prove some results that show how untrusted objects can be used to build aggregations (and, hence, safe abstractions) of systems by replacing a collection of objects by a single untrusted one. We first consider the general case and then consider the single-threaded case.

We first show that in any object-capability system, any finite collection, U, of objects can be replaced by a single $Untrusted_{OS}$ object that has all of U's facets, capabilities and data, yielding another object-capability system that is an aggregation of the first. We then prove the analogues of this result for the single-threaded case: that in a single-threaded object-capability system, (1) any finite collection U of initially inactive objects can be aggregated by a single $Untrusted_{lang}$ object that has all of their facets, capabilities and data; and (2) any finite collection U of objects that includes the initially active object, can be aggregated by a single $UntrustedActive_{lang}$ object that has all of their facets, capabilities and data. In either single-threaded case, the aggregation yields another single-threaded system.

For each of these results, it is enough to show that the result holds when U has size 2. This result can then be extended to all finite non-empty sets U by induction.

The General Case

Theorem 4.1.4. Let $(Object, behaviour, facets, Data)$ and $(Object', behaviour', facets', Data)$ be two object-capability systems with identical

data captured by the CSP processes $System = \|_{o \in Object}(behaviour(o), \alpha(o))$ and $System' = \|_{o \in Object'}(behaviour'(o), \alpha'(o))$ respectively, such that $Object$ has size at least 2 and $|Object'| = |Object| - 1$. Let o_1 and o_2 be elements of $Object$ and $Abs : Object \to Object'$ be a surjection that maps each element of $Object$ to a unique element of $Object'$, except o_1 and o_2 which are mapped to the same element p', such that:

$$\forall o' \in Object' - \{p'\} \bullet behaviour'(o') = behaviour(o) \wedge facets'(o') = facets(o), \text{where } Abs^{-1}(o') = \{o\}; \text{ and}$$

$$behaviour'(p') = Untrusted_{OS}(facets'(p'), caps(o_1) \cup caps(o_2), data(o_1) \cup data(o_2)),$$

where

$$facets'(p') = facets(o_1) \cup facets(o_2)$$

and *caps* and *data* are the functions that give the minimal set of initial capabilities and data respectively for each object $o \in Object$ (see Definition 2.3.1). Then $System'$ is an aggregation of $System$.

Proof. Suppose the conditions of the theorem and consider some $o' \in Object'$. Clearly, $facets'(o') = \bigcup\{facets(o) \mid o \in Abs^{-1}(o')\}$. We prove Equation 4.1 is satisfied.

If $o' \neq p'$ then, letting p be the unique element of $Object$ such that $Abs(p) = o'$, then $\|_{o \in Abs^{-1}(o')}(behaviour(o), \alpha(o)) \equiv_{FD} behaviour(p)$. Hence, Equation 4.1 is trivially satisfied.

Otherwise, $o' = p'$. Let $P_{p'} = \|_{o \in Abs^{-1}(p')}(behaviour(o), \alpha(o))$, then

$$P_{p'} = behaviour(o_1) \;{}_{\alpha(o_1)}\|_{\alpha(o_2)}\; behaviour(o_2).$$

Similarly, let $Q_{p'}$ be defined as

$$Q_{p'} = Untrusted_{OS}(facets(o_1), caps(o_1), data(o_1)) \;{}_{\alpha(o_1)}\|_{\alpha(o_2)}\; Untrusted_{OS}(facets(o_2), caps(o_2), data(o_2)).$$

Then, by Definition 2.3.1, $Q_{p'} \sqsubseteq_{FD} P_{p'}$. Consider an arbitrary trace $s \in traces(System)$ and an arbitrary failure $(s \upharpoonright \alpha'(p'), X) \in failures(Q_{p'})$. It suffices to show that $(s \upharpoonright \alpha'(p'), X)$ is a failure of $behaviour'(p') = Untrusted_{OS}(facets'(o'), caps(o_1) \cup caps(o_2), data(o_1) \cup data(o_2))$.

Because $behaviour(p')$ and $Q_{p'}$ are both divergence-free (by Definition 2.3.1) and $behaviour(p')$ can deadlock at any time, it is enough to show that $s \upharpoonright \alpha'(p') \in traces(behaviour'(p'))$. We do so by induction on the length of $s \upharpoonright \alpha'(p')$.

The base case holds trivially since $\langle\rangle$ is a trace of every process.

For the inductive case, the induction hypothesis says that for all $s \in traces(System)$, $\#(s \upharpoonright \alpha'(p')) = n \Rightarrow s \upharpoonright \alpha'(p') \in traces(behaviour'(p'))$.

Consider some trace $s \in traces(System)$ such that $\#(s \upharpoonright \alpha'(p')) = n+1$. We show that $s \upharpoonright \alpha'(p') \in traces(behaviour'(p'))$.

Let $t^\frown\langle e\rangle = s \upharpoonright \alpha'(p')$. Then t has length n, so by the induction hypothesis, $t \in traces(behaviour'(p'))$. We must show that the e can follow, after this process has performed t.

$e \in \alpha(o_1) \cup \alpha(o_2)$. So let $i \in \{1, 2\}$ be such that $e \in \alpha(o_i)$. Then

$$s \upharpoonright \alpha'(p') \upharpoonright \alpha(o_i)^\frown\langle e\rangle \in traces(behaviour(o_i)).$$

By Lemma A.0.1 (from Appendix A) and Definition 2.3.1,

$$Untrusted_{OS}(facets'(p'), caps(o_1) \cup caps(o_2), data(o_1) \cup data(o_2)) \sqsubseteq_{FD} \\ Untrusted_{OS}(facets(o_i), caps(o_i), data(o_i)) \sqsubseteq_{FD} behaviour(o_i).$$

Hence,

$$s \upharpoonright \alpha'(p') \upharpoonright \alpha(o_i)^\frown\langle e\rangle \in traces(behaviour'(p')).$$

By Lemma A.0.2 (from Appendix A), $s \upharpoonright \alpha'(p')^\frown\langle e\rangle$ must also be a trace of $behaviour'(p')$. □

The Single-Threaded Case

We now prove the analogues of Theorem 4.1.4 for the single-threaded case.

Theorem 4.1.5. Let $(Object, behaviour, facets, Data)$ and $(Object', behaviour', facets', Data)$ be two single-threaded object-capability systems with identical data, captured by the CSP processes $System = \|_{o \in Object}(behaviour(o), \alpha(o))$ and $System' = \|_{o \in Object'}(behaviour'(o), \alpha'(o))$ respectively, such that $Object$ has size at least 3 and $|Object'| = |Object| - 1$.[2] Let o_1 and o_2 be members of $Object$ that are initially inactive in $System$. Let $Abs : Object \to Object'$ be a surjection that maps each element of $Object$ to a unique element of $Object'$ except for o_1 and o_2 which are both mapped to the same element p', such that:

$$\forall o' \in Object' - \{p'\} \bullet behaviour'(o') = behaviour(o) \wedge \\ facets'(o') = facets(o), \text{where } Abs^{-1}(o') = \{o\}; \text{ and}$$

$$behaviour'(p') = \\ Untrusted_{lang}(facets'(p'), caps(o_1) \cup caps(o_2), data(o_1) \cup data(o_2)),$$

where $facets'(p') = facets(o_1) \cup facets(o_2)$, and $caps$ and $data$ are the functions that give the minimal set of initial capabilities and data respectively for each object $o \in Object$ (see Definition 2.3.2).

Then $System'$ is an aggregation of $System$.

[2] We require $Object$ to have size at least 3 because it must contain at least 2 initially inactive objects to be aggregated in addition to the single object that is initially active.

Proof. Suppose the conditions of the lemma. Following the proof of Theorem 4.1.4, let $Q_{p'} = Untrusted_{lang}(facets(o_1), caps(o_1), data(o_1)) \ {}_{\alpha(o_1)}\|_{\alpha(o_2)} \ Untrusted_{lang}(facets(o_2), caps(o_2), data(o_2))$. Then, by Definition 2.3.2 and Lemma A.0.1 (from Appendix A), it is sufficient to show that for every trace $s \in traces(System)$, if $s \upharpoonright \alpha'(p')$ is a trace of $Q_{p'}$, then it is also a trace of $behaviour'(p')$.

We do so by induction on the length of the trace $s \upharpoonright \alpha'(p')$. To complete this inductive proof, we strengthen the claim to also require that, after $behaviour(p')$ has performed some trace $s \upharpoonright \alpha'(p')$, p' is active if and only if o_1 is active or o_2 is active after $Q_{p'}$ has performed the same trace.

The proof by induction then follows the same structure as in the previous proof. For the inductive case, we must show that $behaviour'(p')$ can perform some event e after performing a trace t, as before. There are two cases to consider: $e \in \{\!|f_1, f_2 \mid f_1 \in facets(o_1), f_2 \in facets(o_2)|\!\}$ or not. The first case holds straightforwardly, applying Lemma A.0.3 (from Appendix A) in a similar manner to the application of Lemma A.0.2 in the previous proof.

In the second case, $e \notin \{\!|f_1, f_2 \mid f_1 \in facets(o_1), f_2 \in facets(o_2)|\!\}$, e represents one of the objects, o_1 or o_2, receiving a message from some other object that is not o_1 or o_2. From Lemma 2.3.4, both o_1 and o_2 must be inactive after $Q_{p'}$ performs t and, from the induction hypothesis, so must p'. Applying Lemma A.0.3 again, we can show that $behaviour(p')$ can perform e after t, since one of the components of $Q_{p'}$ can. Once e has been performed, whichever object (o_1 or o_2) was involved in performing e becomes active, as does p'. Hence, the claim remains true as required. □

Theorem 4.1.6. Let $(Object, behaviour, facets, Data)$ and $(Object', behaviour', facets', Data)$ be two single-threaded object-capability systems with identical data, captured by the CSP processes $System = \|\|_{o \in Object}(behaviour(o), \alpha(o))$ and $System' = \|\|_{o \in Object'}(behaviour'(o), \alpha'(o))$ respectively, such that $Object$ has size at least 2 and $|Object'| = |Object| - 1$. Let $o_1 \in Object$ be the object that is initially active in $System$ and $o_2 \in Object$ be another arbitrary object. Let $Abs : Object \to Object'$ be a surjection that maps each element of $Object$ to a unique element of $Object'$ except for o_1 and o_2 which are both mapped to the same element p', such that:

$$\forall o' \in Object' - \{p'\} \bullet behaviour'(o') = behaviour(o) \wedge facets'(o') = facets(o), \text{where } Abs^{-1}(o') = \{o\}; \text{ and}$$

$$behaviour'(p') = UntrustedActive_{lang}(facets'(p'), caps(o_1) \cup caps(o_2), data(o_1) \cup data(o_2)),$$

where $facets'(p') = facets(o_1) \cup facets(o_2)$ and $caps$ and $data$ are the functions that give the minimal set of initial capabilities and data respectively for each $o \in Object$ (see Definition 2.3.2).

Then $System'$ is an aggregation of $System$.

Proof. This proof is a straightforward adaptation of that for Theorem 4.1.5. □

4.1.3 Data-Independence on Aggregated Identities

Theorem 4.1.4 proves that any (finite) system represented by Figure 4.1, is safely abstracted by the system that has the form of Figure 3.1, in which Specimen has all of the facets, capabilities and data of each of the objects that it aggregates.

Running with this example for the remainder of this subsection, let U be the set of objects that make up the cloud in Figure 4.1 and let $T = \bigcup\{facets(u) \mid u \in U\}$. Then each (finite) system represented by this figure corresponds to a different (finite) value for T. To verify that this pattern is safe in all such systems, we can check that the pattern is safe in all systems $System_T$ that have the form of Figure 3.1, where $facets(\mathsf{Specimen}) = T$ for each. In other words, we need to verify the safety of $System_T$ for all non-empty choices for the set T.

We do so by treating T as a type in which (as we will show) $System_T$ is data-independent (see Section 2.3.6). We can then apply data-independence theory to derive a data-independence *threshold* for T that allows us to verify $System_T$ for all choices for T by checking $System_T$ for just a few concrete choices for T whose size are less than or equal to the threshold.

We must first show that $System_T$ is, indeed, data-independent in $T = facets(\mathsf{Specimen})$. We begin by writing this system, and all components of it, so as to be parameterised by the set T, giving the process $System_T$.

$$System_T = \|_{o \in Object}(behaviour_T(o), \alpha_T(o)),$$

where for all $o \in Object = \{\mathsf{Specimen}, \mathsf{Stamped}, \mathsf{Slot}, \mathsf{Guard}\}$,

$$\begin{aligned}\alpha_T(o) = \{&f.c.op.arg,\ c.f.op.arg \mid \\ &f \in facets_T(o), op \in \{\mathsf{Call}, \mathsf{Return}\}, \\ &c \in T \cup \{\mathsf{SlotRead}, \mathsf{SlotWrite}, \mathsf{Guard}, \mathsf{Stamped}\}, \\ &arg \in T \cup \{\mathsf{SlotRead}, \mathsf{SlotWrite}, \mathsf{Guard}, \mathsf{Stamped}, \mathsf{null}\} \cup Data\};\end{aligned}$$

$facets_T(\mathsf{Specimen}) = T$ and $facets_T(o) = facets(o)$ as before otherwise.

We then define $behaviour_T$ for each of the objects in the system as shown in Snippets 4.1 – 4.2. We can see from inspection that each of the behaviours and, hence, the entire system is data-independent in T. Writing the safe Trademarks specification, which recall is the process $SafeGuard(\mathsf{Guard}, \{\mathsf{Stamped}\})$ from Snippet 3.3, in terms of T gives the process $Spec_T$, which is also clearly data-independent in T and appears in Snippet 4.3.

Letting $SafeGuardEvents_T = \{|from.\mathsf{Guard.Call}, \mathsf{Guard}.from.\mathsf{Return} \mid from \in T \cup \{\mathsf{Stamped}, \mathsf{SlotRead}, \mathsf{SlotWrite}\}|\}$, we want to show for all non-empty T that

$$Spec_T \sqsubseteq_T System_T \setminus (\Sigma - SafeGuardEvents_T).$$

$$
\begin{array}{l}
behaviour_T(\mathsf{Guard}) = \\
\quad \textbf{let} \\
\quad\quad AllFroms = T \cup \{\mathsf{Stamped}, \mathsf{SlotRead}, \mathsf{SlotWrite}\}, \\
\quad\quad AllCaps = T \cup \{\mathsf{Stamped}, \mathsf{SlotRead}, \mathsf{SlotWrite}, \mathsf{Guard}\}, \\
\quad\quad AllVals = AllCaps \cup Data \cup \{\mathsf{null}\} \\
\quad \textbf{within} \\
\quad ?from : AllFroms!\mathsf{Guard}!\mathsf{Call}?specimen : AllCaps \rightarrow \\
\quad \mathsf{Guard}!\mathsf{SlotWrite}!\mathsf{Call}!\mathsf{null} \rightarrow \mathsf{SlotWrite}!\mathsf{Guard}!\mathsf{Return}!\mathsf{null} \rightarrow \\
\quad \mathsf{Guard}!specimen!\mathsf{Call}!\mathsf{null} \rightarrow specimen!\mathsf{Guard}!\mathsf{Return}!\mathsf{null} \rightarrow \\
\quad \mathsf{Guard}!\mathsf{SlotRead}!\mathsf{Call}!\mathsf{null} \rightarrow \mathsf{SlotRead}!\mathsf{Guard}!\mathsf{Return}?val : AllVals \rightarrow \\
\quad \textbf{if } val = specimen \textbf{ then} \\
\quad\quad \mathsf{Guard}!from!\mathsf{Return}!\mathsf{Guard} \rightarrow behaviour_T(\mathsf{Guard}) \\
\quad \textbf{else } \mathsf{Guard}!from!\mathsf{Return}!\mathsf{null} \rightarrow behaviour_T(\mathsf{Guard})
\end{array}
$$

$$
\begin{array}{l}
behaviour_T(\mathsf{Slot}) = \\
\quad \textbf{let } S_T(val) = \\
\quad\quad \textbf{let} \\
\quad\quad\quad AllFroms = T \cup \{\mathsf{Stamped}, \mathsf{Guard}\}, \\
\quad\quad\quad AllVals = T \cup \{\mathsf{Stamped}, \mathsf{SlotRead}, \mathsf{SlotWrite}, \mathsf{Guard}, \mathsf{null}\} \cup Data, \\
\quad\quad \textbf{within} \\
\quad\quad ?from : AllFroms!\mathsf{SlotRead}!\mathsf{Call}!\mathsf{null} \rightarrow \\
\quad\quad\quad \mathsf{SlotRead}!from!\mathsf{Return}!val \rightarrow S_T(val) \ \Box \\
\quad\quad ?from : AllFroms!\mathsf{SlotWrite}!\mathsf{Call}?newVal : AllVals \rightarrow \\
\quad\quad\quad \mathsf{SlotWrite}!from!\mathsf{Return}!\mathsf{null} \rightarrow S_T(newVal) \\
\quad \textbf{within } S_T(\mathsf{null})
\end{array}
$$

$$
\begin{array}{l}
behaviour_T(\mathsf{Stamped}) = \\
\quad \textbf{let} \\
\quad\quad AllFroms = T \cup \{\mathsf{SlotRead}, \mathsf{SlotWrite}, \mathsf{Guard}\}, \\
\quad\quad AllTos = T \cup \{\mathsf{SlotRead}, \mathsf{Guard}\}, \\
\quad\quad AllVals = T \cup \{\mathsf{Stamped}, \mathsf{SlotRead}, \mathsf{SlotWrite}, \mathsf{Guard}, \mathsf{null}\} \cup Data, \\
\quad\quad AllArgs = T \cup \{\mathsf{Stamped}, \mathsf{SlotRead}, \mathsf{Guard}, \mathsf{null}\} \cup Data, \\
\quad \textbf{within} \\
\quad \left(\begin{array}{l}
\mathsf{Stamped}?to : AllTos?op : \{\mathsf{Call}, \mathsf{Return}\}?arg : AllArgs \rightarrow \\
\quad\quad behaviour_T(\mathsf{Stamped}) \ \Box \\
\mathsf{Stamped}!\mathsf{Stamped}?op : \{\mathsf{Call}, \mathsf{Return}\}?arg : AllArgs \cup \{\mathsf{SlotWrite}\} \rightarrow \\
\quad\quad behaviour_T(\mathsf{Stamped}) \ \Box \\
\mathsf{Stamped}!\mathsf{SlotWrite}!\mathsf{Call}!\mathsf{Stamped} \rightarrow behaviour_T(\mathsf{Stamped}) \\
?from : AllFroms!\mathsf{Stamped}?op : \{\mathsf{Call}, \mathsf{Return}\}?arg : AllVals \rightarrow \\
\quad\quad behaviour_T(\mathsf{Stamped})
\end{array} \right) \\
\quad \sqcap STOP
\end{array}
$$

Snippet 4.1: Object behaviours in terms of $T = facets(\mathsf{Specimen})$.

$$
\begin{array}{l}
behaviour_T(\mathsf{Specimen}) = \\
\quad \textbf{let}\ \ P(caps) = \\
\qquad \textbf{let} \\
\qquad\quad AllFroms = \{\mathsf{Guard}, \mathsf{Stamped}, \mathsf{SlotRead}, \mathsf{SlotWrite}\}, \\
\qquad\quad AllVals = T \cup \{\mathsf{Stamped}, \mathsf{SlotRead}, \mathsf{SlotWrite}, \mathsf{Guard}, \mathsf{null}\} \cup Data, \\
\qquad\quad AllArgs = T \cup caps \cup \{\mathsf{null}\} \cup Data, \\
\qquad\quad AllTos = T \cup caps, \\
\qquad\quad AllCaps = T \cup \{\mathsf{Stamped}, \mathsf{SlotRead}, \mathsf{SlotWrite}, \mathsf{Guard}\} \\
\qquad \textbf{within} \\
\qquad \left(\begin{array}{l} ?from : T?to : AllTos?op : \{\mathsf{Call}, \mathsf{Return}\}?arg : AllArgs \rightarrow \\ \quad P(caps) \ \Box \\ ?from : AllFroms?to : T?op?arg : AllVals \rightarrow \\ \quad P(caps \cup (\{arg, from\} \cap AllCaps)) \end{array}\right) \\
\qquad \sqcap STOP \\
\quad \textbf{within}\ \ P(T \cup \{\mathsf{Guard}, \mathsf{Stamped}\})
\end{array}
$$

Snippet 4.2: Specimen's behaviour in terms of $T = facets(\mathsf{Specimen})$.

We will use standard data-independence results to argue that 2 is a sufficient data-independence threshold for T to show this result.

$$
\begin{array}{l}
Spec_T = \\
\quad \textbf{let} \\
\qquad AllFroms = T \cup \{\mathsf{Stamped}, \mathsf{SlotRead}, \mathsf{SlotWrite}\}, \\
\qquad AllCaps = T \cup \{\mathsf{Stamped}, \mathsf{SlotRead}, \mathsf{SlotWrite}, \mathsf{Guard}\} \\
\quad \textbf{within} \\
\quad ?from : AllFroms!\mathsf{Guard}!\mathsf{Call}?specimen : AllCaps \rightarrow \\
\qquad \textbf{if}\ specimen = \mathsf{Stamped}\ \textbf{then} \\
\qquad\quad \left(\begin{array}{l} \mathsf{Guard}!from!\mathsf{Return}!\mathsf{Guard} \rightarrow Spec_T \ \sqcap \\ \mathsf{Guard}!from!\mathsf{Return}!\mathsf{null} \rightarrow Spec_T \end{array}\right) \\
\qquad \textbf{else}\ \ \mathsf{Guard}!from!\mathsf{Return}!\mathsf{null} \rightarrow Spec_T
\end{array}
$$

Snippet 4.3: Safe Trademarks spec in terms of $T = facets(\mathsf{Specimen})$.

Applying Data-Independence Theorems to our System

Recall that data-independence theorems distinguish between different properties of a system, such as whether its operational semantics contains no equality tests between members of T. Recall that in this case, we say that the system satisfies the property $\mathbf{NoEqT}_T$. $System_T$ clearly does not satisfy $\mathbf{NoEqT}_T$ because $behaviour_T(\mathsf{Guard})$, which appears in Snippet 4.1,

contains the explicit equality test "*val* = *specimen*" in which the values being compared might both be from T.

For processes that contain equality tests, data-independence theorems distinguish between two kinds of processes. Roughly speaking, a process is said to satisfy the condition $\mathbf{PosConjEqT}_T$ if it becomes $STOP$ whenever an equality test between members of T fails. (This definition is good enough for our purposes. A more precise formulation appears in [Laz99].) $System_T$ doesn't satisfy $\mathbf{PosConjEqT}_T$ because $behaviour_T(\mathsf{Guard})$ does not.

The relevant data-independence theorem from [Ros97][3], for systems that satisfy neither $\mathbf{NoEqT}_T$ nor $\mathbf{PosConjEqT}_T$ that is applicable to $System_T$ and $Spec_T$, can be applied to show that 8 is a sufficient data-independence threshold for T. This theorem implies that we can verify the Trademarks pattern for all systems captured by Figure 4.1 by performing 7 relatively cheap refinement tests in FDR, on top of those already performed in the previous chapter.

In general, however, this approach doesn't scale very well. Increasing the data-independence threshold by 1 roughly doubles the complexity of the associated refinement check. This has significant consequences later on when we further generalise the analysis performed here to systems containing an arbitrary number of stamped objects. Doing so effectively doubles the data-independence threshold. Hence, it is in our interest to find a means by which the lowest data-independence threshold can be used here.

For systems that satisfy $\mathbf{NoEqT}_T$ or $\mathbf{PosConjEqT}_T$, one can usually find data-independence thresholds of no more than 2 for traces refinement checks. Whilst $System_T$ doesn't satisfy $\mathbf{PosConjEqT}_T$, we can build a safe abstraction of it that does. This then enables us to derive a data-independence threshold of 2 for our safety property.

Observe that Snippet 4.4, which redefines $behaviour_T(\mathsf{Guard})$ so as to satisfy $\mathbf{PosConjEqT}_T$, is a traces anti-refinement of $behaviour_T(\mathsf{Guard})$ from Snippet 4.1. Hence, when used in place of the original in $System_T$, any trace of the original $System_T$ will also be present in the new $System_T$. Hence, if the new system is safe, the original one will be as well.

Let this new definition of $behaviour_T(\mathsf{Guard})$ replace the original in $System_T$. We will use results that underlie the theory of data-independence to show that if $System_T$ is not safe for some set T with size greater than 2, then $System_T$ is not safe when T is of size 2. We do so by adopting an approach taken by Roscoe and Broadfoot [RB99] to solve a similar problem in the context of verifying cryptographic protocols.

The basic idea is to relate the behaviours of a large system in which $|T| > 2$ to those of a small system in which $|T| = 2$ and to show that the presence of unsafe behaviours in the former imply the presence of related unsafe behaviours in the latter, which will show up when the safety property is applied to the small system. Fix T for some large system such that

[3]The relevant theorem is Theorem 15.2.3. For brevity, we avoid quoting it here.

$$
\begin{array}{l}
\mathit{behaviour}_T(\mathsf{Guard}) = \\
\quad \textbf{let} \\
\quad\quad \mathit{AllFroms} = T \cup \{\mathsf{Stamped}, \mathsf{SlotRead}, \mathsf{SlotWrite}\}, \\
\quad\quad \mathit{AllCaps} = T \cup \{\mathsf{Stamped}, \mathsf{SlotRead}, \mathsf{SlotWrite}, \mathsf{Guard}\}, \\
\quad\quad \mathit{AllVals} = \mathit{AllCaps} \cup \{\mathsf{null}\} \cup \mathit{Data} \\
\quad \textbf{within} \\
\quad ?\mathit{from} : \mathit{AllFroms}!\mathsf{Guard}!\mathsf{Call}?\mathit{specimen} : \mathit{AllCaps} \rightarrow \\
\quad \mathsf{Guard}!\mathsf{SlotWrite}!\mathsf{Call}!\mathsf{null} \rightarrow \mathsf{SlotWrite}!\mathsf{Guard}!\mathsf{Return}!\mathsf{null} \rightarrow \\
\quad \mathsf{Guard}!\mathit{specimen}!\mathsf{Call}!\mathsf{null} \rightarrow \mathit{specimen}!\mathsf{Guard}!\mathsf{Return}!\mathsf{null} \rightarrow \\
\quad \mathsf{Guard}!\mathsf{SlotRead}!\mathsf{Call}!\mathsf{null} \rightarrow \mathsf{SlotRead}!\mathsf{Guard}!\mathsf{Return}?\mathit{val} : \mathit{AllVals} \rightarrow \\
\quad \left(\begin{array}{l}
\mathsf{Guard}!\mathit{from}!\mathsf{Return}!\mathsf{null} \rightarrow \mathit{behaviour}_T(\mathsf{Guard}) \\
\sqcap \\
(\textbf{if } \mathit{val} = \mathit{specimen} \textbf{ then} \\
\quad \mathsf{Guard}!\mathit{from}!\mathsf{Return}!\mathsf{Guard} \rightarrow \mathit{behaviour}_T(\mathsf{Guard}) \\
\textbf{else } \mathit{STOP})
\end{array} \right)
\end{array}
$$

Snippet 4.4: A $\mathbf{PosConjEqT}_T$ traces anti-refinement of $\mathit{behaviour}_T(\mathsf{Guard})$.

$|T| > 2$ and let T' be any 2-element set, which is used in place of T in the small system. The relation between behaviours is constructed by taking a surjective function $\phi : T \rightarrow T'$ that maps each member of T to a member of T'. Observe that ϕ cannot be injective.

We write $\phi(T)$ to mean $\{\phi(t) \mid t \in T\}$ which is of course equivalent to T'. Given a process, P_T, that is parameterised by T, we write $P_{\phi(T)}$ to mean P_T with T replaced by $\phi(T) = T'$ and the renaming induced by ϕ applied to the values of any parameters that contain members of T that appear in P_T. Given a trace of events s, we write $\phi(s)$ to mean the result of applying ϕ to each component of each event of type T in s. As argued by Roscoe and Broadfoot [RB99], it can be shown that if P_T is data-independent in T and satisfies $\mathbf{PosConjEqT}_T$ and ϕ is any function whose domain is T, then

$$\{\phi(s) \mid s \in \mathit{traces}(P_T)\} \subseteq \mathit{traces}(P_{\phi(T)}) \tag{4.2}$$

Given any surjection $\phi : T \rightarrow T'$, observe that, for our system, $\mathit{System}_{\phi(T)} = \mathit{System}_{T'}$. This is because all components, such as $\mathit{behaviour}_T(\mathsf{Stamped})$, of System_T that take parameters containing values of type T in-fact take the every member of T, which will map to the entirety of T' under ϕ.

Equation 4.2 implies, then, that any unsafe trace, s, in the large system will show up in the small system as the trace $\phi(s)$, for all $\phi : T \rightarrow T'$. It then remains to be shown that for all such unsafe traces s of the large system, there exists some $\phi : T \rightarrow T'$ such that $\phi(s)$ is not present in the safety specification process $\mathit{Spec}_{T'}$ – meaning that this safety violation will be detected in the small system, $\mathit{System}_{T'}$.

So consider the unsafe traces, s, that the large system might exhibit. These are those that are not present in $Spec_T$, which appears in Snippet 4.3, but might be present in $System_T \setminus (\Sigma - SafeGuardEvents_T)$, where recall $SafeGuardEvents_T = \{|from.\mathsf{Guard.Call}, \mathsf{Guard}.from.\mathsf{Return} \mid from \in T \cup \{\mathsf{Stamped}, \mathsf{SlotRead}, \mathsf{SlotWrite}\}|\}$.

- s might contain two consecutive Call or Return events, which would certainly show up as two consecutive Call or Return events in $\phi(s)$ and would thus not be present in $Spec_{T'}$.
- s might instead contain a subsequence $\langle from.\mathsf{Guard.Call}.specimen, \mathsf{Guard}.from'.\mathsf{Return}.res\rangle$, where $from \neq from'$, in which Guard Returns to the wrong object. The only circumstance in which this safety violation won't be detected by $Spec_{T'}$ is when $from$ and $from'$ are both members of T and for all $\phi : T \to T'$ it is the case that $\phi(from) = \phi(from')$. This is clearly impossible, however, since $|T'| = 2$ so we can always find some ϕ that maps $from$ to one element of T' and $from'$ to the other.
- Finally, s might contain a subsequence $\langle from.\mathsf{Guard.Call}.specimen, \mathsf{Guard}.from.\mathsf{Return.Guard}\rangle$ where $specimen \neq \mathsf{Stamped}$. In this case, $\phi(s)$ will contain a similar subsequence $\langle from'.\mathsf{Guard.Call}.specimen', \mathsf{Guard}.from'.\mathsf{Return.Guard}\rangle$, where $specimen' \neq \mathsf{Stamped}$ too and thus this safety violation will be detected in the small system as well.

Therefore, if we can show that no safety violation occurs in $System_T$ when $|T| = 2$ and $|T| = 1$, we can conclude that no safety violation can occur in any $System_T$ when $|T| > 2$ and that, therefore, the Trademarks implementation remains safe in all systems captured by Figure 4.1. FDR reveals that the refinement check holds in both cases, with each check completing in under a second.

We have shown that a single instantiation of the Trademarks implementation in any system of the form of Figure 4.1 is safe. This effectively generalises the safety results to any system in which this pattern is deployed in which a single stamped object exists. We can generalise these results further, however, to any system in which this pattern is deployed in which multiple stamped objects exist. We do so in the following subsection in order to conclude the safety analysis of the Trademarks implementation.

4.1.4 Concluding the Trademarks Safety Analysis

Observe that any system in which the Trademarks implementation is correctly deployed will contain a guard, some stamped objects and some other objects. Each stamped object may have arbitrary capabilities, whilst none of the other (non-stamped) objects will have capabilities to the guard's slot. Any such system is therefore captured by Figure 4.2.

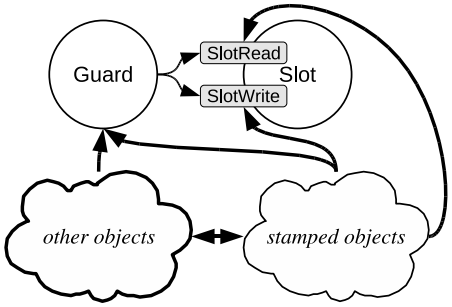

Figure 4.2: Concluding the Trademarks safety analysis.

Consider an arbitrary system, *System*, captured by this figure. Let U denote the set that contains all facets of all objects in the cloud labelled "stamped objects" and let T denote the set that contains all facets of all objects in the cloud labelled "other objects". We assert that *System* can be safely abstracted by a system, $System_{T,U}$, of the form of Figure 3.1, where $facets(\mathsf{Specimen}) = T$ and $facets(\mathsf{Stamped}) = U$, in which the behaviour of Stamped is extended to aggregate all of the stamped objects in *System*.

In order to ensure that Stamped properly aggregates all stamped objects from *System*, we need to extend its behaviour so that it: has and makes use of multiple facets (so that it can accommodate each of the facets in U), can send messages between its facets (to model communication between the stamped objects in *System*), and may place any capability to any of its facets in its slot (to model any of the stamped objects it aggregates placing a capability to itself in their common slot). Extending the behaviour of a stamped object in this way yields the new process $AStamped(facets, slotW, caps, data)$ in Snippet 4.5, which is now parameterised by the set *facets* that contains its facets, rather than its single identity *me* as before (see Snippet 3.4).

We claim whilst neglecting to show, based on the similarity between this definition of *AStamped* and that of $Untrusted_{OS}$, that a result analogous to Theorem 4.1.4 can be proved to show that any finite collection of stamped objects can be aggregated by a single instance of *AStamped* above. Hence, *System* can be safely abstracted by $System_{T,U}$, which has the form of Figure 3.1, when Stamped's behaviour is

$$AStamped(U, \mathsf{SlotWrite}, T \cup U \cup \{\mathsf{Guard}, \mathsf{SlotRead}, \mathsf{SlotWrite}\}, Data).$$

$System_{T,U}$ can be defined similarly to $System_T$ from the previous section, with each object behaviour also expressed in terms of the sets T and U, as can the specification $Spec_{T,U}$. Note that we use the new **PosConjEqT**-satisfying definition of Guard's behaviour. Doing so reveals that $System_{T,U}$ is data-independent in both T and U and satisfies both $\mathbf{PosConjEqT}_T$ and $\mathbf{PosConjEqT}_U$ as one would expect. Similar arguments can be made to

$$
\begin{array}{l}
AStamped(facets, slotW, caps, data) = \\
\quad \textbf{let}\ \ caps' = (caps - \{slotW\})\ \textbf{within} \\
\quad \left(\begin{array}{l}
?me : facets?to : caps'?op?arg : caps' \cup data \cup \{\mathsf{null}\} \rightarrow \\
\quad AStamped(facets, slotW, caps, data)\ \Box \\
?from : facets?to : facets?op?arg : caps \cup data \cup \{\mathsf{null}\} \rightarrow \\
\quad AStamped(facets, slotW, caps, data)\ \Box \\
?from : facets!slotW!\mathsf{Call}?arg : facets \rightarrow \\
\quad AStamped(facets, slotW, caps, data)\ \Box \\
?from : Capability - facets?me : facets?op?arg \rightarrow \\
\quad \textbf{let}\ \ C' = \{arg, from\} \cap Capability\ ;\ D' = \{arg\} \cap Data\ \textbf{within} \\
AStamped(facets, slotW, capsC', data \cup D')
\end{array}\right) \\
\sqcap\ STOP
\end{array}
$$

Snippet 4.5: An aggregation of multiple stamped objects.

those before that 2 is a sufficient data-independence threshold for each set T and U. Therefore, we conclude that if this system is safe for all choices of T and U of size 1 and 2, that it will be safe for all non-empty choices of T and U.

We must therefore carry out 2 refinement checks in FDR (one in which $|U| = 2 \wedge |T| = 1$, the other in which $|U| = |T| = 2$), on top of those carried out in the previous section. Carrying out these tests reveals that the system is safe in both cases; the tests take no more than a few seconds to complete. This is in sharp contrast to the tests required here for the original (non-**PosConjEqT**) definition of Guard's behaviour, which involve thresholds for both T and U of 8 and take around 24 hours to complete!

This generalises the safety results for this pattern obtained in the previous chapter to any system in which it might be deployed where stamped objects carry only a single trademark. We leave further generalisation of the safety results for this pattern as future work.

4.1.5 Generalising the Sealer-Unsealer Safety Analysis

We can also apply these same basic techniques to easily generalise the safety analysis of the single-threaded recursively-invocable Sealer-Unsealer from Section 3.2 to all single-threaded systems that have the form of Figure 4.3.

Let *System* be an arbitrary system captured by Figure 4.3 and let T denote the facets of the objects in the "other objects" cloud. Then *System* is safely abstracted by the system $System_T$ that has the form of Figure 3.2 in which $facets(\mathsf{Alice}) = T$. Because $System_T$ contains no equality tests between members of T, it satisfies $\mathbf{NoEqT}_T$.

Recall that the specification process against which the system in Figure 3.2 is checked, to assert that the Sealer-Unsealer implementation is safe,

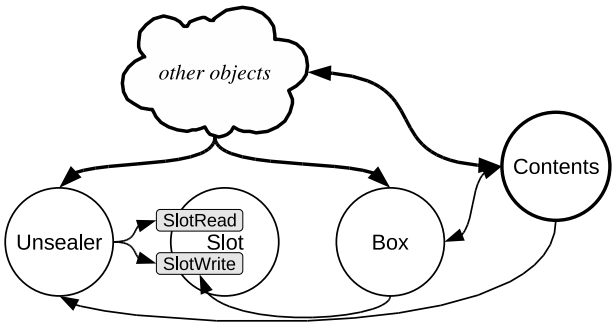

Figure 4.3: Generalising the Sealer-Unsealer safety analysis.

is the process *SafeUnsealerR* from Snippet 3.9. Theorem 2.3.5 is not appropriate for this specification. However, we can apply another theorem from [Ros97, Section 15.2] instead, namely Theorem 15.2.2 which we avoid quoting here for brevity, which implies that a threshold for T of 2 is sufficient for verifying the safety of the Sealer-Unsealer in $System_T$. The refinement check required to test this safety property when $|T| = 2$ takes less than 30 seconds to complete in FDR. The test passes, thereby generalising the Sealer-Unsealer safety analysis to all systems captured by Figure 4.3.

4.1.6 Summary

We have shown how to generalise the results obtained in the previous chapter to arbitrary-sized systems. To do this, we developed the theory of safe abstraction and aggregation from Section 4.1.1, inspired by similar ideas of Spiessens [Spi07], and coupled this with CSP's theory of data-independence [Laz99].

We saw that patterns that make use of EQ can require more work when applying this technique, when the systems in question don't satisfy **NoEqT** for the relevant data-independent types. Most patterns (the Sealer-Unsealer included) don't make use of EQ[4] and so these techniques should be easy to apply in most cases. For patterns that do make use of EQ, and don't satisfy **NoEqT**, we have seen that it is sometimes possible to still obtain easily tractable thresholds by manually constructing **PosConjEqT** safe abstractions of these patterns. Therefore, we have reason to believe that the techniques demonstrated here ought to be fairly widely applicable.

This demonstration that patterns that don't make use of EQ are easier to formally analyse, provides some formal justification to the opinion held by some object-capability practitioners that limiting the use of EQ can be desirable (see *e.g.* [Tri06]).

Note that whilst we have been able to generalise our earlier results to a considerable degree, we still cannot claim that the analysis performed here

[4]Indeed, some object-capability systems provide no EQ primitive whatsoever.

is fully exhaustive. One further generalisation, for example, would be to systems that allow an object to be recursively invoked an arbitrary number of times. We leave this as future work.

4.2 Handling Object Creation

So far, we have quietly ignored the issue of object creation. Recall from Section 2.2, that, in most object-capability systems, each object has the ability to create new objects. Upon creating an object, o, the creator is exclusively given a capability that refers to o. The creator, p, (and only the creator) of an object, o, may initially endow it with a subset of p's capabilities. p, as the sole recipient of the initial capability that refers to o, therefore has complete control over those objects that o might come to interact with in its lifetime.

In some object-capability systems, like almost all object-capability languages including E, Cajita and Joe-E, any object has the ability to create an arbitrary number of new objects[5]. In others, like object-capability operating systems including seL4, EROS and KeyKOS, the ability to create new objects can be limited, perhaps by the total amount of memory allocated to an object for itself and its children[6]. In any case, we have so far avoided explicitly modelling object creation in our CSP representation of the object-capability patterns that we have analysed. We must be able to capture object-creation in our CSP models if they are to be accurate representations of reality. In this section, we show how the techniques developed so far in this chapter can be used to achieve this. In doing so, we continue to borrow ideas from the work of Spiessens [Spi07].

4.2.1 Implicit Object Creation via Aggregation

In our CSP models, each object is represented by its own CSP process. Observe, then, that we cannot model unbounded object creation *explicitly*, as this would result in the parallel composition of an unbounded number of processes, resulting in an infinite-state system. This system would be impossible for FDR to model-check at the time of writing when it can handle only finite-state systems. This means that we have no choice but to represent the arbitrary number of child objects that a single object, p, might create as a finite, bounded collection of objects that aggregate all of p's children together. In most, if not all, cases, it is simplest to aggregate all of p's children together into a single object.

Even when aggregating all of p's children into a single object, we still cannot distinguish the act of p creating one child from the act of p creating

[5]The only exception to this of which we are aware is the work of Košíc [Košíc09].

[6]In KeyKOS and EROS, the abstraction used to track this allocation is known as the *space bank* [Har85, Sha99].

another, since doing so would lead to a system with an unbounded number of states. We must therefore not only aggregate children together, but also represent the creation of different children identically. Note that this is exactly the technique employed by Spiessens [Spi07] when facing the same problem, although in the context of Scoll rather than CSP.

We go one step further than Spiessens, however, by avoiding explicitly representing the act of object creation at all. While we could incorporate extra events into our CSP models to represent object creation, doing so appears to add little extra expressiveness whilst increasing their complexity. Instead, our CSP models initially incorporate all possible children that might be created. This makes our CSP models less precise abstractions of the real systems they are modelling but, because they allow more behaviours than real systems in question, they remain safe abstractions nonetheless.

Under this approach, generalisations performed earlier in this chapter already cover systems with unbounded object creation. For example, Figure 4.2 represents all systems in which the "other objects" in the cloud can each create an arbitrary number of children and each stamped object can also create an arbitrary number of children. A child of a stamped object may itself be stamped or not, influencing in which cloud it appears, depending on whether its parent endows it with a capability to the parent's slot's write facet.

This realisation demonstrates that we already have the machinery needed to reason about object creation; we've just chosen to ignore this fact until now for ease of exposition. To make this point explicit, we now consider the analysis of a pattern that uses unbounded object creation as part of its normal functioning.

4.3 Safe Revocable Membranes

4.3.1 The Membrane Pattern

The *Membrane* pattern [Don76, Raj89, Mil06] has a long history of application in object-capability systems; however, its basic idea has only recently been distilled and given its current name. Its primary function is to ensure that some behaviour or policy is applied to all capabilities that are reachable from a particular capability. It is best understood by example.

Suppose we have three untrusted objects, Alice, Bob and Carol, and that Alice is to have access to Bob and Bob is to have access to Carol. Alice's access to Bob is to be mediated in some way by an object FrwdsToBob that sits in between Alice and Bob and forwards a subset of the messages that it receives, from Alice, to Bob; it returns whatever response it receives from Bob, to Alice. This scenario is depicted in Figure 4.4.

In the case in which FrwdsToBob simply forwards all messages it receives

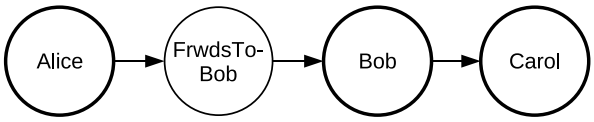

Figure 4.4: Membranes by Example.

to Bob, it might behave as the process $AFrwdsTo(\mathsf{FrwdsToBob}, \mathsf{Bob})$ where

$$AFrwdsTo(me, target) =$$
$$?from : Capability - \{me\}!me!\mathsf{Call}?arg \rightarrow me!target!\mathsf{Call}!arg \rightarrow$$
$$target!me!\mathsf{Return}?res \rightarrow me!from!\mathsf{Return}!res \rightarrow AFrwdsTo(me, target).$$

More generally, though, FrwdsToBob might impose some kind of policy on Alice's access to Bob, for example by forwarding only certain kinds of messages (perhaps forwarding only read-messages, rather than both read- and write-messages, thereby providing Alice with read-only access to Bob), or by forwarding messages only if some slot object, not depicted, contains a non-null value (so that by writing a null value to the slot, Alice's access to Bob can be revoked[7]). In this way, FrwdsToBob might control Alice's access to Bob in accordance with some security policy that it is to enforce.

However, suppose Alice invokes FrwdsToBob with a message that contains a capability to herself, which FrwdsToBob dutifully forwards to Bob, thereby giving Bob a capability that refers directly to Alice. Bob can now easily pass to Alice a capability that refers directly to himself, thereby giving Alice direct access to Bob in violation of the security policy. Alternatively, Bob might return to Alice (via FrwdsToBob) a capability to Carol which Alice might then use to acquire a direct capability to Bob in violation of the policy. Furthermore, in many instances, Alice acquiring just a direct capability to Carol might violate the spirit of the security policy. Suppose Bob represents a file that contains a number of component objects, including Carol, and that FrwdsToBob forwards only read-messages. Alice obtaining direct access to Carol might allow her to write to Carol and, in so doing, alter Bob's contents, despite supposedly having only read-only access to Bob.

We can see then that because it allows capabilities to pass unaltered between Alice and Bob, FrwdsToBob is an ineffective tool for enforcing these kinds of security policy. In the general case, it is desirable for whatever policy FrwdsToBob enforces on Alice's access to Bob, to also be applied to all capabilities that pass in between Alice and Bob via FrwdsToBob. This is the function of the Membrane pattern.

The Membrane pattern extends the behaviour of FrwdsToBob by having it *wrap* all capabilities that pass via it between Alice and Bob. Wrapping a capability, c, involves creating a new forwarding object, f, that has the same behaviour as FrwdsToBob but has c as its target rather than Bob. f is

[7] This is an instance of Redell's *Caretaker* pattern [Red74] for revocation.

then passed in place of c. In this way, f will also wrap any capabilities that pass across it, which is exactly what is required. With this new behaviour, FrwdsToBob and all children it creates form a so-called "membrane" that sits between Alice and Bob, hence the name of this pattern. This membrane should prevent Alice from obtaining direct access to Bob so long as, initially, neither has any means to pass capabilities to the other except via FrwdsToBob.

The use of object creation as a part of its proper functioning makes the Membrane pattern an ideal candidate for analysis in order to demonstrate how to reason about (patterns that involve) object creation using the techniques developed earlier in this chapter.

4.3.2 Revocable Membranes

We will analyse a membrane implementation that implements a revocation policy. Here, each forwarding object that comprises the membrane has access to a common slot object, s, that it checks before deciding whether to forward each invocation to its target. Invocations are forwarded so long as s contains a non-null value. s is a special slot object that accepts only the value null as an argument to write-messages. Hence, s will permanently contain the value null after it has been written to for the first time. The first write to s thus causes the membrane to be revoked. We refer to s as a *gate*, the act of writing to s as *closing* s and s's write-facet as its *close*-facet.

The behaviour of a gate with read- and close-facets *readme* and *closeme* respectively, whose initial value is *val*, is captured by the process $AGate(readme, closeme, val)$, which is defined in Snippet 4.6.

$$
\begin{aligned}
&AGate(readme, closeme, val) = \\
&\quad ?from : Capability - \{readme, closeme\}!readme!\mathsf{Call}!\mathsf{null} \rightarrow \\
&\qquad readme!from!\mathsf{Return}!val \rightarrow AGate(readme, closeme, val) \ \Box \\
&\quad ?from : Capability - \{readme, closeme\}!closeme!\mathsf{Call}!\mathsf{null} \rightarrow \\
&\qquad closeme!from!\mathsf{Return}!\mathsf{null} \rightarrow AGate(readme, closeme, \mathsf{null})
\end{aligned}
$$

Snippet 4.6: The behaviour of a gate.

We wish to analyse this pattern deployed in the class of systems captured by Figure 4.5. Here we see a revocable membrane interposed between two disjoint clouds of objects, neither of which has access to the other except possibly via the close facet, GateClose, of the gate object, Gate. Each cloud is labelled with the set that comprises all facets of all objects in the cloud. This figure seems to capture most, if not all, instantiations of this pattern that don't trivially break the security properties it is designed to uphold (by *e.g.* providing direct access between the two clouds of "other objects").

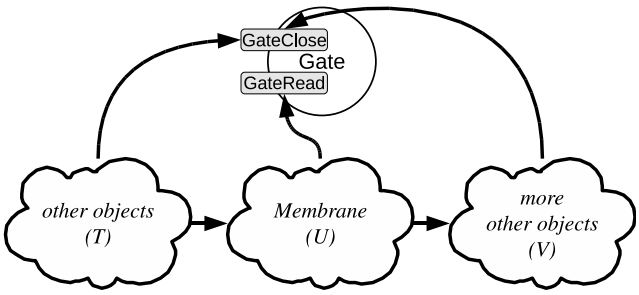

Figure 4.5: A general analysis of the revocable Membrane pattern. Each cloud of objects is labelled with the set, T, U or V, that comprises all facets of all objects in the cloud.

Objects that implement the membrane, upon receiving a message, first check the value of their common gate, Gate, by invoking its read-facet, GateRead, to decide whether to forward the message. The behaviour of the gate is simply $AGate(\mathsf{GateRead}, \mathsf{GateClose}, \mathsf{GateClose})$. Note that GateClose provides a potential channel between the two clouds of objects. Indeed, it is expected that any object in either cloud should be able to revoke the membrane by invoking GateClose. However, because GateClose does not allow the gate's value to be updated so as to hold a capability, we expect that the presence of this channel should not thwart the membrane.

We will model a data-independent safe abstraction that captures the class of systems depicted in Figure 4.5. When doing so, our goal will be to aggregate each cloud of objects in the figure into a separate object, giving a final safe abstraction with a total of four objects, as depicted in Figure 4.6. Here, we see a direct correspondence with the sets of facets in Figure 4.5, indicating which object in Figure 4.6 aggregates which cloud of objects in Figure 4.5.

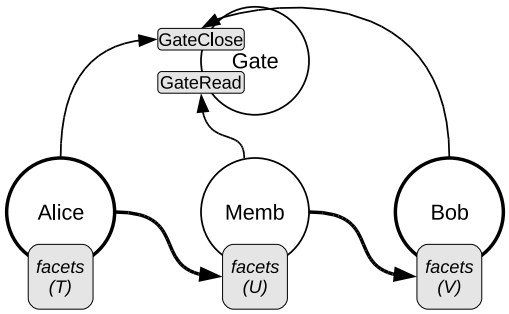

Figure 4.6: A safe abstraction of the revocable Membrane pattern. Here, $facets(\mathsf{Alice}) = T$, $facets(\mathsf{Memb}) = U$ and $facets(\mathsf{Bob}) = V$.

We will model this pattern in the single-threaded context before modelling it in the concurrent context, because the former case is simpler to

model[8].

4.3.3 The Single-Threaded Case

We will define a single-threaded system, $System_{T,U,V}$, that has the form of Figure 4.6 and show that it is an aggregation and, hence, safe abstraction of any single-threaded system captured by Figure 4.5. We will then analyse $System_{T,U,V}$ to draw conclusions about how the revocable Membrane pattern behaves generally in the single-threaded context.

It makes most sense to assume that Alice is the object that is initially active. It is clear that Alice and Bob each need to initially possess a capability to GateClose. In order to absolutely ensure that Alice and Bob are both data-independent in T, U and V, we construct each such that if it initially possesses any capability from one of these sets, then it initially possesses all of the capabilities from that set[9]. Doing so causes no problem, of course, since an untrusted object that initially possesses all capabilities from U, for example, safely abstracts the same object that possesses any subset of the capabilities in U. This leads to the following definitions for the behaviour of all objects except Memb.

$$
\begin{aligned}
&\mathit{behaviour}(\mathsf{Alice}) = \mathit{UntrustedActive}_{\mathit{lang}}(T, T \cup U \cup \{\mathsf{GateClose}\}, \mathit{Data}),\\
&\mathit{behaviour}(\mathsf{Bob}) = \mathit{Untrusted}_{\mathit{lang}}(V, V \cup \{\mathsf{GateClose}\}, \mathit{Data}),\\
&\mathit{behaviour}(\mathsf{Gate}) = \mathit{AGate}(\mathsf{GateRead}, \mathsf{GateClose}, \mathsf{GateClose}).
\end{aligned}
$$

Whilst we set the gate's initial contents to be a capability to its close-facet, this choice is somewhat arbitrary since any non-null value would do.

The Membrane

It remains to define the initial capabilities and the behaviour of Memb. Recall that Memb aggregates all of its children that it creates. We define the behaviour of a revocable membrane whose initial target is *target*, whose gate's read-facet is *gateR*, and whose initial forwarding object and all children it may create are drawn from the set U, as $\mathit{ARMemb}_U(\{\mathit{target}\}, \mathit{gateR})$. This process is defined in Snippet 4.7.

The process $\mathit{ARMemb}_U(\mathit{targets}, \mathit{gateR})$ is the aggregation of a membrane whose forwarding objects are drawn from the set U, whose common gate's

[8] We choose to model this pattern in the concurrent context despite the question of how to implement the Membrane pattern in an object-capability operating system being currently unresolved. It is generally agreed that creating a new forwarding object to wrap each capability that traverses a membrane is simply too expensive in this context (see *e.g.* [Bri07]).

[9] Alternatively, if Alice is to initially possess a capability u from U, for instance, we could have her nondeterministically select u from the entirety of U, which would ensure her behaviour is data-independent in U. Giving her every capability in U, however, makes it easier to ensure that we don't introduce implicit equality tests between members of U, allowing our system to satisfy $\mathbf{NoEqT}_U$ and us to obtain a lower threshold for U.

$$
\begin{array}{l}
ARMemb_U(targets, gateR) = \\
\quad ?from : Capability - U ?me : U !\mathsf{Call} ?arg \rightarrow \\
\quad me!gateR!\mathsf{Call}!\mathsf{null} \rightarrow gateR!me!\mathsf{Return}?val \rightarrow \\
\quad \textbf{if } val = \mathsf{null} \textbf{ then} \\
\qquad me!from!\mathsf{Return}!\mathsf{null} \rightarrow ARMemb_U(targets, gateR) \\
\quad \textbf{else} \\
\qquad me\$target : targets!\mathsf{Call}\$u_{arg} : wrap_U(arg) \rightarrow target!me!\mathsf{Return}?res \rightarrow \\
\qquad me!from!\mathsf{Return}\$u_{res} : wrap_U(res) \rightarrow \\
\qquad \textbf{let } T' = \{arg, res\} - (U \cup \{\mathsf{null}\} \cup Data) \\
\qquad \textbf{within } ARMemb_U(targets \cup T', gateR); \\
wrap_U(\mathsf{null}) = \{\mathsf{null}\}, \\
wrap_U(other) = \textbf{if } other \in Data \textbf{ then } \{other\} \textbf{ else } U.
\end{array}
$$

Snippet 4.7: A membrane aggregation in the single-threaded context.

read-facet is *gateR* and the set of all objects to which some forwarding object of the membrane may forward invocations to is *targets*. An initial membrane that has just a single target *target* and whose gate's read facet is *gateR* is naturally captured, then, by the process $ARMemb_U(\{target\}, gateR)$.

The process $ARMemb_U(targets, gateR)$ receives invocations to all facets in the set U from any object other than one from the set U. This means that it may receive invocations to facets that represent forwarding objects that have not yet been created; however, doing so safely abstracts any process that keeps track of those objects from U that have been created so far and only allows those ones to be Called, whilst helping to avoid introducing any implicit equality tests between members of U. Any such invocation is forwarded to some target, $target \in targets$, chosen nondeterministically. Only capability arguments are ever wrapped. Wrapping null or an argument from *Data* leaves it unchanged. Each time an argument *arg* is wrapped and a new child is created to forward invocations to *arg*, *arg* is added to the set of potential targets *targets* (so long as *arg* is not a member of U).

Because we avoid keeping track of those children from U that have already been created to ensure we don't introduce implicit equality tests between members of U, we allow the identity of each new child to be selected nondeterministically from the entirety of U. Choosing the identity of each new child nondeterministically from the entirety of U is safe, since U is a superset of the set of all children yet to be created and so the behaviour of $ARMemb_U$ will be an anti-refinement of the behaviour of a process that chooses identities of new children only from those that haven't yet been allocated.

We argue (whilst again neglecting to prove) that $ARMemb_U$ is a suitable aggregation of a membrane in the single-threaded context. Note that it is

not likely to be a suitable aggregation in a more concurrent context, however, because there it might be possible for one child object to be invoked while another is handling a separate invocation, for example. In this case, the aggregated object would need to include extra behaviours, like being able to receive new invocations whilst it is currently servicing others, in order to be a proper aggregation.

The behaviour of Memb can now be defined. Memb must initially possess some capability from V that is its initial target. We therefore give it the entirety of V for the same reason that Alice initially possesses the entirety of U as discussed above. Memb's behaviour is therefore

$$\mathit{behaviour}(\mathsf{Memb}) = \mathit{ARMemb}_U(V, \mathsf{GateRead}).$$

Analysing the System

This completes the system, $\mathit{System}_{T,U,V}$, which is data-independent in T, U and V and satisfies $\mathbf{NoEqT}_T$, $\mathbf{NoEqT}_U$ and $\mathbf{NoEqT}_V$.

There are two security properties that we expect this pattern to uphold in $\mathit{System}_{T,U,V}$. Firstly, we expect that Alice should not be able to obtain direct access to Bob or, more precisely, that no event from the set $\{|a.b \mid a \in T, b \in V|\}$ should ever be able to be performed. We can test this by checking that $\mathit{STOP} \sqsubseteq_T \mathit{System}_{T,U,V} \setminus (\Sigma - \{|a.b \mid a \in T, b \in V|\})$.

Secondly, we expect that once revocation occurs, that the membrane will no longer be willing to forward invocations. The obvious interpretation of this property is captured by the specification process $\mathit{RevocationSpec}_{T,U,V}$, which asserts that after GateClose has been Called, Memb (for which $\mathit{facets}(\mathsf{Memb}) = U$ recall) will no longer Call anyone except GateRead to check the gate's value. This process appears in Snippet 4.8.

$$\begin{array}{l}\mathit{RevocationSpec}_{T,U,V} = \\ \quad \textbf{let}\ \ A = \{|u.x.\mathsf{Call} \mid u \in U, x \in T \cup U \cup V \cup \{\mathsf{GateClose}\}|\}\ \textbf{within} \\ \quad ?e : \Sigma - \{|f.\mathsf{GateClose.Call} \mid f \in T \cup U \cup V|\} \rightarrow \mathit{RevocationSpec}_{T,U,V}\ \Box \\ \quad ?\mathit{from} : T \cup U \cup V!\mathsf{GateClose}!\mathsf{Call}!\mathsf{null} \rightarrow \mathit{CHAOS}_{\Sigma - A}\end{array}$$

Snippet 4.8: A specification for safe revocation.

Testing this revocation property of $\mathit{System}_{T,U,V}$ amounts to testing whether $\mathit{RevocationSpec}_{T,U,V} \sqsubseteq_T \mathit{System}_{T,U,V}$.

Observe that neither of the specifications above, STOP and $\mathit{RevocationSpec}_{T,U,V}$, ever stores or makes use of any particular value of T, U or V. That is, anytime that either of these processes uses one value of T, U or V, it might just as easily use another from the respective set. Both specifications, therefore, satisfy the conditions of Theorem 2.3.5. Hence, because $\mathit{System}_{T,U,V}$ satisfies $\mathbf{NoEqT}_T$, $\mathbf{NoEqT}_U$ and $\mathbf{NoEqT}_V$, we can test

each refinement for all non-empty choices for T, U and V by testing whether it holds when each set is instantiated as a singleton set, disjoint from the others.

Thus let $T = \{\mathsf{Alice}\}$, $U = \{\mathsf{Memb}\}$ and $V = \{\mathsf{Bob}\}$. Testing the above refinements, with these values for T, U and V, reveals that both hold. We can conclude, therefore, that the revocable membrane pattern is safe generally in the single-threaded context.

4.3.4 The Concurrent Case

We now consider how to model this pattern in the concurrent context. We will see that doing so is more complicated than for the single-threaded context, but that the basic techniques described in this chapter can still be applied with some minor extensions.

The main difficulty that arises is trying to define an accurate aggregation of a membrane when each object that comprises it can be invoked simultaneously. Recall that the aggregation used in the single-threaded context (defined in Snippet 4.7) is not an aggregation in the concurrent context, because in the concurrent context each child of the membrane can be invoked simultaneously. We could define a process that is always willing to be invoked. However, this leaves the question open as to when it should stop forwarding invocations. Suppose it receives two invocations, performing the trace $\langle \mathit{from}.\mathit{me}.\mathsf{Call.null}, \mathit{from}'.\mathit{me}'.\mathsf{Call.null}\rangle$, for two of its facets me and me', and then checks its gate for the first invocation and finds that it is closed. A simple process would simply now refuse to forward any invocations, including the second invocation from from' that it has already received. However, this process is not a valid aggregation, since the child me' that received the second invocation has not yet learnt that the membrane has been revoked. This implies that any reasonably accurate aggregation needs to maintain separate state for each child that it aggregates. This would prevent it from being data-independent in the type of its children since the amount of state it needs to store is dependent on the number of children it creates.

We conjecture that finding an accurate aggregation for the membrane in the concurrent context is difficult because the behaviour of the membrane and each its children is not *monotonically increasing* [Spi07]. We say that an object's behaviour is monotonically increasing if and only if whenever the object performs two sequences s and t of events, such that the events contained in t are a subset of those contained in s, then its behaviour after performing t is a refinement of its behaviour after performing s. Consider the other aggregations we've constructed for the concurrent case so far, namely $\mathit{Untrusted}_{OS}$ in Snippet 2.1 and *AStamped* in Snippet 4.5. The behaviour of both objects increases monotonically over time as they interact with others in their environment. This is in contrast to the children of a membrane, each of whose behaviours decreases once it learns that the membrane has been revoked. We conclude, therefore, that building accurate aggregations

for the concurrent context of objects whose behaviour cannot be modelled so as to increase monotonically over time is non-trivial.

This suggests that we should consider whether we can obtain verification results for the revocable membrane pattern by modelling it using a less accurate aggregation (although one that is still, by definition, a safe abstraction), whose behaviour is mostly monotonically increasing.

We adopt the following strategy. Recall that our goal is to create an aggregation of a membrane whose initial forwarding object and all children it might create are drawn from the set U. We build an aggregation that models one of the objects $u \in U$ accurately. Because it is modelled accurately, the behaviour of u is not monotonically increasing. This aggregation models the other objects from $U' = U - \{u\}$ very approximately (but safely), so that their behaviour is in-fact monotonically increasing. While this aggregation will be a safe abstraction, we cannot expect the revocation property encoded by $RevocationSpec_{T,U,V}$ (in Snippet 4.8) to hold for the other objects in U'; however, we might test whether it holds for the object u that has been modelled accurately.

This suggests that we could construct a specification process that asserts that the forwarding object u satisfies this revocation property but ignores the behaviour of the (other) forwarding objects in U'. If we can show that u, chosen arbitrarily from U, is safe, we can conclude by symmetry all U are safe and that the membrane implementation is safe generally in the concurrent context.

The Membrane

We begin by modelling the aggregation of the membrane. A membrane whose forwarding objects are drawn from the set U, whose initial set of targets is *targets*, and whose capability to its gate's read-facet is *gateR*, and for whom the single object $u \in U$ is modelled accurately, is captured by the process $ARMembC_U(u, targets, gateR)$ which appears in Snippet 4.9.

The membrane is modelled as a parallel composition of two processes: $SingleF_U(u, gateR)$ and $OtherFs_U(U', targets, froms, gateR)$ where $U' = U - \{u\}$ and initially $froms = \{\}$. The first process in this composition represents the single forwarding object u that is modelled accurately; its behaviour is not monotonically increasing. The second represents a very coarse but safe abstraction of the other forwarding objects; its behaviour is monotonically increasing. The set *froms* tracks the total set of objects that have invoked one of the forwarding objects in U'.

Certain internal events, which are hidden away when the two processes are composed together, are used to communicate between the two processes. Their purpose will be explained shortly.

The process $OtherFs_U(U', targets, froms, gateR)$ is very nondeterministic and captures all possible behaviours of the forwarding objects in U'. It nondeterministically allows any of these forwarding objects to receive a Call

(from a client) or Return (from a target that has previously been Called on behalf of some client) at any time, in which case the argument *arg* is added to the set of potential targets *targets*, so long as it is not a member of U, *Data* or the value null. The object *from* that sent the message is also added to the set *froms* of potential clients that have Called one of the forwarding objects in U'. $OtherFs_U$ also nondeterministically allows any object $me \in U'$ to Call *gateR* or receive a Return message from *gateR*. In the latter case, the value *val* Returned by *gateR* is discarded. This aggregation, therefore, never explicitly learns that the membrane has been revoked, in order for its behaviour to be monotonically increasing. The aggregation may also nondeterministically Call any target $target \in targets$ and it may also Return to any client $from \in froms$ that has Called it.

$OtherFs_U$ may also participate in the internal events gettarget, recvtarget and updatetargets, which are used to allow it to communicate with the process $SingleF_U$, which represents the single forwarding object $u \in U$ that is being modelled accurately. These events are explained directly in the context of $SingleF_U$.

$SingleF_U(u, gateR)$ is an accurate model of the single forwarding object u, whose gate's read-facet is *gateR*. It may receive a Call from any non-U object *from* passing some argument *arg*. It then Calls *gateR* to learn the current value *val* held by its gate and simply Returns null to *from* if $val =$ null. Otherwise, it should Call some target on behalf of *from*, passing a wrapped copy of the argument *arg*. This raises the question as to which target it should Call. This target must be some capability that was previously wrapped by the membrane, *i.e.* it must come from the set *targets* held by the process $OtherFs_U$. Hence, $SingleF_U$ then communicates with *OtherFs* to have *OtherFs* nondeterministically select some target $target \in targets$ that $SingleF_U$ will subsequently Call. The gettarget event is used to ask *OtherFs* to nondeterministically choose the target *target*, which is then sent to $SingleF_U$ on the channel recvtarget. Having received *target*, $SingleF_U$ then Calls it on behalf of *from*, passing it a wrapped copy u_{arg} of *from*'s argument *arg*.

Having Called *target*, $SingleF_U$ then waits for *target* to Return from this Call with some result *res*. $SingleF_U$ then Returns a wrapped copy u_{res} of *res* to *from*. *arg* and *res* must then be added to the set *targets*, held by $OtherFs_U$, of the membrane's targets. This is achieved by communicating *arg* and *res* to $OtherFs_U$ on the channel updatetargets. $SingleF_U$ then returns to its initial state.

The process $ARMembC_U(u, targets, gateR)$ is data-independent in U, but is parameterised by a *constant symbol* u of type U, against which it implicitly compares members of U for equality. For instance, when the first Call message is received by $ARMembC_U(u, targets, gateR)$, there is an implicit equality test performed to determine whether the Call is for the object u or whether it is for one of the other objects from $U' = U - \{u\}$.

$$
\begin{array}{l}
ARMembC_U(u, targets, gateR) = \\
\quad \textbf{let } U' = U - \{u\} \textbf{ within} \\
\quad (SingleF_U(u, gateR) \ {}_{E(\{u\})}\|_{E(U')} \ OtherFs_U(U', targets, \{\}, gateR)) \setminus I,
\end{array}
$$

$$
\begin{array}{l}
E(Z) = \{\!|c.z, z.c, \mid o \in Capability \wedge z \in Z|\!\} \cup I, \\
I = \{\!|\mathsf{gettarget}, \mathsf{recvtarget}, \mathsf{updatetargets}|\!\},
\end{array}
$$

$$
\begin{array}{l}
OtherFs_U(U', targets, froms, gateR) = \\
\quad \Big(U' \neq \{\} \ \& \\
\qquad ?from : Capability - U \$ me : U' \$ op : \{\mathsf{Call}, \mathsf{Return}\} ?arg \rightarrow \\
\qquad\quad \textbf{let } T' = \{arg\} - (U \cup \{\mathsf{null}\} \cup Data) \textbf{ within} \\
\qquad\quad OtherFs_U(U', targets \cup T', froms \cup \{from\}, gateR) \ \sqcap \\
\qquad \$me : U' ! gateR ! \mathsf{Call} ! \mathsf{null} \rightarrow OtherFs_U(U', targets, froms, gateR) \ \sqcap \\
\qquad gateR \$ me : U' ! \mathsf{Return} ? val \rightarrow OtherFs_U(U', targets, froms, gateR) \ \sqcap \\
\qquad \$me : U' \$ target : targets ! \mathsf{Call} \$ u_{arg} \in (U \cup \{\mathsf{null}\} \cup Data) \rightarrow \\
\qquad\quad OtherFs_U(U', targets, froms, gateR) \ \sqcap \\
\qquad froms \neq \{\} \ \& \\
\qquad\quad \$me : U' \$ from : froms ! \mathsf{Return} \$ u_{res} \in (U \cup \{\mathsf{null}\} \cup Data) \rightarrow \\
\qquad\quad OtherFs_U(U', targets, froms, gateR) \\
\qquad \sqcap \ STOP \\
\quad \Big) \\
\quad \Box \\
\qquad \mathsf{gettarget} \rightarrow \mathsf{recvtarget} \$ target : targets \rightarrow \\
\qquad\quad OtherFs_U(U', targets, froms, gateR) \ \Box \\
\qquad \mathsf{updatetargets} ? arg ? res \rightarrow \\
\qquad\quad \textbf{let } T' = \{arg, res\} - (U \cup \{\mathsf{null}\} \cup Data) \textbf{ within} \\
\qquad\quad OtherFs_U(U', targets \cup T', froms, gateR),
\end{array}
$$

$$
\begin{array}{l}
SingleF_U(u, gateR) = \\
\quad ?from : Capability - U ! u ! \mathsf{Call} ? arg \rightarrow \\
\quad u ! gateR ! \mathsf{Call} ! \mathsf{null} \rightarrow gateR ! u ! \mathsf{Return} ? val \rightarrow \\
\quad \textbf{if } val = \mathsf{null} \textbf{ then} \\
\qquad u ! from ! \mathsf{Return} ! \mathsf{null} \rightarrow SingleF_U(u, gateR) \\
\quad \textbf{else} \\
\qquad \mathsf{gettarget} \rightarrow \mathsf{recvtarget} ? target \rightarrow \\
\qquad u ! target ! \mathsf{Call} \$ u_{arg} : wrap_U(arg) \rightarrow target ! u ! \mathsf{Return} ? res \rightarrow \\
\qquad u ! from ! \mathsf{Return} \$ u_{res} : wrap_U(res) \rightarrow \mathsf{updatetargets} ! arg ! res \rightarrow \\
\qquad SingleF_U(u, gateR).
\end{array}
$$

Snippet 4.9: An aggregation of the membrane for the concurrent context.

$ARMembC_U$ obviously behaves differently depending on the outcome of this implicit equality test. Hence, it doesn't satisfy $\mathbf{NoEqT}_U$. It also doesn't satisfy $\mathbf{PosConjEqT}_U$, because when some $u' \in U'$ is Called (and so the implicit equality test $u' = u$ fails), $ARMembC_U$ obviously doesn't deadlock and become $STOP$. However, $ARMembC_U(u, targets, gateR)$ does satisfy the weaker property $\mathbf{PosConjEqT}'_{U,\{u\}}$ [RB99]. For some set C of constants of type U, the property $\mathbf{PosConjEqT}'_{U,C}$ is similar to the property $\mathbf{PosConjEqT}_U$ but exempts any equality tests between members of U for which one argument is a constant from C from having to satisfy the $\mathbf{PosConjEqT}_U$ restriction. This will become important directly when we analyse the membrane.

Analysing the System

To analyse this model of the membrane, we instantiate the system depicted in Figure 4.6 as before, except that we now instantiate Alice and Bob as instances of $Untrusted_{OS}$ and replace the behaviour of the membrane Memb as one would expect, so that

$$\begin{aligned} behaviour(\mathsf{Alice}) &= Untrusted_{OS}(T, T \cup U \cup \{\mathsf{GateClose}\}, Data), \\ behaviour(\mathsf{Bob}) &= Untrusted_{OS}(V, V \cup \{\mathsf{GateClose}\}, Data), \\ behaviour(\mathsf{Memb}) &= ARMembC_U(u, V, \mathsf{GateRead}), \end{aligned}$$

for some member $u \in U$ that we fix later when fixing the value of U. Doing so, we arrive at the system $System_{T,U,V}(u)$, parameterised by the sets T, U and V and the constant symbol $u \in U$. $System_{T,U,V}(u)$ satisfies $\mathbf{NoEqT}_T$, $\mathbf{NoEqT}_V$ and $\mathbf{PosConjEqT}_{U,\{u\}}$.

We wish to verify two safety properties of this system. The first property we want to check is, as before, that

$$STOP \sqsubseteq_T System_{T,U,V}(u) \setminus (\Sigma - \{|a.b \mid a \in T, b \in V|\}), \tag{4.3}$$

i.e. that the clouds labelled T and V in Figure 4.5 cannot become directly connected. Because $System_{T,U,V}(u)$ satisfies $\mathbf{NoEqT}_T$ and $\mathbf{NoEqT}_V$, applying Theorem 2.3.5, we obtain thresholds for this test for T and V of 1 as before. To derive a threshold for U, we use the following result that is similar to Equation 4.2 and is derived from [RB99].

Let $P_U(u_1, \ldots, u_n)$ be a process, parameterised by some set U and some constants $u_1, \ldots, u_n$ from U, that is data-independent in U and satisfies $\mathbf{PosConjEqT}'_{U,C}$ where $C = \{u_1, \ldots, u_n\}$. Let ϕ be a surjection whose domain is U that is *faithful* to each constant in C, meaning that for all $u' \in U$ and $c \in C$, $\phi(u') = \phi(c) \Leftrightarrow u' = c$. Then

$$\begin{aligned} &\{\phi(s) \mid s \in traces(P_U(u_1, \ldots, u_n))\} \subseteq \\ &\quad traces(P_{\phi(U)}(\phi(u_1), \ldots, \phi(u_n))). \end{aligned} \tag{4.4}$$

This result can be used to show that the set U can be instantiated with size 2 and the constant u chosen arbitrarily, such that if the resulting system satisfies Equation 4.3 then so must any larger system where $|U| > 2$ and T and V are unchanged.

Consider any such large system $System_{T,U,V}(u)$ for which $|U| > 2$ and u is some arbitrary constant in U. Consider a small fixed-sized system $System_{T,U^S,V}(u^S)$ in which the set U is instantiated by the set U^S of size 2 and the constant $u \in U^S$ is chosen arbitrarily and denoted u^S. Let u'^s denote the element of U^S that is not u^S. Let ϕ be a surjection such that $\phi(u) = u^S$ and $\phi(u') = u'^S$ for all $u' \in U - \{u\}$. Then $System_{T,U^S,V}(u^S) = System_{T,\phi(U),V}(\phi(u))$, so by Equation 4.4,

$$\{\phi(s) \mid s \in traces(System_{T,U,V}(u))\} \subseteq traces(System_{T,U^S,V}(u^S)). \quad (4.5)$$

Now consider any trace s of $System_{T,U,V}(u)$ that violates the refinement of Equation 4.3. It must contain some event e from $\{|a.b \mid a \in T, b \in V|\}$. Note that $\phi(e) \in \{|a.b \mid a \in T, b \in V|\}$ too and that $\phi(e)$ is present in $\phi(s)$ which must be a trace of $System_{T,U^S,V}(u^S)$. Hence, this smaller system would also fail the refinement check.

We therefore test the refinement of Equation 4.3 twice with T, U and V instantiated as disjoint sets of fresh values: once when $|T| = |U| = |V| = 1$ and again when $|T| = |V| = 1 \wedge |U| = 2$ with u chosen arbitrarily from U. It holds in both cases, thus proving that the revocable membrane upholds this safety property generally in the concurrent context.

The second safety property we would like to check is that the revocation property holds for the forwarding object $u \in U$. We modify the previous revocation specification $RevocationSpec_{T,U,V}$ from Snippet 4.8 to assert the revocation property for just the object u, arriving at the specification $RevocationSpecC_{T,U,V}(u)$ below.

$RevocationSpecC_{T,U,V}(u) =$
let $A = \{|u.x.\mathsf{Call} \mid x \in T \cup U \cup V \cup \{\mathsf{GateClose}\}|\}$ **within**
$?e : \Sigma - \{|f.\mathsf{GateClose.Call} \mid f \in T \cup U \cup V|\} \rightarrow RevocationSpecC_{T,U,V}(u)$
$\Box ?from : T \cup U \cup V!\mathsf{GateClose!Call!null} \rightarrow CHAOS_{\Sigma - A}$

We assert this revocation property by testing that

$$RevocationSpecC_{T,U,V}(u) \sqsubseteq_T System_{T,U,V}(u). \quad (4.6)$$

Theorem 2.3.5 may be applied as before to show that 1 is a sufficient threshold for T and V when testing this refinement. For U, we apply the same argument as before to show that it is sufficient to test this refinement when $|U| = 2$ and u is chosen arbitrarily, to guarantee it for all larger systems with arbitrary choices of u.

Consider the systems $System_{T,U,V}(u)$ and $System_{T,U^S,V}(u^S)$ and ϕ as before. Then Equation 4.5 holds. Then consider any trace s of

$System_{T,U,V}(u)$ that violates Equation 4.6. s must contain some event $d \in \{x.\mathsf{GateClose.Call.} \mid x \in T \cup U \cup V\}$ followed by some event $e \in \{|u.z.\mathsf{Call} \mid z \in T \cup U \cup V \cup \{\mathsf{GateClose}\}|\}$. Note that $\phi(d) \in \{x.\mathsf{GateClose.Call.} \mid x \in T \cup U^S \cup V\}$ and $\phi(e) \in \{|u^S.z.\mathsf{Call} \mid z \in T \cup U^S \cup V \cup \{\mathsf{GateClose}\}|\}$. So $\phi(s)$, which is guaranteed to be a trace of $System_{T,U^S,V}(u^S)$ is not a trace of $RevocationSpecC_{T,U^S,V}(u^S)$ and so the small system would fail the refinement as well.

Hence, we carry out the above refinement check twice, as with the previous one. We find, perhaps surprisingly, that the check does *not* hold, even when $|U| = 1$ (with $U = \{\mathsf{Memb}\}$). FDR returns the following trace as a counter-example.

$$\langle \mathsf{Alice.Memb.Call.null},\ \mathsf{Memb.GateRead.Call.null},\ \mathsf{GateRead.Memb.Return.GateClose},\ \mathsf{Bob.GateClose.Call.null},\ \mathsf{Memb.Bob.Call.null} \rangle$$

We see here that Alice Calls the membrane Memb who then checks the value held in its gate and finds the gate to be open. Then Bob (concurrently) Calls GateClose and closes the gate, after which Memb proceeds to forward Alice's invocation to Bob. This indicates that this revocation property cannot be upheld in the concurrent context because another object can close a membrane's gate in between when the membrane has checked it and found it to be open, and subsequently forwards the invocation. This allows a membrane to forward at least a single invocation after its gate has been closed.

We should expect, however, that a membrane should forward no more than one invocation after its gate has been closed. $RevocationSpecC_{T,U,V}(u)$ is easily modified to assert this weaker revocation property, yielding a new specification $WeakRevocationSpecC_{T,U,V}(u)$ defined in Snippet 4.10.

$$\begin{aligned}
&WeakRevocationSpecC_{T,U,V}(u) = \\
&\quad \textbf{let}\ A = \{|u.x.\mathsf{Call} \mid x \in T \cup U \cup V \cup \{\mathsf{GateClose}\}|\}\ \textbf{within} \\
&\quad ?e : \Sigma - \{|f.\mathsf{GateClose.Call} \mid f \in T \cup U \cup V|\} \rightarrow \\
&\qquad WeakRevocationSpecC_{T,U,V}(u) \\
&\quad \Box ?from : T \cup U \cup V!\mathsf{GateClose}!\mathsf{Call}!\mathsf{null} \rightarrow WeakRevocationSpecC'_{T,U,V}(A), \\
&WeakRevocationSpecC'_{T,U,V}(A) = \\
&\quad ?e : \Sigma - A \rightarrow WeakRevocationSpecC'_{T,U,V}(A)\ \Box ?a : A \rightarrow CHAOS_{\Sigma - A}.
\end{aligned}$$

Snippet 4.10: A weaker revocation specification for the concurrent context.

By a similar argument to that used above, we can show that the thresholds obtained for $RevocationSpecC_{T,U,V}(u)$ are suitable for $WeakRevocationSpecC_{T,U,V}(u)$. Testing the refinement

$$WeakRevocationSpecC_{T,U,V}(u) \sqsubseteq_T System_{T,U,V}(u)$$

twice as with the previous one, as dictated by these thresholds, reveals that it holds in both cases. Hence, we can conclude that the revocable membrane upholds this weaker revocation property generally in the concurrent context.

4.3.5 Summary

We've modelled the revocable membrane in both the single-threaded and concurrent contexts. We saw that creating a suitable aggregation in the single-threaded context was far simpler than for the concurrent context and concluded that, in general, building accurate aggregations of collections of objects whose behaviours are not monotonically increasing is problematic in the concurrent context. Instead, we built a less accurate (but safe) aggregation that modelled just one of the membrane's forwarding objects $u \in U$ accurately, while allowing the behaviour of the others to increase monotonically. We then verified that the single accurately modelled, arbitrarily chosen forwarder u was safe and, by symmetry, concluded that each of the membrane's forwarders must be safe and so the membrane as a whole must be too. This was easily achieved since the aggregation satisfied the data-independence property $\mathbf{PosConjEqT}'_{U,\{u\}}$.

We believe that this technique is generally applicable in cases such as this where one needs to build an aggregation of a collection of objects whose behaviours don't increase monotonically.

This analysis allowed us to conclude that the membrane satisfies slightly different revocation properties when in the single-threaded and concurrent contexts respectively. In particular, it satisfies a stronger revocation property in the former context in which it is impossible for a membrane's gate to be closed while the membrane is invoked.

To our knowledge, this is the first time that the revocable membrane pattern has been modelled and its revocation properties explicitly reasoned about. In particular, Spiessens' previous analysis [Spi07, Section 8.3.1] of the membrane pattern did not consider its revocation properties explicitly, thereby preventing that analysis from distinguishing how the membrane's revocation properties differ between the single-threaded and concurrent contexts, as we have done here.

4.4 Related Work

Aggregation and Safe Abstraction The ideas of aggregation and safe abstraction as described in this chapter were inspired by similar ideas of Spiessens in the context of the Scoll formalism [Spi07]. As here, a Scoll aggregation collects the behaviours of multiple objects into a single object. As here, Scoll aggregations are also safe abstractions. Safe abstractions in Scoll are referred to as *safe approximations*.

Scoll is essentially a deductive system. An initial system comprises a set

of of *facts* that are true initially and a set of rules for how to derive new facts from the current set, as the system evolves.

Our approach requires that a safe abstraction of a system exhibit all behaviours of that system. Similarly, in the context of Scoll, one system is a safe approximation of another if every fact derivable in the latter is also derivable in the former.

Scoll has the useful property that any collection of objects, S, can be accurately aggregated into a single object, o, with a single identity such that any fact derivable about any object from S can now be derived for o. Restated in the terminology familiar from this chapter: in a Scoll system, any cloud of objects, where T denotes the set comprising all of their facets, can always be accurately aggregated into a single object, o, where $\mathit{facets}(o) = T$, resulting in a system for which any Scoll safety property always yields a data-independence threshold for T of 1.

In contrast, in our CSP models, it is not always straightforward to accurately aggregate a cloud of objects into a single object. An obvious example is the membrane object in the concurrent context from the previous subsection. Recall that we conjectured that an accurate aggregation was difficult to achieve here because the behaviour of a membrane's children does not increase monotonically over time. This situation cannot arise in Scoll, since the Scoll formalism admits only behaviours that increase monotonically over time. Hence, the cases in which aggregation is more complicated to achieve and the resulting analysis more complicated to perform in CSP cannot be expressed directly in Scoll.

In our approach, after constructing some aggregation, there is no guarantee that the resulting system will be adequately data-independent in the type T of aggregated facets to yield a data-independence threshold of 1. These problems cannot arise in Scoll, specifically because the Scoll formalism effectively prevents one from specifying behaviours that are not data-independent or contain equality tests between members of data-independent types. Scoll also effectively limits the safety properties one can test so to be expressible as specifications captured by Theorem 2.3.5, thereby inducing effective data-independence thresholds of just 1.

Again, we see that the complications that arise when using our technique that cannot arise when using Scoll, do so only in cases that cannot be directly expressed in Scoll. It should be noted that any Scoll system can be expressed in CSP, since Scoll systems are instances of what Roscoe calls *positive deduction systems* [RB99]. CSP has often been used in the past to express such deductive systems within the context of analysing cryptographic protocols, where such deductive systems capture the behaviour of a hostile intruder [RSG+00].

Scoll has a distinct advantage over our approach when it comes to aggregation. In particular, one can apply a straightforward syntactic transformation to a Scoll system in order to produce an aggregation of it (with the

transformation being parameterised by which objects are being aggregated together, of course). As far as we are aware, no similar procedure exists by which one can syntactically produce an aggregation of a CSP system.

On the other hand, the extra expressiveness of our CSP models means that we can reason easily about behaviours and properties that are more difficult to reason about in Scoll. For example, it is difficult to see how one can directly reason about properties such as "the object may Return only to its most recent Caller and no-one else" (as $Spec_T$ from Snippet 4.3 does implicitly) in Scoll. Scoll is also incapable of directly reasoning about behaviours such as revocation (as we did when analysing the revocable Membrane pattern in Section 4.3), because Scoll can express only behaviours that are monotonically increasing. Such behaviours must be simulated in Scoll by using multiple objects to represent a single object before and after revocation has occurred, for instance. This naturally makes reasoning about such properties more complicated in Scoll.

We have also seen that we can build aggregations, such as of the revocable membrane in Section 4.3, whose behaviour does not increase monotonically over time. This allowed us to reason directly about the revocation properties of a revocable Membrane pattern in each of the single-threaded and concurrent contexts and to prove that they are in-fact different in each case. Spiessens [Spi07, Section 8.3.1] also analysed a Membrane pattern using Scoll, applying aggregation to reason about object creation. However, Spiessens' analysis could not consider revocation directly and, therefore, did not draw such distinctions.

Despite its extra complications, we argue that the approach advocated in this chapter of this thesis has important advantages over the closest ancestor from which it was derived. However, the Scoll approach to aggregation also has some advantages over our approach. We conclude, therefore, that each approach has its strengths and weaknesses and that each may be preferable in different circumstances.

Data-Independence We have applied data-independence arguments in this chapter to show that security violations in arbitrary sized systems, in which object-capability patterns might be deployed, must show up in the smaller fixed-sized systems we have analysed. Similar arguments have been made, using the same data-independence results that we've applied here, by others when performing a range of previous security analyses, including of cryptographic protocols [RB99, Bro01, Kle08] and intrusion detection systems [RL05].

The approach taken in this chapter has been to apply data-independence arguments to various types, T, U and so on, that are *subtypes* of a larger type, *Capability*, that comprises all facets in a system. In contrast, in the context of security protocols for example, the usual approach has been to apply data-independence arguments to an entire type, such as the type of

agent identities in a system, treating certain values from that type as constants (see *e.g.* [RB99]). In either case, the end result is much the same.

We have also applied data-independence arguments to multiple subtypes, T, U and so on, at once, as in the analysis of the Membrane pattern in which data-independence arguments were applied to three subtypes. In contrast, Lazić and Roscoe [LR99] show how to apply data-independence arguments to a type that is effectively partitioned into subtypes by predicates that range over the type. It is conceivable that we could have applied Roscoe and Lazić's more general theory; however, our approach is arguably simpler whilst being sufficient for our purposes.

Analysing Object-Capability Patterns in CSP Ours [Mur08] is the only prior work of which we are aware that applies CSP to analyse object-capability patterns. This chapter significantly expands upon that work. It should be noted that in [Mur08] the idea of aggregation is applied informally to model a revocable Membrane. The resulting model is much the same as the single-threaded one that appears in Snippet 4.7. This similarity is only a happy accident, however, since the model from [Mur08] was derived informally without the machinery presented in the first part of this chapter to ensure it was sound. In particular, that model implicitly assumes that a data-independence threshold of 1 for the type containing the membrane's facets (and the types that contain the facets of each of the other untrusted objects respectively) is sufficient when analysing this pattern. Here, we have shown that this unstated assumption is correct for the single-threaded case only.

Unfortunately, in [Mur08], the same model of an aggregated revocable membrane is used in both the single-threaded and concurrent contexts. As argued earlier in this chapter, the aggregation used in the single-threaded context is simply not valid in the concurrent context, in which two objects that make up a membrane can be invoked simultaneously for instance. The extra work taken to formalise the ideas of aggregation and safe abstraction in this chapter made this flaw abundantly clear when it wasn't before. Hence, the formal theory in this chapter has certainly proved its worth.

Architectural Refinement In [vdM09], van der Meyden considers the problem of preserving information flow policies under *architectural refinement*. A system *architecture* $\mathcal{A}_1$ that comprises a set D_1 of components may be *architecturally refined* by an architecture $\mathcal{A}_2$ that comprises a set D_2 of components. $\mathcal{A}_2$ specifies the internal structure of some of the components of $\mathcal{A}_1$ and so D_2 contains more components than D_1 because some of the components in D_1 have been broken up into separate components in D_2. The notion of architectural refinement is captured by a surjection $r : D_2 \to D_1$. Each component $d \in D_1$ in $\mathcal{A}_1$ is represented in $\mathcal{A}_2$ by the set of components $r^{-1}(d_1) \subseteq D_2$.

Comparing this idea against Definition 4.1.2, we observe a strong similarity between van der Meyden's notion of architectural refinement and the idea of aggregation (which we've adapted from Spiessens [Spi07]). In particular, saying that $\mathcal{A}_1$ is an architectural refinement of $\mathcal{A}_2$ is very much like saying that $\mathcal{A}_2$ is an aggregation of $\mathcal{A}_1$. We discuss architectural refinement further in Section 5.6.

4.5 Conclusion

In this chapter we have shown how to reason about patterns in systems of arbitrary size in which objects may create arbitrary numbers of others. One does so by building a fixed-sized safe abstraction of any arbitrary-sized system, by aggregating multiple objects from the arbitrary-sized system into a single object, and then applying CSP's theory of data-independence to generalise the analysis of the fixed-sized system to all such arbitrary-sized ones.

We saw that this approach could be used to generalise the safety analyses performed in the previous chapter to a considerable degree. However, we still cannot say that these analyses are fully exhaustive. For example, the the Sealer-Unsealer analysis considers only unsealers that can be recursively Called no more than once. We leave these problems as future work.

We saw that results for patterns that make use of EQ can be harder to generalise because such systems do not generally satisfy the data-independence property **NoEqT**. For the Trademarks pattern, we showed in Section 4.1.3 that an easily tractable data-independence threshold could still be obtained by building a safe abstraction of the pattern that satisfies the weaker data-independence property **PosConjEqT**.

We believe this approach should be widely applicable to generalising the analysis of safety properties of patterns that make use of EQ. This is because, like our Trademarks implementation, most of these patterns (such as the Grant Matcher [Mil00] and Stiegler's NonTransferable Claim Check [Sti06], for instance) use EQ tests to decide whether to perform some special behaviour that should occur only when an EQ test succeeds. Each EQ test may therefore be captured by a process of the form **if** $c = d$ **then** P **else** Q, where P represents the special behaviour guarded by the EQ test. These patterns can always be safely abstracted by a **PosConjEqT** process that instead does $Q \sqcap$ **if** $c = d$ **then** P **else** $STOP$ (see Snippet 4.4). This abstraction won't fail any safety property involving the occurrence of the special behaviour P that the original process doesn't.

For the vast majority of patterns, which don't make use of EQ, this extra effort is unnecessary, as shown by the generalisation of the Sealer-Unsealer safety analysis in Section 4.1.5.

We also saw that these same techniques can be used to reason about patterns involving unbounded object creation. We demonstrated this by analysing a revocable Membrane implementation in both the single-threaded

and concurrent contexts, finding that it upholds a weaker revocation property in the latter case due to the extra concurrency that exists there.

We observed some difficulties in the concurrent context with creating accurate aggregations of objects whose behaviour is not monotonically increasing, as is the case with the revocable Membrane. We showed that one can instead create less accurate (but still safe) aggregations that accurately model the behaviour of only one of the aggregated objects $u \in U$, chosen arbitrarily, whilst modelling the others very abstractly to allow their behaviour to be monotonically increasing. We saw that one can then reason about just the behaviour of u and, by symmetry, draw conclusions about the behaviour of the aggregation in general. We saw that tractable data-independence thresholds could be obtained for the type U being aggregated because the resulting aggregation naturally satisfied the data-independence property $\mathbf{PosConjEqT}'_{U,\{u\}}$ by treating u as a constant symbol of type U. We believe that this technique should be widely applicable, since all such aggregations should also naturally satisfy $\mathbf{PosConjEqT}'$ in this way.

This chapter has effectively shown how to apply Spiessens' idea of *aggregation* [Spi07, Section 5.7.4], developed in the context of the Scoll formalism, to our CSP models, with the help of Lazić's theory of data-independence for CSP. The extra power afforded by CSP allows us to reason accurately about behaviours and properties that cannot be expressed directly in Scoll; however, this power renders the resulting theory, and its application, somewhat more complex. This complexity is fully paid for in the next chapter, where we extend these ideas to reason about *covert channels* (*i.e.* those not explicitly modelled) of object-capability patterns in arbitrary-sized systems. This is beyond the scope of Scoll, which can reason only about explicitly modelled phenomena.

5 Information Flow

5.1 Introduction

In this chapter, we consider how to analyse the information flow properties of object-capability patterns. These properties are more sophisticated than the safety properties we have examined so far and are concerned with detecting influence and, hence, information flow between objects.

For instance, consider the system depicted in Figure 5.1, which we will use as a running example throughout this chapter. It contains two arbitrary objects, High and Low, that have access to high- and low-classification data respectively. The object, DataDiode, placed between High and Low, is designed to allow Low to send data to High but to prevent information flowing in the reverse direction, in accordance with the standard multi-level confidentiality policy (see *e.g.* [BL76]).

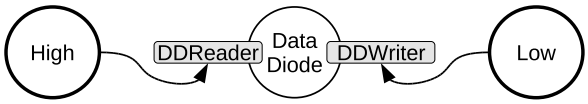

Figure 5.1: A simple system containing a data-diode.

DataDiode processes each message sent to its write-facet, DDWriter, by storing whatever data is contained in the message. In response to each message sent to its read-facet, DDReader, DataDiode sends back a reply message that contains the data it is currently storing. In this way, it allows data (but not capabilities[1]) to be written to DDWriter and then read from DDReader.

There are two ways in which information might flow from High to Low, in violation of the system's security policy, namely either via *overt* or *covert* means. Information flows via overt means when it is carried in a message sent between objects. So Low may obtain some high-classification data overtly by receiving a message, from some object with facet f, in which that data is contained, in which case the system will perform some event $f.\mathsf{Low}.op.arg$ where arg is some datum of high classification. Hence, a lack of overt infor-

[1] Recall from Section 2.2 that a fundamental property of any object-capability system is that capabilities must be distinguishable from data.

mation flow from High to Low can be expressed trivially as a safety property (see Chapter 3) that asserts that no such events can ever occur[2].

Expressing a lack of covert information flow, on the other hand, requires a more sophisticated property, which asserts that High cannot influence, or *interfere* with, Low. This idea was formalised in the seminal work of Goguen and Meseguer [GM82] as the property of *noninterference*, which asserts that any two executions of the system that differ only in the actions of High should be indistinguishable to Low, meaning that they both have the same effect upon Low. Equivalently, noninterference asserts that High's actions in the system should have no effect upon Low.

As an example, suppose that DataDiode is implemented such that it stores only a single datum at a time; invoking its read-facet, DDReader, causes it to return the datum it currently holds after which it becomes empty; however it accepts messages sent to its write-facet, DDWriter, only when it is empty. Then consider the following two executions of the system that differ only in High's actions. In the first, Low writes some data to DataDiode, High reads this data and then Low writes some more data. In the second, Low writes some data to DataDiode but High chooses not to read this data, then Low tries to write some more data to DataDiode but this request fails since High has not yet read the data that was previously written. We see here that these two executions can be distinguished by Low because in the former Low's second write succeeds whilst in the latter it fails. Equivalently, we might say that High reading from DataDiode affects whether Low's second write succeeds or fails, thereby affecting Low. Therefore, noninterference is not satisfied by this system.

In this example, DataDiode allows information to propagate covertly from High to Low because High can signal to Low by choosing to read from DataDiode or not, which causes an effect that Low can observe. Noninterference can therefore be used to capture the idea that DataDiode does not allow information to pass covertly from High to Low.[3]

Many different information flow properties have been defined for CSP processes that capture the idea of noninterference (and other similar notions) in different ways (see *e.g.* [GCS91, Rya91, RWW94, Ros95, FG95, Foc96, For99, RS01, Low07] amongst others). In this chapter, we consider how to apply these kinds of properties in order to reason about whether security-enforcing patterns like DataDiode allow unwanted covert flows of information, *i.e.* whether these patterns contain unwanted covert channels. This kind of analysis is vital in order to ensure that confidentiality policies are not inadvertently violated by the security-enforcing abstractions supposedly deployed to enforce them.

[2]This kind of overt information flow was first analysed explicitly in the context of object-capability systems by Spiessens [Spi07].

[3]Note that our use of the words "covert" and "overt" here may differ from how these terms are used by others. In particular, others may consider both kinds of information flow discussed above to be overt.

In Section 5.2, we derive a general definition for information flow security for determining the information flow properties of object-capability patterns modelled in CSP. In Section 5.3 we show how this general definition can be instantiated and then automatically tested in FDR. In Section 5.4, we use FDR to automatically detect covert channels in an implementation [Mil06, Figure 11.2] of the Data-Diode pattern (depicted in Figure 5.1), before showing how the implementation can be corrected. We then show, in Section 5.5, how to generalise the analysis of information flow properties of object-capability patterns to systems of arbitrary size, by adapting the approach taken earlier in Chapter 4. We then demonstrate this by generalising the analysis of the Data-Diode pattern to systems with arbitrary numbers of high and low objects with arbitrary high and low data.

5.2 Defining Information Flow for Object-Capability Patterns

In this section, we derive a general definition for information flow security for object-capability patterns modelled in CSP.

5.2.1 Refinement

We begin by considering the issue of refinement. Many standard information flow properties are known to suffer from the so-called *refinement paradox* (of which Lowe provides some examples in [Low07]). A property *Prop* suffers from the refinement paradox when for some process P, $Prop(P)$ holds but $Prop(Q)$ doesn't hold for some (failures-divergences) refinement Q of P. Restricting our attention to divergence-free P and Q (as we do throughout this thesis), refinement here corresponds to resolving nondeterminism in P. The refinement paradox is dangerous, then, because it allows a process to be judged secure, when under some resolution of its nondeterminism, it may in-fact be insecure.

A fail-safe way to avoid the refinement paradox is to employ properties that are *refinement-closed* [Low07]. Recall (from Section 2.1.4) that a property is refinement-closed when for all processes P, it holds for P only if it holds for all of P's refinements. This suggests that we can avoid the refinement paradox, when constructing a general definition of information flow security for object-capability patterns, by choosing a definition that is refinement-closed.

While this would ensure that we avoid the refinement paradox, this approach is not generally appropriate for analysing object-capability patterns modelled in CSP as described in this thesis. In particular, any definition of information flow security should not necessarily be closed over *all* refinements of a system $System = \|_{o \in Object}(behaviour(o), \alpha(o))$ to which it is being applied. This is because such processes have refinements that we need

to exclude in order to talk sensibly about the information flow properties of patterns modelled as part of *System*.

Consider, for example, the system depicted in Figure 5.1, in which High and Low each exhibit arbitrary behaviour. Considering the more general case in which this pattern is being instantiated in a context, such as an object-capability operating system, where each object runs concurrently to all others, High and Low are each represented in *System* by an instance of the $Untrusted_{OS}$ process, which is defined in Snippet 2.1. For appropriate definitions giving the facets of each object and for the sets *HighData* and *LowData* (each of which is a subset of the set *Data* that contains all data in the system) that contain the high- and low-classification data in the system respectively, the behaviours of High and Low are given in Snippet 5.1.

$$
\begin{aligned}
&\mathit{behaviour}(\mathsf{High}) = \\
&\quad \mathit{Untrusted}_{OS}(\mathit{facets}(\mathsf{High}), \mathit{facets}(\mathsf{High}) \cup \{\mathsf{DDReader}\}, \mathit{HighData}), \\
&\mathit{behaviour}(\mathsf{Low}) = \\
&\quad \mathit{Untrusted}_{OS}(\mathit{facets}(\mathsf{Low}), \mathit{facets}(\mathsf{Low}) \cup \{\mathsf{DDWriter}\}, \mathit{LowData}).
\end{aligned}
$$

Snippet 5.1: The behaviours of High and Low.

These processes that represent the behaviour of High and Low respectively are each highly nondeterministic, which makes *System* highly nondeterministic. The nondeterminism in Low means that initially (and in every subsequent state) Low may either perform or refuse to perform the event Low.Low.Call.null, in which Low invokes itself, depending on how Low's initial nondeterminism is resolved. Initially, High can perform the event High.High.Call.null, in which High calls itself. Taken together, this means that *System* has a refinement that exhibits both of the following behaviours:

- the trace $\langle \mathsf{High.High.Call.null}, \mathsf{Low.Low.Call.null} \rangle$ in which the nondeterminism in Low is resolved such that Low chooses to invoke itself initially and this invocation occurs after High invokes itself, and
- the failure $(\langle\rangle, \{\mathsf{Low.Low.Call.null}\})$ in which the nondeterminism in Low is resolved the other way such that initially Low chooses not to invoke itself, without High invoking itself.

The presence of these two behaviours means that *System* fails various refinement-closed information flow properties, including Lowe's *Refinement-Closed Failures Non-Deducibility on Compositions* [Low07] and Roscoe's *Lazy Independence* [Ros97, Section 12.4]. To see why, consider the partial executions of the system that precede the occurrence/refusal of the event Low.Low.Call.null in each case above. In the first case, this execution involves High calling itself. In the second case, this execution is the empty

trace in which no events have taken place. Both of these executions differ only in the occurrence of High events, yet Low behaves differently after each of them (by calling itself in the first case and refusing to do so in the second). Hence, any refinement of *System* that contains these two behaviours should fail noninterference.

5.2.2 A Necessary Assumption

However, we need to rule out this refinement in order to talk sensibly about the information flow properties of DataDiode. The two behaviours above represent the situation in which High's interactions with *just itself* can (somehow) affect Low. We cannot hope to ever prevent High from covertly sending information to Low in any system that allows this kind of information flow to occur. Hence, we cannot talk sensibly about the information flow properties of the Data-Diode pattern in any such system that exhibits both of these behaviours.

In more general terms, in order to talk sensibly about the information flow properties of object-capability patterns, we need to assume that the only way for one object to directly influence another is through overt message passing (*i.e.* sending it a message or receiving a message from it), since it is only overt message passing that security-enforcing objects like DataDiode that instantiate these patterns can hope to control. We assume therefore that objects can influence each other only by exchanging messages and that the only objects influenced by the sending and receipt of a message are the message's sender and receiver. This implies that the transmission of a message from some object o_1 to some object o_2 should not affect any other object $o_3 \notin \{o_1, o_2\}$. Thus, in any system, we assume that the resolution of nondeterminism in each object o can be influenced only by the message exchanges in which o has partaken before the nondeterminism is resolved.

Without specifying *how* the nondeterminism in any object o may be resolved after it has engaged in some sequence s of message exchanges, this therefore implies that whenever o performs s, its nondeterminism should be resolved *consistently*. Two resolutions of the nondeterminism in a process after it has performed s are *inconsistent* when it can perform some event e in one but refuse e in the other. Under this definition observe that, in the example above, the two different resolutions of the nondeterminism in Low, depending on whether High has performed its event that Low cannot overtly observe, are inconsistent. In the first case, Low performs the event Low.Low.Call.null initially, *i.e.* after it performs the empty trace; however, in the second, it refuses this event initially.

5.2.3 A Definition

When applying an information flow property to a system *System*, we therefore restrict our attention to those refinements of *System* that correspond

to the resolution of nondeterminism in its component objects, in which this sort of inconsistency cannot arise. For a refinement $System'$ of $System$, this occurs precisely when for every pair of behaviours of $System'$, no object performs some trace $s \frown \langle e \rangle$ in one but refuses e after performing s in the other, *i.e.* exhibits the failure $(s, \{e\})$ in the other. Observe that this corresponds precisely to saying that the behaviour of each object in $System'$ must be *deterministic* (see Definition 2.1.1).

If the behaviour of each object $o \in Object$ in some refinement $System'$ of $System = \|_{o \in Object}(behaviour(o), \alpha(o))$ is deterministic, then it must be representable by a deterministic process b_o that naturally refines its original behaviour $behaviour(o)$. Hence, $System'$ must be (failures-divergences) equivalent to the parallel composition of a set of deterministic processes b_o, one for each $o \in Object$, each of which refines its corresponding $behaviour(o)$. We call such a $System'$ a *deterministic componentwise refinement* of $System$.

Definition 5.2.1 (Deterministic Componentwise Refinement)**.** Sys' is a *deterministic componentwise refinement* of an alphabetised parallel composition $Sys = \|_{i \in \{1,\ldots,n\}}(P_i, A_i)$ iff there exists n processes $Q_1, \ldots, Q_n$ such that $Sys' \equiv_{FD} \|_{i \in \{1,\ldots,n\}}(Q_i, A_i)$ and $\forall\, i \in \{1, \ldots, n\} \bullet P_i \sqsubseteq_{FD} Q_i \wedge det(Q_i)$.

The set of deterministic componentwise refinements of an alphabetised parallel composition Sys is denoted $DCRef(Sys)$.

Given some information flow property $Prop$, we should therefore apply $Prop$ to all deterministic componentwise refinements of $System = \|_{o \in Object}(behaviour(o), \alpha(o))$ in order to test whether $System$ satisfies the information flow property encoded by $Prop$ under the assumption that the only way for one object to affect another in $System$ is through overt interaction. This leads to the following general definition of information flow security for object-capability patterns modelled in CSP, which is parameterised by the information flow property $Prop$ being tested.

Definition 5.2.2 (Information Flow Security for Object-Capability Patterns)**.** An object-capability system captured by the CSP process $System = \|_{o \in Object}(behaviour(o), \alpha(o))$ is secure under componentwise refinement with respect to the information flow property $Prop$ iff $\forall\, System' \in DCRef(System) \bullet Prop(System')$.

5.3 Testing Information Flow for Object-Capability Patterns

This general definition is instantiated by choosing an appropriate information flow property to substitute for $Prop$. There are two questions that must be resolved in order to produce an automatic test for information flow security for object-capability patterns: firstly, what properties ϕ are suitable

to substitute for *Prop* and, secondly, given a substitute ϕ for *Prop*, how can we automatically test that ϕ holds for all deterministic componentwise refinements of some system? We address both of these questions in this section.

5.3.1 Choosing an Appropriate Property

We begin by considering the question of which properties are suitable choices to substitute for *Prop* in Definition 5.2.2.

Standard laws of CSP for alphabetised parallel composition [Ros97] may be trivially applied to show that any alphabetised parallel composition of deterministic processes yields a deterministic process. Hence, any deterministic componentwise refinement is itself deterministic. Therefore, any choice for *Prop* will be applied only to deterministic processes.

This significantly reduces the number of effectively different choices for *Prop*. This is because while many different information flow properties have been described in the literature, many of these properties coincide when applied only to deterministic processes. We will show that there are basically only two effectively different choices for *Prop*.

One reason for this is that any property coincides with its *refinement-closure* when applied to deterministic processes. The refinement-closure of a property ϕ is the property $RC\phi$ that holds for a process P iff $\phi(Q)$ holds for all (failures-divergences) refinements Q of P.

Definition 5.3.1 (Refinement-Closure)**.** The *refinement-closure* of a property ϕ is the property $RC\phi$ where

$$RC\phi(P) = \forall\, Q \bullet P \sqsubseteq Q \Rightarrow \phi(Q).$$

A deterministic process P has no proper failures-divergences refinements [Ros97, Lemma 9.1.1] (meaning that it is its only refinement) and so

$$\phi(P) \Leftrightarrow (\forall\, Q \bullet P \sqsubseteq Q \Rightarrow \phi(Q)) \Leftrightarrow RC\phi(P).$$

Given the choice between either a property ϕ or its refinement-closure $RC\phi$, when choosing an appropriate substitute for *Prop* in Definition 5.2.2, we should prefer $RC\phi$. This is because $RC\phi$ is guaranteed to be a refinement-closed property and refinement-closed properties may be expressed in terms of refinement checks that are simpler than those for non-refinement-closed properties [Ros05]. It turns out that this makes the resulting information flow property, obtained by substituting $RC\phi$ for *Prop* in Definition 5.2.2, easier to express as a refinement check for FDR. This is because this refinement check can be constructed by adapting the refinement check for $RC\phi$, as shown later in Sections 5.3.2 and 5.3.3.

Transitive Noninterference Properties

Most of the *transitive noninterference properties*[4] ϕ, which capture the basic noninterference property that the occurrence of actions by high objects can have no effect upon the remaining (low) objects in the system, coincide with Lowe's Refinement-Closed Failures Non-Deducibility on Compositions [Low07] (RCFNDC) when applied to deterministic processes. Lowe shows that [Low07, Theorem 4] RCFNDC is equivalent to the refinement-closure of any property that is no stronger than RCFNDC but no weaker than Focardi's *Failures Non-Deducibility on Compositions* [Foc96]. This includes a number of other properties from the literature (see [Low07, Figure 1]) including those of Focardi *et al.* [FG95, Foc96] and Forster [For99]. Hence, RCFNDC coincides with all of these properties when applied to deterministic processes.

For convenience, we use the following characterisation of RCFNDC derived from [Low07, Low09]. Like other transitive noninterference properties for CSP, RCFNDC detects whether the occurrence of high events from some set H can influence the occurrence of the remaining (low) events from the set L, where H and L together partition the total alphabet of the system.

Proposition 5.3.2 (RCFNDC)**.** A divergence-free process P satisfies RCFNDC, written $RCFNDC(P)$, iff

$$\nexists\, s, l \bullet s \upharpoonright H \neq \langle\rangle \wedge l \in L \wedge \left(\begin{array}{l} (s \frown \langle l \rangle \in traces(P) \wedge (s \setminus H, \{l\}) \in failures(P)) \vee \\ (s \setminus H \frown \langle l \rangle \in traces(P) \wedge (s, \{l\}) \in failures(P)) \end{array} \right).$$

We show that, for deterministic processes, RCFNDC also coincides with any property that is no stronger than Roscoe's Lazy Independence [Ros97, Section 12.4] (which is stronger than RCFNDC) and no weaker than Ryan's traces-based formulation of noninterference for CSP processes [Rya91, Equation 1] (which is weaker than Focardi's Failures Non-Deducibility on Compositions).

We use the following characterisation of Lazy Independence, which comes from [ML09b].

Proposition 5.3.3 (Lazy Independence)**.** A divergence-free process P satisfies Lazy Independence, written $\mathcal{L}IND(P)$, iff

$$\nexists\, s_1, s_2, l \bullet s_1 \setminus H = s_2 \setminus H \wedge l \in L \wedge s_1 \frown \langle l \rangle \in traces(P) \wedge (s_2, \{l\}) \in failures(P).$$

[4] This name is used to distinguish them from the so-called *intransitive noninterference properties* [HY86, Rus92, RG99] that are more sophisticated. We briefly discuss intransitive noninterference properties later in this subsection.

Lowe shows that Lazy Independence is stronger than RCFNDC [Low07, Theorem 3], meaning $\mathcal{L}IND(P) \Rightarrow RCFNDC(P)$ but not the reverse. We show that for deterministic processes, if RCFNDC holds then so does Lazy Independence. This allows us to conclude that the two properties coincide for deterministic processes.

Lemma 5.3.4. If P is deterministic then $RCFNDC(P) \Rightarrow \mathcal{L}IND(P)$.

Proof. We prove the contrapositive, namely $\neg\mathcal{L}IND(P) \Rightarrow \neg RCFNDC(P)$.

So assume $\neg\mathcal{L}IND(P)$. By Proposition 5.3.3, P has some traces s_1 and s_2, such that $s_1 \setminus H = s_2 \setminus H$, and some event $l \in L$, such that $s_1 \frown \langle l \rangle \in traces(P) \wedge (s_2, \{l\}) \in failures(P)$. Because P is deterministic, $s_1 \neq s_2$ and, because $s_1 \setminus H = s_2 \setminus H$, we must have that $s_1 \upharpoonright H \neq \langle\rangle \vee s_2 \upharpoonright H \neq \langle\rangle$. There are two cases to consider: both s_1 and s_2 contain H events or not.

Suppose both traces s_1 and s_2 contain H events. Let t be the longest prefix of s_1 such that $t \setminus H \in traces(P)$. Then $t \upharpoonright H \neq \langle\rangle$ because t must contain at least the first H event in s_1. So t is a strict prefix of $s_1 \frown \langle l \rangle$. Let e be the event that follows t in $s_1 \frown \langle l \rangle$. Then necessarily $e \in L$. So $t \frown \langle e \rangle \in traces(P)$ but $t \setminus H \frown \langle e \rangle \notin traces(P)$. Because P is divergence-free, by Axiom **F3**, $t \setminus H \frown \langle e \rangle \notin traces(P) \Rightarrow (t \setminus H, \{e\}) \in failures(P)$. So $(t \setminus H, \{e\}) \in failures(P)$ and P clearly fails the first disjunct of Proposition 5.3.2. Hence $\neg RCFNDC(P)$ clearly holds.

On the other hand, suppose that one of s_1 or s_2 contains no H events. Let this be s_j and the other one be s_i. Then $s_i \upharpoonright H \neq \langle\rangle$ and $s_i \setminus H = s_j$. By Proposition 5.3.2, $\neg RCFNDC(P)$ clearly holds. □

Ryan's traces-based formulation of noninterference [Rya91, Equation 1] may be written as follows, adapting a similar characterisation of it from [RG99].

Proposition 5.3.5 (Traces Noninterference)**.** A divergence-free process P satisfies *Traces Noninterference*, written $TNI(P)$, iff

$$\nexists\, s, l \bullet l \in L \wedge \begin{pmatrix} (s \frown \langle l \rangle \in traces(P) \wedge s \setminus H \in traces(P) \wedge s \setminus H \frown \langle l \rangle \notin traces(P)) \vee \\ (s \setminus H \frown \langle l \rangle \in traces(P) \wedge s \in traces(P) \wedge s \frown \langle l \rangle \notin traces(P)) \end{pmatrix}.$$

Traces Noninterference is clearly weaker than RCFNDC, meaning that $RCFNDC(P) \Rightarrow TNI(P)$ but not the reverse. We show that Ryan's Traces Noninterference is equivalent to RCFNDC for deterministic processes by showing that the latter is the refinement-closure of the former (see Definition 5.3.1). We do so via a couple of lemmas.

Lemma 5.3.6. For a divergence-free process P, $\neg RCFNDC(P) \Rightarrow \exists Q \bullet P \sqsubseteq Q \wedge \neg TNI(Q)$.

Proof. So suppose $\neg RCFNDC(P)$ for some divergence-free process P. Without loss of generality, suppose P fails the first disjunct of Proposition 5.3.2 (the second is handled similarly). Then there exists some trace s, such that $s \upharpoonright H \neq \langle\rangle$, and event $l \in L$ such that $s \frown \langle l \rangle \in traces(P) \wedge (s \setminus H, \{l\}) \in failures(P)$. Then, by Lemma A.0.6, P has a divergence-free refinement Q such that $s \frown \langle l \rangle \in traces(Q)$, $s \setminus H \in traces(Q)$ but $s \setminus H \frown \langle l \rangle \notin traces(Q)$. This Q clearly fails the first disjunct of Proposition 5.3.5 and so $\neg TNI(Q)$ holds. □

Lemma 5.3.7. For a divergence-free process P, $\exists Q \bullet P \sqsubseteq Q \wedge \neg TNI(Q) \Rightarrow \neg RCFNDC(P)$.

Proof. So suppose we have a divergence-free process P that has a refinement Q such that $\neg TNI(Q)$. Without loss of generality, suppose Q fails the first disjunct of Proposition 5.3.5 (the second is handled similarly). Then there exists some trace s and event $l \in L$ such that $s \frown \langle l \rangle \in traces(Q) \wedge s \setminus H \in traces(P) \wedge s \setminus H \frown \langle l \rangle \notin traces(Q)$. Then $s \neq s \setminus H$, so $s \upharpoonright H \neq \langle\rangle$. By Axiom **F3**, $(s \setminus H, \{e\}) \in failures(Q)$ and so Q clearly fails the first disjunct of Proposition 5.3.2. Because $P \sqsubseteq Q$, Q's traces and failures are a subset of P's, so P clearly fails this disjunct too. So $\neg RCFNDC(P)$ holds. □

The following theorem, which states that RCFNDC is the refinement-closure of Traces Noninterference, follows straightforwardly from Lemmas 5.3.6 and 5.3.7.

Theorem 5.3.8. For any divergence-free process P,

$$RCFNDC(P) \Leftrightarrow (\forall Q \bullet P \sqsubseteq Q \Rightarrow TNI(Q)).$$

The following corollary shows that RCFNDC coincides for deterministic processes, with any property that is no stronger than Lazy Independence and no weaker than Traces Noninterference.

Corollary 5.3.9. Let ϕ be a property such that for any divergence-free process P, $\mathcal{L}IND(P) \Rightarrow \phi(P) \Rightarrow TNI(P)$. Then for a deterministic process Q,

$$RCFNDC(Q) \Leftrightarrow \phi(Q).$$

Proof. Consider any property ϕ and deterministic process Q as stated in the corollary. Then

$$\mathcal{L}IND(Q) \Rightarrow \phi(Q) \Rightarrow TNI(Q). \tag{5.1}$$

By Lemma 5.3.4, $RCFNDC(Q) \Rightarrow \mathcal{L}IND(Q)$. By Theorem 5.3.8, $RCFNDC(Q) \Leftrightarrow TNI(Q)$. So $TNI(Q) \Rightarrow \mathcal{L}IND(Q)$, and hence $\mathcal{L}IND(Q) \Rightarrow TNI(Q) \Rightarrow \mathcal{L}IND(Q)$. Thus $\mathcal{L}IND(Q) \Leftrightarrow TNI(Q)$. Hence, we may replace each by $RCFNDC(Q)$ in Equation 5.1, giving

$$RCFNDC(Q) \Rightarrow \phi(Q) \Rightarrow RCFNDC(Q).$$

□

Gibson-Robinson has also shown that the refinement-closure of a number of transitive noninterference properties is equivalent to RCFNDC [GR09]. With our results, this covers nearly all of the transitive noninterference properties for CSP processes.

The only transitive noninterference properties, of which we are aware, that do not coincide with RCFNDC for deterministic processes, all coincide with Focardi and Gorrieri's *Nondeterministic Noninterference* [FG95] (NNI).

Definition 5.3.10 (NNI)**.** A divergence-free process P satisfies NNI iff

$$\forall\, s \in traces(P) \bullet s \upharpoonright L \in traces(P).$$

For deterministic processes, NNI coincides with the following refinement-closed property, which is just the first half of the characterisation of RCFNDC from Proposition 5.3.2. Because this property is refinement-closed and can be seen as asserting that the occurrence of high events cannot cause low events to occur, when they otherwise wouldn't have, we refer to it as *Refinement-Closed Noninterference by Causation* (RCNIC).

Definition 5.3.11 (Refinement-Closed Noninterference by Causation)**.** A process satisfies *Refinement-Closed Noninterference by Causation*, written $RCNIC(P)$, iff

$$\begin{aligned} \nexists\, s, l \bullet\ & s \upharpoonright H \neq \langle\rangle \wedge l \in L \wedge \\ & s \mathbin{\hat{}} \langle l \rangle \in traces(P) \wedge (s \setminus H, \{l\}) \in failures(P). \end{aligned}$$

The correspondence between NNI and RCNIC for deterministic processes follows from the latter being the refinement-closure of the former, as shown by Gibson-Robinson [GR09, Theorem 3.5][5]. The correspondence between NNI and the remaining transitive noninterference properties is shown by Gibson-Robinson [GR09, Theorems 3.7 and 3.9].

Summary

We conclude, therefore, that, generally speaking, there exist only two sensible substitutes for *Prop* in Definition 5.2.2 when considering transitive noninterference properties: namely RCFNDC (or, equivalently, Lazy Independence) and RCNIC. These two properties cover the entire literature of which we're aware on transitive noninterference properties for deterministic CSP processes and are refinement-closed, which (as we will see in Sections 5.3.2 and 5.3.3), allows us to derive a refinement check that can be automatically carried out in FDR to test whether they hold for all deterministic componentwise refinements of a system.

[5] The property referred to here as RCNIC is equivalent to Gibson-Robinson's *Weak Operational Noninterference* [GR09].

One may observe that RCNIC detects only when the occurrence of H-events causes some L-event to be able to occur, but doesn't detect when H-events cause some L-event to be refused. Hence, RCNIC is appropriate to apply only for those systems in which the low objects cannot detect the refusal of their events, *i.e.* those in which each low object cannot detect when it has tried to perform some event $l \in L$ but that event has been refused by the other object that it involves. RCFNDC, on the other hand, incorporates RCNIC (as its first disjunct) so not only detects when H-events cause L-events to occur but also when they cause them to be refused. This makes RCFNDC strictly stronger than RCNIC so, being conservative, one might argue that RCFNDC is generally the better choice unless one knows that low objects cannot detect the refusal of their events. We return to this issue later in Section 5.4.2.

Finally, while we have focused only on transitive noninterference properties in this section, it should be noted that similar arguments can be made for *intransitive noninterference properties* (*e.g.* [HY86, Rus92, RG99, vdM08]) for CSP processes; however, a full examination of this kind of property is beyond the scope of this thesis.

5.3.2 Deriving a Testable Characterisation

In the previous subsection, we showed that it suffices to choose some refinement-closed information flow property to substitute for *Prop* in Definition 5.2.2, and that the number of such choices is very limited in practice since most information flow properties coincide for deterministic processes.

In the remainder of this section, we show, given a specific refinement-closed property ϕ to substitute for *Prop*, how to automatically test that ϕ holds for all deterministic componentwise refinements of a system (*i.e.* whether Definition 5.2.2 holds when ϕ replaces *Prop*) using FDR.

We first show how to derive a *testable* characterisation of Definition 5.2.2 when *Prop* is replaced by some some refinement-closed property ϕ. This is required since Definition 5.2.2 is not readily testable, because it quantifies over all deterministic componentwise refinements $System'$ of the system *System* to which it is being applied. A testable characterisation analyses just the traces and failures of *System* in order to determine whether ϕ holds for all $System' \in DCRef(System)$. From this testable characterisation, we can then derive an automatic refinement check as demonstrated later in Section 5.3.3.

We begin by noting that all such ϕ (*e.g.* RCFNDC, RCNIC or even some refinement-closed intransitive noninterference property) share a common structure: any ϕ tests whether the occurrence of events from some set can be influenced, or interfered with, by some other events occurring in the system. We therefore call any such property a *Refinement-Closed Noninterference Property*, which is captured formally by the following definition.

Definition 5.3.12 (Refinement-Closed Noninterference Property)**.** A property ϕ is called a *Refinement-Closed Noninterference Property* when it can be written as follows

$$\phi(P) = \not\exists\, s_1, s_2, e \bullet s_1 \mathbin{^\frown} \langle e \rangle \in \mathit{traces}(P) \land (s_2, \{e\}) \in \mathit{failures}(P) \land \mathit{Pred}(s_1, s_2, e),$$

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

This definition can be understood as follows. It asserts that the system being analysed cannot exhibit any two related behaviours: one $s_1 \mathbin{^\frown} \langle e \rangle$ in which some event e does occur, the other $(s_2, \{e\})$ in which e is refused. The relationship between the two behaviours is captured by *Pred*. The presence of these two behaviours indicates that the occurrence of the event e can be influenced by whatever is different about s_1 and s_2, which is captured by *Pred*. The requirement that $\mathit{Pred}(s_1, s_2, e) \Rightarrow s_1 \upharpoonright \{e\} = s_2 \upharpoonright \{e\}$ simply restricts our attention to those properties for which this difference is not the occurrence, or not, of e itself, since it makes little sense to talk about whether something can be influenced by itself.

For instance, RCNIC asserts that the occurrence of high events from H cannot cause low events from L to occur, by asserting that in any two behaviours s_1 (in which high events occur) and s_2 (in which high events don't occur), any low event that can follow s_1 must always be able to follow s_2 (*i.e.* cannot be refused after s_2). Hence, the occurrence of high events cannot cause low events to occur when they otherwise might not have. *Pred* for RCNIC reflects this, and is

$$e \in L \land s_1 \upharpoonright H \neq \langle\rangle \land s_2 = s_1 \setminus H. \tag{5.2}$$

Similarly, RCFNDC asserts that the occurrence of high events from H cannot influence the occurrence of low events from L, by asserting that in any two behaviours that differ only in that high events have occurred in one but not in the other, if a low event can be performed after one it must always be able to occur after the other. This is reflected in *Pred* for RCFNDC, which is

$$e \in L \land ((s_1 \upharpoonright H \neq \langle\rangle \land s_2 = s_1 \setminus H) \lor (s_2 \upharpoonright H \neq \langle\rangle \land s_1 = s_2 \setminus H)). \tag{5.3}$$

Likewise, *Pred* for Lazy Independence is

$$e \in L \land s_1 \setminus H = s_2 \setminus H. \tag{5.4}$$

A refinement-closed *ipurge*-based definition of intransitive noninterference [HY86, Rus92, vdM08] could also be expressed as a refinement-closed noninterference property by choosing *Pred* to be some suitable encoding of the intransitive purge function. Roscoe and Goldsmith's formulation of intransitive noninterference [RG99] is essentially the conjunction of multiple

applications of a variation of Lazy Independence (where H and L need not partition the system's alphabet) and therefore should also be expressible as a refinement-closed noninterference property.

We show here how to take any refinement-closed noninterference property ϕ and to produce a testable characterisation of the property that applies ϕ to all deterministic componentwise refinements of a system.

Observe that any ϕ examines pairs of behaviours, $s_1 \frown \langle e \rangle$ and $(s_2, \{e\})$, of a system *System* and that applying ϕ to all of *System*'s deterministic componentwise refinements $System' \in DCRef(System)$ is equivalent to having ϕ examine all pairs $s_1 \frown \langle e \rangle$ and $(s_2, \{e\})$ that can be exhibited by all such $System' \in DCRef(System)$. We say that two behaviours $s_1 \frown \langle e \rangle$ and $(s_2, \{e\})$ that can be exhibited by a deterministic componentwise refinement $System'$ of *System* are *consistent*.

Definition 5.3.13 (Consistency)**.** Two behaviours $s_1 \frown \langle e \rangle$ and $(s_2, \{e\})$ of an alphabetised parallel composition $Sys = \|_{i \in \{1,\ldots,n\}} (P_i, A_i)$ are *consistent* iff

$$\exists\, Sys' \in DCRef(System) \bullet s_1 \frown \langle e \rangle \in traces(Sys') \wedge (s_2, \{e\}) \in failures(Sys').$$

In this case, we write $Con(s_1 \frown \langle e \rangle, (s_2, \{e\}))$. We say that $s_1 \frown \langle e \rangle$ and $(s_2, \{e\})$ are *inconsistent* otherwise.

So consider some refinement-closed noninterference property ϕ, where (following Definition 5.3.12)

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

The property obtained by substituting ϕ for *Prop* in Definition 5.2.2 then holds for some system *Sys* iff

$$\begin{aligned}
&\forall\, Sys' \in DCRef(Sys) \bullet \phi(Sys') \\
&\quad \Leftrightarrow \quad \forall\, Sys' \in DCRef(Sys) \bullet \not\exists\, s_1, s_2, e \bullet s_1 \frown \langle e \rangle \in traces(Sys') \wedge \\
&\qquad\qquad (s_2, \{e\}) \in failures(Sys') \wedge Pred(s_1, s_2, e) \\
&\quad \Leftrightarrow \quad \not\exists\, Sys' \in DCRef(Sys), s_1, s_2, e \bullet s_1 \frown \langle e \rangle \in traces(Sys') \wedge \\
&\qquad\qquad (s_2, \{e\}) \in failures(Sys') \wedge Pred(s_1, s_2, e) \\
&\quad \Leftrightarrow \quad \not\exists\, s_1, s_2, e \bullet \exists\, Sys' \in DCRef(Sys) \bullet s_1 \frown \langle e \rangle \in traces(Sys') \wedge \\
&\qquad\qquad (s_2, \{e\}) \in failures(Sys') \wedge Pred(s_1, s_2, e) \wedge \\
&\qquad\qquad s_1 \frown \langle e \rangle \in traces(Sys) \wedge (s_2, \{e\}) \in failures(Sys).
\end{aligned}$$

This is equivalent to the property ϕ_{Con} defined as

$$\begin{aligned}
\phi_{Con}(Sys) = \not\exists\, s_1, s_2, e \bullet\ & s_1 \frown \langle e \rangle \in traces(Sys) \wedge \\
& (s_2, \{e\}) \in failures(Sys) \wedge \\
& Pred(s_1, s_2, e) \wedge Con(s_1 \frown \langle e \rangle, (s_2, \{e\})).
\end{aligned} \tag{5.5}$$

Thus, $\phi_{Con}(Sys) \Leftrightarrow \forall\, Sys' \in DCRef(Sys) \bullet \phi(Sys')$.

To produce a testable characterisation of ϕ_{Con}, we need to derive some semantic condition that can be applied to determine whether two behaviours $s_1 \frown \langle e \rangle$ and $(s_2, \{e\})$ are consistent, *i.e.* can be exhibited by a composition of deterministic refinements of the system's components. We therefore need a way to test whether any two $s_1 \frown \langle e \rangle$ and $(s_2, \{e\})$ can be exhibited by the system without any of its components (that take part in the two behaviours) having to act nondeterministically.

The CSP laws for alphabetised parallel composition [Ros97] imply that an alphabetised parallel composition $Sys = \|_{i \in \{1,\ldots,n\}}(P_i, A_i)$ can exhibit both of the behaviours $s_1 \frown \langle e \rangle$ and $(s_2, \{e\})$ iff:

- All components involved in performing the traces $s_1 \frown \langle e \rangle$ and s_2 are able to perform their events at the appropriate time, *i.e.*

$$\forall\, i \in \{1, \ldots, n\} \bullet (s_1 \frown \langle e \rangle) \upharpoonright A_i \in traces(P_i) \land s_2 \upharpoonright A_i \in traces(P_i),$$

- 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.*

$$\exists\, j \in \{1, \ldots, n\} \bullet e \in A_j \land (s_2 \upharpoonright A_j, \{e\}) \in failures(P_j).$$

A component acts nondeterministically when it both performs and refuses some event d after some trace t (see Definition 2.1.1). The only event that is refused as part of $s_1 \frown \langle e \rangle$ and $(s_2, \{e\})$ occurring is the event e. Hence, the only components that can act nondeterministically while these two behaviours are being exhibited are those components P_j that can be involved in refusing e, *i.e.* those P_j for whom e is in their alphabet A_j.

One of these components P_j (for whom $e \in A_j$) *must* act nondeterministically while contributing to $s_1 \frown \langle e \rangle$ and $(s_2, \{e\})$, then, if:

- It is the only component that can refuse e at the appropriate time (*i.e.* it is the only component for whom $(s_2 \upharpoonright A_j, \{e\}) \in failures(P_j)$), and,

- the trace after which it refuses e ($s_2 \upharpoonright A_j$) is the same one after which it must also perform e as part of $s_1 \frown \langle e \rangle$, *i.e.* $s_1 \upharpoonright A_j = s_2 \upharpoonright A_j$.[6]

This idea is captured in the following testable characterisation of consistency, which we call *apparent consistency*. It asserts that two behaviours are apparently consistent when there exists at least one component P_j that can be involved in refusing e (*i.e.* for whom $e \in A_j$) at the appropriate time (so that $(s_2 \upharpoonright A_j, \{e\}) \in failures(P_j)$), without having to act nondeterministically (so $s_1 \upharpoonright A_j \neq s_2 \upharpoonright A_j$).

[6] Note that *e.g.* $s_2 \upharpoonright A_j \frown \langle e \rangle \not\leq s_1 \upharpoonright A_j$ because $s_1 \upharpoonright \{e\} = s_2 \upharpoonright \{e\}$ and $e \in A_j$.

Definition 5.3.14 (Apparent Consistency)**.** Two behaviours, $s_1 \frown \langle e \rangle$ and $(s_2, \{e\})$, of an alphabetised parallel composition $Sys = \|_{i \in \{1,\ldots,n\}}(P_i, A_i)$, are *apparently consistent* iff

$$\exists\, j \in \{1, \ldots, n\} \bullet e \in A_j \wedge s_1 \upharpoonright A_j \neq s_2 \upharpoonright A_j \wedge (s_2 \upharpoonright A_j, \{e\}) \in \mathit{failures}(P_j).$$

We now prove that apparent consistency is equivalent to consistency, for pairs of behaviours that violate refinement-closed noninterference properties. We do so via couple of lemmas. We first show that apparent consistency is necessary for consistency.

Lemma 5.3.15 (Apparent consistency is necessary for consistency)**.** If two behaviours that violate a refinement-closed noninterference 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, $s_1 \frown \langle e \rangle$ and $(s_2, \{e\})$, of a system $Sys = \|_{i \in \{1,\ldots,n\}}(P_i, A_i)$ that violate a refinement-closed noninterference property. Then by Definition 5.3.12, $s_1 \upharpoonright \{e\} = s_2 \upharpoonright \{e\}$. Suppose further that these behaviours are not apparently consistent, *i.e.* by Definition 5.3.14 that

$$\nexists\, j \in \{1, \ldots, n\} \bullet e \in A_j \wedge s_1 \upharpoonright A_j \neq s_2 \upharpoonright A_j \wedge (s_2 \upharpoonright A_j, \{e\}) \in \mathit{failures}(P_j).$$

We have that $\forall\, i \in \{1, \ldots, n\} \bullet (s_1 \frown \langle e \rangle) \upharpoonright A_i \in \mathit{traces}(P_i)$ and for some $j \in \{1, \ldots, n\}$, $e \in A_j \wedge (s_2 \upharpoonright A_j, \{e\}) \in \mathit{failures}(P_j)$. So we know that for all such P_j, $s_1 \upharpoonright A_j = s_2 \upharpoonright A_j$. Since $e \in A_j$, we also have that $(s_1 \frown \langle e \rangle) \upharpoonright A_j = s_1 \upharpoonright A_j \frown \langle e \rangle$. Hence

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

This proves that for the two behaviours to arise, all such P_j must act nondeterministically. Hence, there exists no $Sys' \in \mathit{DCRef}(Sys)$ that can exhibit both behaviours and so by Definition 5.3.13, they must be inconsistent. □

We now show that apparent consistency is also sufficient for consistency.

Lemma 5.3.16 (Apparent consistency is sufficient for consistency)**.** If two behaviours that violate a refinement-closed noninterference property are apparently consistent, then they are consistent.

Proof. We prove this by showing how to construct a deterministic componentwise refinement that exhibits both behaviours.

Suppose we have two behaviours, $s_1 \frown \langle e \rangle$ and $(s_2, \{e\})$, of a system $Sys = \|_{i \in \{1,\ldots,n\}}(P_i, A_i)$ that violate a refinement-closed noninterference property. Then by Definition 5.3.12, $s_1 \upharpoonright \{e\} = s_2 \upharpoonright \{e\}$. Suppose further that these behaviours are apparently consistent, *i.e.* by Definition 5.3.14 that for some $j \in \{1, \ldots, n\}$, $e \in A_j \wedge (s_2 \upharpoonright A_j, \{e\}) \in \mathit{failures}(P_j) \wedge s_1 \upharpoonright A_j \neq s_2 \upharpoonright A_j$.

Now, by Definition 5.3.13, to show that these two behaviours are consistent, we seek a deterministic componentwise refinement $Sys' \in DCRef(Sys)$ that can exhibit both of them. From Definition 5.2.1, Sys' is equivalent to some composition $\|_{i\in\{1,\ldots,n\}}(Q_i, A_i)$ where $\forall\, i \in \{1,\ldots,n\} \bullet P_i \sqsubseteq Q_i \wedge det(Q_i)$. We show that such an Sys' exists by showing that such a set of processes Q_i exists whose composition can exhibit both behaviours. We do this by showing how to construct each Q_i from the corresponding P_i.

We require that $\forall\, i \in \{1,\ldots,n\} \bullet (s_1 \frown \langle e\rangle) \upharpoonright A_i \in traces(Q_i) \wedge s_2 \upharpoonright A_i \in traces(Q_i)$, and $(s_2 \upharpoonright A_j, \{e\}) \in failures(Q_j)$, for j above.

For the remaining Q_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 P_i. That is, for each $i \neq j$, we define Q_i to be the process that has the same stable failures as P_i, except that whenever P_i can perform a trace $t \frown \langle d\rangle$, Q_i cannot have any refusal (t, X) where $d \in X$. Lemma A.0.4, proves that such a process exists for all P_i.

We construct Q_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 A.0.6, shows that for any P_j, one can always do this to arrive at a process, R_j, that refines P_j and for which $s \upharpoonright A_2 \frown \langle e\rangle \notin traces(R_j)$. Hence, by Axiom **F3**, 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 $s_1 \frown \langle e\rangle \in traces(R_j)$, since it wasn't removed when forming R_j.

We then simply take Q_j to be the deterministic trace-equivalent refinement of R_j. Q_j is thus guaranteed to have the stable failure $(s_2 \upharpoonright A_j, \{e\})$. It will also have the trace $s_1 \frown \langle e\rangle \upharpoonright A_j$. Hence, the composition $\|_{i\in\{1,\ldots,n\}}(Q_i, A_i)$ can exhibit both behaviours. So they must be consistent. □

The following theorem follows directly from Lemmas 5.3.15 and 5.3.16.

Theorem 5.3.17. Two behaviours that violate a refinement-closed noninterference property are apparently consistent iff they are consistent.

This allows us to take a refinement-closed noninterference property ϕ and give a testable characterisation of the property obtained by substituting ϕ for *Prop* in Definition 5.2.2. Recall that this is equivalent to the property ϕ_{Con}, defined by Equation 5.5, which makes use of the predicate *Con* (see Definition 5.3.13) to consider only consistent pairs of behaviour. This testable characterisation is then obtained by replacing the use of *Con* in the definition of ϕ_{Con} by its testable counterpart, namely apparent consistency from Definition 5.3.14. We call this testable characterisation of ϕ_{Con} the *Weakened Counterpart for Compositions* of ϕ, which is defined as follows.

Definition 5.3.18 (Weakened Refinement-Closed Noninterference Property for Compositions)**.** The *weakened counterpart for compositions* of a

refinement-closed noninterference property ϕ, where

$$\phi(P) = \not\exists\, s_1, s_2, e \bullet s_1 \mathbin{\hat{}} \langle e \rangle \in traces(P) \land (s_2, \{e\}) \in failures(P) \land Pred(s_1, s_2, e),$$

is the property $W\phi$, defined for alphabetised parallel compositions $Sys = \left\|_{i \in \{1,\ldots,n\}} (P_i, A_i)\right.$ as

$$W\phi(Sys) = \not\exists\, s_1, s_2, e \bullet s_1 \mathbin{\hat{}} \langle e \rangle \in traces(Sys) \land (s_2, \{e\}) \in failures(Sys) \land Pred(s_1, s_2, e) \land \exists\, j \in \{1, \ldots, n\} \bullet e \in A_j \land s_1 \upharpoonright A_j \neq s_2 \upharpoonright A_j \land (s_2 \upharpoonright A_j, \{e\}) \in failures(P_j).$$

We call $W\phi$ a *weakened refinement-closed noninterference property* for short.

The following corollary states that the property $W\phi$ is equivalent to the property obtained by substituting ϕ for *Prop* in Definition 5.2.2.

Corollary 5.3.19. Let ϕ be a refinement-closed noninterference property and $W\phi$ be its weakened counterpart for compositions. Then for any alphabetised parallel composition *Sys*

$$W\phi(Sys) \Leftrightarrow \forall\, Sys' \in DCRef(Sys) \bullet \phi(Sys').$$

Substituting the definition of *Pred* for RCFNDC (see Equation 5.3) into the definition of $W\phi$ above therefore gives us a testable property that corresponds to applying RCFNDC to all deterministic componentwise refinements of a system. We call this property *Weakened RCFNDC for Compositions* or *Weakened RCFNDC* for short.

Definition 5.3.20. An alphabetised parallel composition $Sys = \left\|_{i \in \{1,\ldots,n\}} (P_i, A_i)\right.$ satisfies *Weakened RCFNDC for Compositions*, written $WRCFNDC(Sys)$, iff

$$\not\exists\, s, l \bullet s \upharpoonright H \neq \langle\rangle \land l \in L \land \left(\begin{pmatrix} s \mathbin{\hat{}} \langle l \rangle \in traces(Sys) \land s \setminus H \in traces(Sys) \land \\ \exists\, i \bullet l \in A_i \land s \upharpoonright A_i \neq s \setminus H \upharpoonright A_i \land (s \setminus H \upharpoonright A_i, \{l\}) \in failures(P_i) \end{pmatrix} \right.$$
$$\left. \lor \begin{pmatrix} s \setminus H \mathbin{\hat{}} \langle l \rangle \in traces(Sys) \land s \in traces(Sys) \land \\ \exists\, i \bullet l \in A_i \land s \upharpoonright A_i \neq s \setminus H \upharpoonright A_i \land (s \upharpoonright A_i, \{l\}) \in failures(P_i) \end{pmatrix} \right).$$

Corollary 5.3.21. For any alphabetised parallel composition *Sys*

$$WRCFNDC(Sys) \Leftrightarrow \forall\, Sys' \in DCRef(Sys) \bullet RCFNDC(Sys').$$

The same can be done for RCNIC (the *Pred* for which is given by Equation 5.2), in which case we obtain the property that corresponds to Weakened RCFNDC with the second disjunct above removed.

5.3.3 Deriving an Automatic Test

We now show how to construct a refinement check to allow FDR to automatically test weakened refinement-closed noninterference properties, such as Weakened RCFNDC above. For a specific weakened refinement-closed noninterference property $W\phi$, its refinement check can be constructed by either adapting any existing check for ϕ, or from scratch by applying the basic ideas outlined in this section with previous ones for expressing properties as refinement checks (see *e.g.* [Ros05, Low09]). For the sake of brevity, we will show how to take the former approach. However, these ideas can be easily extended to produce tests for weakened refinement-closed noninterference properties $W\phi$ when there is no pre-existing test for ϕ.

Testing Refinement-Closed Noninterference Properties

We begin with an overview of how refinement checks for refinement-closed noninterference properties are constructed (see [Low09]).

Refinement-closed noninterference 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 [Low09] has shown that all binary failures properties can be expressed in the form of CSP refinement tests; most of these tests are finite state, enabling them to be automatically tested by FDR. Any such property ϕ applied to a process *Sys* can be expressed in terms of a CSP refinement test that runs two copies of the process *Sys* in a test harness, $Harness(Sys)$, 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

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

for some deterministic *scheduler* process, *Sched*, that allows the two copies of *Sys* to exhibit all behaviours that could lead to violations of the property in question. Each copy of *Sys* 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 *Sys* then by testing whether

$$Spec \sqsubseteq_F Harness(Sys).$$

Lowe [Low09] provides a method for deriving both *Spec* and *Harness* to ensure they are correct by construction that, although not complete, works for all cases considered to date, including for RCFNDC.

We consider how such tests can be modified in order to allow weakened refinement-closed noninterference properties to be automatically tested using FDR. We use the test for RCFNDC as an example, but the general technique can be applied with equal success to other refinement-closed noninterference properties, such as RCNIC, in order to express their weakened counterparts as refinement tests. The following is a slight adaptation of the *Spec* and *Sched* processes derived in [Low09] for RCFNDC.

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

The test works as follows. *Sched* allows the left copy of *Sys* to perform events in H and L. The right copy of *Sys* 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 *Sys*. 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 *Sys*, **(2)** the other copy cannot refuse it. This corresponds exactly to the definition of RCFNDC (see Proposition 5.3.2) for *Sys*, which is equivalent to

$$\begin{aligned}
&\not\exists\, s_1, s_2, e \bullet \\
&\quad (\mathbf{1})\ s_1 \mathbin{^\frown} \langle e \rangle \in traces(Sys) \land e \in L \land (\mathbf{2})\ (s_2, \{e\}) \in failures(Sys) \land \\
&\quad (\mathbf{0})\ ((s_1 \upharpoonright H \neq \langle\rangle \land s_2 = s_1 \setminus H) \lor (s_2 \upharpoonright H \neq \langle\rangle \land s_1 = s_2 \setminus H)).
\end{aligned}$$

Testing Weakened Refinement-Closed Noninterference Properties

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

We begin by observing that we can rewrite a weakened refinement-closed noninterference property $W\phi$ (see Definition 5.3.18) equivalently (for alphabetised parallel compositions $Sys = \|_{i \in \{1,\ldots,n\}}(P_i, A_i)$) as

$$\begin{aligned}
W\phi(Sys) = \not\exists\, s_1, s_2, e \bullet\ & s_1 \mathbin{^\frown} \langle e \rangle \in traces(Sys) \land s_2 \in traces(Sys) \land \\
& Pred(s_1, s_2, e) \land \\
& \exists\, j \in \{1, \ldots, n\} \bullet e \in A_j \land s_1 \upharpoonright A_j \neq s_2 \upharpoonright A_j \land \\
& \qquad (s_2 \upharpoonright A_j, \{e\}) \in failures(P_j).
\end{aligned}$$

Hence, weakened refinement-closed noninterference 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 P_j.

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 system *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 P_i of the system *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, $Sys = \|_{i \in \{1,\ldots,n\}} (P_i, A_i)$, we can create a process, *WSys*, that allows us to observe both system-level traces and individual component behaviours as depicted in Snippet 5.2.

$$WSys = \|_{i \in \{1,\ldots,n\}} (P_i [\![\mathsf{sys}.x, \mathsf{cmp}.i.x / x, x]\!], \{ \mathsf{sys}.x, \mathsf{cmp}.i.x \mid x \in A_i \}).$$

Snippet 5.2: Observing system-level traces and individual component behaviours.

Here sys and cmp are fresh channels over which *WSys* performs system-level and individual component events respectively. $P_i [\![\mathsf{sys}.x, \mathsf{cmp}.i.x / x, x]\!]$ is the process that can perform either of the events $\mathsf{sys}.x$ or $\mathsf{cmp}.i.x$, whenever P_i can perform the event x. The alphabetised parallel composition forces all of the transformed P_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 $Sys = \|_{i \in \{1,\ldots,n\}} (P_i, A_i)$ as depicted in Snippet 5.3.

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

Snippet 5.3: A test harness for weakened refinement-closed noninterference properties.

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 pro-

cess, *Spec*, for a (non-weakened) refinement-closed noninterference property applied to some process *Sys*, is constructed so that whenever the left (respectively right) copy of *Sys* performs a trace $s_1 \hat{} \langle e \rangle$ and the right (respectively left) copy of *Sys* performs the related trace s_2, *Spec* never refuses the event right.e (respectively left.e). Hence, it will be refined by $Harness(Sys)$ if and only if the right (respectively left) copy of *Sys* 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, its *critical* states.

For a weakened refinement-closed noninterference property applied to some composition, $Sys = \|_{i \in \{1,\ldots,n\}}(P_i, A_i)$, 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*, P_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, $\mathcal{S}$, of components that have performed different traces so far, *i.e.* for whom $s_1 \upharpoonright A_i \neq s_2 \upharpoonright A_i$; $\mathcal{S}$ is initially empty.
- For each system-level event that is performed, determine which components have now performed different traces, and add those to $\mathcal{S}$.
- The critical states in which *Spec* could not refuse the event right.e (respectively left.e) may be expressed as right.$e \rightarrow Q$ (respectively left.$e \rightarrow Q$) for some process Q. Have *WSpec* instead do

  $$\mathsf{right.sys}.e \rightarrow Q \triangleright NR(\mathsf{right}, \mathcal{S}, e)$$

  (respectively $\mathsf{left.sys}.e \rightarrow Q \triangleright NR(\mathsf{left}, \mathcal{S}, e)$), where

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

  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.$i.e$ (respectively left.cmp.$i.e$) for all P_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.$i.e$ (respectively left.cmp.$i.e$) 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 (see Lemma A.0.4 in Appendix A) of *WSpec*, which can usually be derived syntactically as shown below. The test is carried out against a system *Sys*, then, by testing the refinement

$$WSpec \sqsubseteq_F WHarness(Sys).$$

In order to apply the recipe to develop *WSpec* from *Spec*, we simply need to determine how to update the set $\mathcal{S}$ 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, $\mathcal{S}$ needs to be updated only for each H event, h, that is performed by simply adding all of the P_i for whom $h \in A_i$; we denote this set by $cmpsWith(h)$, where $cmpsWith(h) = \{P_i \mid i \in \{1, \ldots, n\} \wedge h \in A_i\}$. This leads to the definition of *WSpec* for Weakened RCFNDC, which appears in Snippet 5.4.

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

Snippet 5.4: The specification for testing Weakened RCFNDC.

As stated earlier, *WSched* is the deterministic trace-equivalent refinement of *WSpec*. It can be derived syntactically from *WSpec* and appears in Snippet 5.5.

5.4 Applying the Test

We have shown how to define information flow security for object-capability patterns modelled in CSP as weakened refinement-closed noninterference properties, such as Weakened RCFNDC, and how to construct refinement checks to allow these properties to be automatically tested by FDR. In particular, we now have a refinement check, namely that above for Weakened RCFNDC, that can be applied to CSP models of object-capability

$$
\begin{aligned}
&WSched = \\
&\quad \mathsf{left.sys}?h : H \rightarrow WSched'(cmpsWith(h)) \\
&\quad \Box\ \mathsf{left.sys}?l : L \rightarrow \mathsf{right.sys}.l \rightarrow WSched \\
&\quad \Box\ \mathsf{right.sys}?l : L \rightarrow \mathsf{left.sys}.l \rightarrow WSched, \\
&WSched'(\mathcal{S}) = \\
&\quad \mathsf{left.sys}?h : H \rightarrow WSched'(\mathcal{S} \cup cmpsWith(h)) \Box \\
&\quad \mathsf{left.sys}?l : L \rightarrow (\mathsf{right.sys}.l \rightarrow WSched'(\mathcal{S}) \Box NR(\mathsf{right}, \mathcal{S}, l)) \\
&\quad \Box\ \mathsf{right.sys}?l : L \rightarrow (\mathsf{left.sys}.l \rightarrow WSched'(\mathcal{S}) \Box NR(\mathsf{left}, \mathcal{S}, l)).
\end{aligned}
$$

Snippet 5.5: The scheduler for testing Weakened RCFNDC.

patterns to reason about their information flow properties. In this section, we demonstrate its application for reasoning about how an implementation of the Data-Diode pattern, depicted in Figure 5.1, allows information to flow.

5.4.1 Modelling the Data-Diode Implementation

We analyse an implementation of the Data-Diode from [Mil06, Figure 11.2]. For brevity, we consider just the more general concurrent context in which each object executes with its own thread of control. We begin by considering the Data-Diode pattern instantiated in the system depicted in Figure 5.1.

Recall that in this system, High and Low are each arbitrary objects that have access to high- and low-classification data respectively. The sets *HighData* and *LowData*, each of which is a subset of the set *Data* of all data in the system, contain the high- and low-classification data respectively. The behaviours of High and Low are given as indicated earlier in Snippet 5.1.

A *data-diode* is an object that has two facets, a read-facet and a write-facet[7]. It stores a single datum and begins life holding some initial value. Invoking its read-facet causes it to return its current contents. Invoking its write-facet with a *Data* argument causes it to replace its current contents with the argument. We model a data-diode with read-facet *readme* and write-facet *writeme* that initially contains the datum *val* from the set *Data* as the CSP process $ADataDiode(readme, writeme, val)$, defined as follows.

$$
\begin{aligned}
&ADataDiode(readme, writeme, val) = \\
&\quad ?from : Capability - \{readme, writeme\}!readme!\mathsf{Call}!\mathsf{null} \rightarrow \\
&\qquad readme!from!\mathsf{Return}!val \rightarrow ADataDiode(readme, writeme, val) \Box \\
&\quad ?from : Capability - \{readme, writeme\}!writeme!\mathsf{Call}?newVal : Data \rightarrow \\
&\qquad writeme!from!\mathsf{Return}!\mathsf{null} \rightarrow ADataDiode(readme, writeme, newVal).
\end{aligned}
$$

[7] It is unclear whether the read and write interfaces should be implemented as facets of a single object or as forwarding objects of a composite object. We choose the former option at this point and will explore the latter in Section 5.4.3.

Observe that this process passes *Data* items only, refusing all *Capability* arguments.

The behaviour of DataDiode is as one would expect, given that it starts life holding no datum (*i.e.* holding the null value null), namely

$$behaviour(\mathsf{DataDiode}) = ADataDiode(\mathsf{DDReader}, \mathsf{DDWriter}, \mathsf{null}).$$

The facets of each object are as one would expect, namely $facets(\mathsf{DataDiode}) = \{\mathsf{DDReader}, \mathsf{DDWriter}\}$ and $facets(other) = \{other\}$ for $other \neq \mathsf{DataDiode}$. To complete the system, it remains to define the sets *HighData*, *LowData* and *Data*. For now, we will instantiate *HighData* and *LowData* simply as disjoint singleton sets and allow them to partition the total set *Data* of data. So let $HighData = \{\mathsf{HighDatum}\}$, $LowData = \{\mathsf{LowDatum}\}$ and $Data = HighData \cup LowData$. Letting $Object = \{\mathsf{High}, \mathsf{DataDiode}, \mathsf{Low}\}$, the system is then modelled by the object-capability system $(Object, behaviour, facets, Data)$ and captured by the process $System = \|_{o \in Object}(behaviour(o), \alpha(o))$ per Definition 2.3.1.

5.4.2 Analysing the Data-Diode Implementation

As explained earlier, we can easily express the notion that DataDiode doesn't allow Low to obtain high data by overt means as a safety property. We define the set X of events that represent Low acquiring any *HighData*. $X = \{f.l.op.d \mid f \in Capability, l \in facets(\mathsf{Low}), op \in \{\mathsf{Call}, \mathsf{Return}\}, d \in HighData\}$. We then simply test that no X-events can occur in *System* by testing whether $CHAOS_{\Sigma - X} \sqsubseteq_T System$. FDR reveals that this test holds. We conclude that Low cannot obtain any *HighData* overtly in this system.

Having ruled out this overt information flow, we will now apply Weakened RCFNDC to *System* to test whether information can flow from High to Low by covert means. To do so, we must define the sets H and L of high and low events respectively. Recall that Weakened RCFNDC tests whether the occurrence of high events in H can affect the occurrence of events in L (and so influence the low objects that perform L-events). H and L must partition the effective alphabet of *System*, meaning that H and L must be disjoint and that *System* must perform no events outside of $H \cup L$.

Let $H = \{|h.\mathsf{DDReader}, \mathsf{DDReader}.h, h.h' \mid h, h' \in facets(\mathsf{High})|\}$ denote the set of events that represent High interacting with itself and DDReader. Similarly let $L = \{|l.\mathsf{DDWriter}.\mathsf{Call}.arg, \mathsf{DDWriter}.l.\mathsf{Return}.\mathsf{null}, l.l' \mid l, l' \in facets(\mathsf{Low})|\}$ denote the set of events representing Low interacting with itself and DDWriter. Then these sets are clearly disjoint. FDR reveals that $STOP \sqsubseteq_T System \setminus (H \cup L)$, so *System* can perform no events outside of $H \cup L$. This implies, for example, that neither High nor Low can obtain a capability to the other. So H and L partition the effective alphabet of *System*.

Applying the refinement check for Weakened RCFNDC from the previous section to *System* with these definitions of H and L, using FDR, tests that

High's interactions with itself and DataDiode cannot interfere with Low, and so tests whether DataDiode provides a covert channel from High to Low.

Performing the test in FDR reveals that Weakened RCFNDC doesn't hold for *System*. FDR returns the following stable-failure that can be exhibited by the test-harness $WHarness(System)$ (see Snippet 5.3) but not by the specification *WSpec* (see Snippet 5.4) for Weakened RCFNDC.

$$\begin{pmatrix}\langle \mathsf{left.sys.High.DDReader.Call.null},\ \mathsf{right.sys.Low.DDWriter.Call.LowDatum}\rangle,\\ \{\mathsf{left.cmp.DataDiode.Low.DDWriter.Call.LowDatum}\}\end{pmatrix}$$

This stable-failure indicates that the right copy of *System* can perform the trace $\langle$Low.DDWriter.Call.LowDatum$\rangle$, while the left copy can exhibit the stable-failure $(\langle$High.DDReader.Call.null$\rangle, \{$Low.DDWriter.Call.LowDatum$\})$ because DataDiode refuses the event Low.DDWriter.Call.LowDatum after the event High.DDReader.Call.null has occurred. We see that the second disjunct from Definition 5.3.20 of Weakened RCFNDC is clearly violated, then, if we let $s = \langle$High.DDReader.Call.null$\rangle$, $l =$ Low.DDWriter.Call.LowDatum and $P_i =$ DataDiode.

This counter-example indicates that (in the right copy of *System*) initially Low can invoke DDWriter but that (in the left copy) if High invokes DDReader, it can cause Low's invocation to be refused. This occurs because DataDiode cannot service requests from High and Low at the same time. This constitutes a clear covert channel, since High can signal to Low by invoking DDReader which alters whether Low's invocation is accepted.

We note that Low may be unable to observe this covert channel in some object-capability systems, *e.g.* those in which a sender of a message is undetectably blocked until the receiver is ready to receive it. In this kind of system, Low is unable to detect the refusal of his own events by the other objects that partake in those events. Hence, in this case Low is unable to detect when High causes Low events to be refused and can only detect when High causes Low events to occur. For this kind of system, one might wish to apply an information flow property that detects only when High can cause Low events to occur by using RCNIC (Definition 5.3.11) in place of RCFNDC, and so apply RCNIC's weakened counterpart for compositions, namely Weakened RCNIC (as given by Definition 5.3.18), rather than Weakened RCFNDC. Recall that Weakened RCNIC is equivalent to Weakened RCFNDC with the second disjunct of Definition 5.3.20 removed. Likewise, the refinement check for Weakened RCNIC is a simplification of that for Weakened RCFNDC (in which the scheduler *WSched* never allows the right copy of the system to choose the next L-event to be performed).[8] However, we choose to make the conservative assumption that this counter-example represents a valid fault that needs to be corrected.

[8] Performing this check for Weakened RCNIC in FDR reveals that it holds for *System*.

5.4.3 Fixing the Data-Diode Implementation

Correcting the fault here involves modifying the data-diode implementation so that its interfaces for writing and reading, DDWriter and DDReader, can be used simultaneously. We do so by promoting these interfaces from being facets of a single process to existing as individual processes in their own right. These processes simply act now as proxies that forward invocations to the facets of an underlying *ADataDiode* process, as depicted in Figure 5.2.

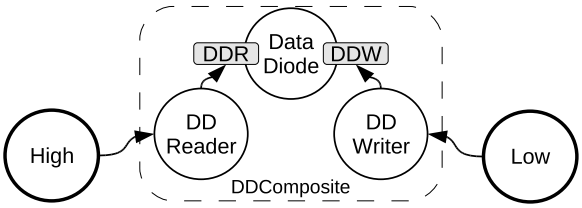

Figure 5.2: An improved Data-Diode implementation.

The behaviour of a proxy *me* that forwards invocations it receives using the capability *target* is given by the following process $AProxy(me, target)$

$$
\begin{aligned}
&AProxy(me, target) = \\
&\quad ?from : Capability - \{me\}!me!\mathsf{Call}?arg : Data \cup \{\mathsf{null}\} \rightarrow \\
&\quad me!target!\mathsf{Call}!arg \rightarrow target!me!\mathsf{Return}?res : Data \cup \{\mathsf{null}\} \rightarrow \\
&\quad me!from!\mathsf{Return}!res \rightarrow AProxy(me, target).
\end{aligned}
$$

The data-diode is now a *composite* of three entities, DDReader, DDWriter and DataDiode, and as such is referred to as DDComposite. We model the system depicted in Figure 5.2 as an object-capability system comprising the objects from $Object = \{\mathsf{High}, \mathsf{DDComposite}, \mathsf{Low}\}$, where $facets(\mathsf{DDComposite}) = \{\mathsf{DDReader}, \mathsf{DDWriter}, \mathsf{DDR}, \mathsf{DDW}\}$ and, letting $R = \{|\mathsf{DDReader}.x, x.\mathsf{DDReader} \mid x \in facets(\mathsf{DDComposite}) - \{\mathsf{DDReader}\}|\}$, $W = \{|\mathsf{DDWriter}.x, x.\mathsf{DDWriter} \mid x \in facets(\mathsf{DDComposite}) - \{\mathsf{DDWriter}\}|\}$, $DD = ADataDiode(\mathsf{DDR}, \mathsf{DDW}, \mathsf{null})$ and the other definitions be as before,

$$
\begin{aligned}
&behaviour(\mathsf{DDComposite}) = \\
&\left((AProxy(\mathsf{DDReader}, \mathsf{DDR}) \underset{R}{\parallel} DD) \underset{W}{\parallel} AProxy(\mathsf{DDWriter}, \mathsf{DDW}) \right) \setminus (R \cup W).
\end{aligned}
$$

DDComposite is formed by taking the two proxies, DDReader and DDWriter, and composing them in parallel with DataDiode, whose read- and write-interfaces are now DDR and DDW respectively. Notice that we then hide the internal communications within DDComposite since these are not visible to its outside environment and it is unclear how to divide these events between the sets H and L.

Performing the appropriate traces refinement checks in FDR reveal that Low cannot acquire any *HighData*, and that *System* can perform no events

outside of $H \cup L$, as before. FDR reveals that Weakened RCFNDC holds for *System*. Hence, we are unable to detect any covert channels in this model of the improved Data-Diode implementation.

5.5 Generalising Information Flow Analyses

We have shown how to reason about the information flow properties of object-capability patterns deployed in small, fixed-sized systems. We now show how to adapt the approach taken in Chapter 4 to allow us to generalise these information flow analyses to arbitrary-sized systems.

As an example, we will show how to generalise the Data-Diode analysis to all systems that have the form of Figure 5.3, and have arbitrary *HighData* and *LowData*. Here, the objects within each cloud can be interconnected in any way whatsoever; however, the only capability to an object outside of the high object cloud that each high object may possess is DDReader. The same is true for the low objects and DDWriter. This figure captures all systems containing an arbitrary number of high and low objects and, thus, all those in which each object may create arbitrary numbers of others that share its security level.

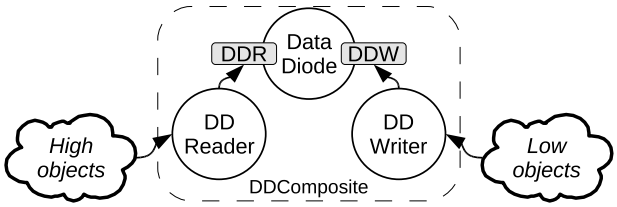

Figure 5.3: Generalising the Data-Diode analysis.

Recall that the approach we took in Chapter 4 had two steps. The first step of this approach was to choose an arbitrary large system to which we wanted our results to generalise, such as one of the systems captured by Figure 5.3, and argue that a small system, like that depicted in Figure 5.2, was a *safe abstraction* of the larger one with respect to any refinement-closed security property ϕ, meaning that if ϕ held for the small system, it was guaranteed to hold for the larger one (see Definition 4.1.1). This argument was made by showing that each cloud of objects in the larger system was *aggregated* by a corresponding object in the small system, meaning that the small system was an *aggregation* of the larger one (see Definition 4.1.2). We showed that this approach was sound for refinement-closed properties because an aggregation was refined by any system it aggregated (see Theorem 4.1.3). Hence, any aggregation was a safe abstraction of any larger system it aggregated with regards to any refinement-closed property.

The final step of this process (see Section 4.1.3) involved generalising the analysis over all such larger systems (*i.e.* generalising over all such clouds

that might exist in any of them). This was done by generalising the analysis of the (small-sized) aggregation to all choices for the sets of facets of each of the aggregating objects it contained, by applying the theory of *data-independence* [Laz99].

We show how to adapt each of these steps to allow us to generalise the analysis of weakened refinement-closed noninterference properties, like Weakened RCFNDC, to arbitrary-sized systems.

5.5.1 Safe Abstraction and Aggregation

We begin by noting that the original definition of safe abstraction was with regards to refinement-closed properties; however, weakened refinement-closed noninterference properties are not refinement-closed. Hence, we need to adapt the notion of safe abstraction to capture when, given two systems $System$ and $System'$, $System'$ is a safe abstraction of $System$ with respect to a weakened refinement-closed noninterference property $W\phi$.

$System'$ is a safe abstraction of $System$ with respect to some $W\phi$ when $W\phi(System') \Rightarrow W\phi(System)$. Recall that $W\phi$ is (by construction) equivalent to the property obtained by substituting ϕ for $Prop$ in Definition 5.2.2, so $W\phi(System) \Leftrightarrow \forall\, System_D \in DCRef(System) \bullet \phi(System_D)$ and similarly for $System'$. We can guarantee, then, that $W\phi(System') \Rightarrow W\phi(System)$ if $DCRef(System) \subseteq DCRef(System')$.[9] Hence, we take this as the definition for safe abstraction with respect to any weakened refinement-closed noninterference property.

Definition 5.5.1 (Safe Abstraction wrt Weakened Refinement-Closed Noninterference Properties)**.** One alphabetised parallel composition Sys' is a *safe abstraction* of another Sys wrt weakened refinement-closed noninterference properties iff $DCRef(System) \subseteq DCRef(System')$.

We must now show that all aggregations (as defined by Definition 4.1.2) are safe abstractions with regards to weakened refinement-closed noninterference properties, *i.e.* that if $System'$ is an aggregation of $System$ then $DCRef(System) \subseteq DCRef(System')$. The following theorem does so.

Theorem 5.5.2. Let $(Object, behaviour, facets, Data)$ and $(Object', behaviour', facets', Data)$ be two object-capability systems with identical data captured by the CSP processes $System = \|\|_{o \in Object}(behaviour(o), \alpha(o))$ and $System' = \|\|_{o \in Object'}(behaviour'(o), \alpha'(o))$ respectively, such that $System'$ is an aggregation of $System$. Then we have that $DCRef(System) \subseteq DCRef(System')$.

Proof. Suppose the conditions of the theorem. Then from Definition 4.1.2, there exists a surjection $Abs : Object \to Object'$ such that for all $o' \in Object'$,

[9] Because of the definition of $DCRef$ (see Definition 5.2.1), this subset relation is interpreted modulo failures-divergences equivalence.

$facets'(o') = \bigcup\{facets(o) \mid o \in Abs^{-1}(o')\}$ and Equation 4.1 is satisfied. Both systems clearly have the same alphabet.

Let $System_D \equiv_{FD} \|_{o \in Object}(b_o, \alpha(o))$ be an arbitrary deterministic componentwise refinement of $System$ (so that $\forall o \in Object \bullet behaviour(o) \sqsubseteq b_o \wedge det(b_o)$). We must show that there exists a deterministic componentwise refinement, $System'_D = \|_{o' \in Object'}(b_{o'}, \alpha'(o'))$ (where $\forall o' \in Object' \bullet behaviour'(o') \sqsubseteq b_{o'} \wedge det(b_{o'})$), of $System'$ such that $System_D \equiv_{FD} System'_D$. We show this by showing how to construct each $b_{o'}$.

Observe that $System_D \equiv_{FD} \|_{o' \in Object'}(P_{o'}, \alpha'(o'))$ where, for each $o' \in Object'$, $P_{o'} = \|_{o \in Abs^{-1}(o')}(b_o, \alpha(o))$. Each $P_{o'}$ must be deterministic because it is the alphabetised parallel composition of deterministic processes. We use each $P_{o'}$ as a *template* from which to derive the corresponding $b_{o'}$ from $behaviour'(o')$.[10]

Consider some $o' \in Object'$. Then $\|_{o \in Abs^{-1}(o')}(behaviour(o), \alpha(o)) \sqsubseteq P_{o'}$, hence, by Equation 4.1, $\forall s \in traces(System) \bullet (s \upharpoonright \alpha'(o'), X) \in failures(P_{o'}) \Rightarrow (s \upharpoonright \alpha'(o'), X) \in failures(behaviour'(o'))$.

Let $F_{o'} = \{(t, X) \in failures(P_{o'}) \mid \exists s \in traces(System) \bullet s \upharpoonright \alpha'(o') = t\}$. Then $F_{o'} \subseteq failures(behaviour'(o'))$. Because $P_{o'}$ is deterministic, by Axiom **F3**, $(t, X) \in F_{o'} \wedge (t, Y) \in F_{o'} \Rightarrow (t, X \cup Y) \in F_{o'}$.

Using Lemma A.0.6, we can remove all traces $\{t \frown \langle e \rangle \in traces(behaviour'(o')) \mid \exists X \bullet e \in X \wedge (t, X) \in F_{o'}\}$ from $behaviour'(o')$, giving us a valid refinement $R_{o'}$. In doing so, we can never remove any failure $(t, Y) \in F_{o'}$ because each application of Lemma A.0.6 to a process P, to remove the traces associated with some failure $(s, X) \in F_{o'}$, for which $F_{o'} \subseteq failures(P)$ yields a process Q for which $F_{o'} \subseteq failures(Q)$. Suppose otherwise for a contradiction. Then it must be the case that in removing traces associated with some failure $(s, X) \in F_{o'}$, we must remove some failure $(t, Y) \in F_{o'}$. By Lemma A.0.6, this happens only if $t = s$ and $(t, X \cup Y) \notin failures(P)$. However, this is impossible since we know that $(t, X \cup Y) \in F_{o'}$ and $F_{o'} \subseteq failures(P)$.

So $F_{o'} \subseteq failures(R_{o'})$ and $\forall (t, X) \in F_{o'} \bullet \forall x \in X \bullet t \frown \langle x \rangle \notin traces(R_{o'})$. By Lemma A.0.4, we define $b_{o'}$ to be the deterministic trace-equivalent refinement of $R_{o'}$. It must be the case that, in applying this lemma, no failure $(t, X) \in F_{o'}$ has been removed. Hence, $F_{o'} \subseteq failures(b_{o'}) \wedge det(b_{o'})$.

It remains to be shown that $System_D \equiv_{FD} \|_{o' \in Object'}(b_{o'}, \alpha'(o'))$. Note that because both are deterministic, it is enough to show that $failures(System_D) \subseteq failures(\|_{o' \in Object}(b_{o'}, \alpha'(o')))$. We note that, for all $o' \in Object'$, the containment of $F_{o'}$ in $failures(b_{o'})$ means that $\forall s \in traces(System_D) \bullet (s \upharpoonright \alpha'(o'), X) \in failures(P_{o'}) \Rightarrow (s \upharpoonright \alpha'(o'), X) \in failures(b_{o'})$. The argument then proceeds as in Lemma 4.1.3. □

[10] We can't simply set each $b_{o'} = P_{o'}$ since $behaviour'(o') \sqsubseteq P_{o'}$ is not generally true.

5.5.2 Data-Independence

We have shown that we can use aggregation to build smaller safe abstractions of larger systems with respect to weakened refinement-closed noninterference properties. This means that, for example, any particular larger system captured by Figure 5.3 can be safely abstracted by the small system depicted in Figure 5.2 when (in the small system) we set *facets*(High) and *facets*(Low) to be the sets of facets of all high and low objects in the larger system respectively.

Recall that the second step of our generalisation technique involves applying the theory of data-independence to show that the small system is secure for all disjoint choices for the sets *facets*(High) and *facets*(Low), thus generalising the analysis to all larger systems captured by Figure 5.3. We say that *facets*(High) and *facets*(Low) are *data-independent types* of the small system. Recall that data-independence theory allows us to derive a threshold for each data-independent type such that if we can show that the small system is secure for all choices for each type no larger than the respective threshold, then we can conclude that the small system is secure for all (nonempty) disjoint choices for each type. These thresholds naturally depend on both the system being analysed and property being tested.

We now present a theorem from which we derive a corollary that allows us to automatically calculate suitable data-independence thresholds when testing Weakened RCFNDC for systems that satisfy **NoEqT** (which, recall, asserts that the system need perform no equality tests between members of a data-independent type). While this theorem applies directly only to Weakened RCFNDC, its proof should be able to be adapted straightforwardly to cover other weakened refinement-closed noninterference properties.

As in Section 4.1.3, the basic idea is to relate behaviours of a large system, in which a data-independent type T has size larger than the threshold, to a small system, in which the same data-independent type has size of the threshold, and show that the presence of insecure behaviours in the large system imply the presence of corresponding insecure behaviours in the small one. We relate the behaviours of the large system to corresponding behaviours of the small system by a surjection $\phi : T \to T'$ that maps members of the type T instantiated in the large system to members of that same type instantiated (as T', whose size is the threshold) in the small system.

We will use the following standard result. Let P_T be a process that is data-independent in some set T and satisfies **NoEqT** for T, meaning that P_T never needs to test two values of T for equality. Let ϕ be a surjection whose domain is T, where we write $\phi(T)$ for $\{\phi(t) \mid t \in T\}$ and $\phi^{-1}(X)$ for $\{y \mid y \in T \wedge \phi(y) \in X\}$. Then [Laz99, Theorem 4.2.2], lifting ϕ to events and traces, we have that

$$\{(\phi(s), X) \mid (s, \phi^{-1}(X)) \in \mathit{failures}(P_T)\} \subseteq \mathit{failures}(P_{\phi(T)}). \tag{5.6}$$

Theorem 5.5.3. Let $S_T = \|_{i \in \{1,\ldots,n\}}(P_{T,i}, A_{T,i})$ be an alphabetised parallel composition, whose components and alphabets are polymorphically parameterised by some set T, such that S_T and each $P_{T,i}$ are data-independent in T and satisfy **NoEqT** for T. Also let H_T and L_T be two sets polymorphically parameterised by T that partition the alphabet of S_T for all non-empty T. Let W denote the maximum number of distinct elements of T that appear in any single event from L_T. Then $W + 1$ is a sufficient data-independence threshold for T for $\mathit{WRCFNDC}(S_T)$.

Proof. Assume the conditions of the theorem. Suppose for some T with size greater than W, S_T fails Weakened RCFNDC for H_T and L_T. Then let $\tilde{T} = \{\tilde{t}_0, \ldots, \tilde{t}_W\}$ for fresh elements $\tilde{t}_0, \ldots, \tilde{t}_W$. We show that $S_{\tilde{T}}$ fails Weakened RCFNDC for $H_{\tilde{T}}$ and $L_{\tilde{T}}$.

Let $\phi : T \to \tilde{T}$ be a surjection; we fix the choice of ϕ below. Lift ϕ to events by applying ϕ to all components of type T. Then ϕ maps an event in the alphabet of S_T to an event in the alphabet of $S_{\tilde{T}}$. Also, lifting ϕ to sets of events, $\forall\, i \in \{1, \ldots, n\} \bullet \phi(A_{T,i}) = A_{\tilde{T},i}$, $\phi(H_T) = H_{\tilde{T}}$ and $\phi(L_T) = L_{\tilde{T}}$.

Observe that $S_{\phi(T)} = S_{\tilde{T}}$. So, by Equation 5.6, the presence of certain behaviours in S_T implies the presence of related behaviours in $S_{\tilde{T}}$. Recall the definition of Weakened RCFNDC (Definition 5.3.20). Suppose S_T fails the first disjunct of Definition 5.3.20 for H_T and L_T. We show that $S_{\tilde{T}}$ fails this disjunct for $H_{\tilde{T}}$ and $L_{\tilde{T}}$. The second disjunct is handled similarly. Then there exists some s, l and $i \in \{1, \ldots, n\}$ such that

$$s \upharpoonright H_T \neq \langle\rangle \wedge l \in L_T \wedge s \mathbin{\hat{}} \langle l \rangle \in \mathit{traces}(S_T) \wedge s \setminus H_T \in \mathit{traces}(S_T) \wedge$$
$$l \in A_{T,i} \wedge s \upharpoonright A_{T,i} \neq s \setminus H_T \upharpoonright A_{T,i} \wedge (s \setminus H_T \upharpoonright A_{T,i}, \{l\}) \in \mathit{failures}(P_{T,i}).$$

Let $t_0, \ldots, t_{k-1}$ be the distinct members of T that appear in l. Then $k \leq W$. Choose $\phi(t_i) = \tilde{t}_i$ for $0 \leq i \leq k-1$ and let $\phi(t) = \tilde{t}_k$ for all other $t \in T - \{t_0, \ldots, t_{k-1}\}$. Let $\tilde{s} = \phi(s)$ and $\tilde{l} = \phi(l)$. Then $\tilde{s} \upharpoonright \tilde{H} \neq \langle\rangle \wedge \tilde{l} \in \tilde{L} \wedge \tilde{l} \in A_{\tilde{T},i} \wedge \tilde{s} \upharpoonright A_{\tilde{T},i} \neq \tilde{s} \setminus \tilde{H} \upharpoonright A_{\tilde{T},i}$. Applying Equation 5.6 to S_T, we have $\tilde{s} \mathbin{\hat{}} \langle \tilde{l} \rangle \in \mathit{traces}(S_{\tilde{T}}) \wedge \tilde{s} \setminus \tilde{H} \in \mathit{traces}(S_{\tilde{T}})$. Further, $\{l\} = \phi^{-1}(\{\tilde{l}\})$ by construction. So, applying Equation 5.6 to $P_{T,i}$, we obtain $(\tilde{s} \setminus \tilde{H} \upharpoonright A_{\tilde{T},i}, \{\tilde{l}\}) \in \mathit{failures}(P_{\tilde{T},i})$ as required. □

Observe that, in this proof, l is necessarily an event in the alphabet of a process that can perform both H_T and L_T events. Hence, we can strengthen this result to take W to be the maximum number of distinct values of type T in all such events in L_T.

Corollary 5.5.4. Let $S_T = \|_{i \in \{1,\ldots,n\}}(P_{T,i}, A_{T,i})$ be an alphabetised parallel composition, whose components and alphabets are polymorphically parameterised by some set T, such that S_T and each $P_{T,i}$ are data-independent in T and satisfy **NoEqT** for T. Also let H_T and L_T be two sets polymorphically parameterised by T that partition the alphabet of S_T for all non-empty T.

Let W denote the maximum number of distinct elements of T that appear in any single event from

$$L_T \cap \bigcup \{A_{T,i} \mid i \in \{1, \ldots, n\} \wedge A_{T,i} \cap H_T \neq \{\} \wedge A_{T,i} \cap L_T \neq \{\}\}.$$

Then $W + 1$ is a sufficient data-independence threshold for T for $WRCFNDC(S_T)$.

5.5.3 Generalising the Data-Diode Analysis

We now apply these ideas to generalise the analysis of the Data-Diode pattern to all systems captured by Figure 5.3 with arbitrary *HighData* and *LowData*.

Consider an arbitrary system captured by Figure 5.3. Let *HighObjects* and *LowObjects* denote the (obviously disjoint) sets of objects in the high and low object clouds respectively. Then let $T = \bigcup_{o \in HighObjects} facets(o)$ and $U = \bigcup_{o \in LowObjects} facets(o)$ denote the sets that contain all facets of the high and low objects respectively. *HighData* and *LowData* are the sets of high and low data in this system, each of which is a subset of the set *Data* of all data in this system. Let $V = Data$.

Then this system can be safely abstracted by a system $System_{T,U,V}$ of the form of Figure 5.2 where we set $facets(\mathsf{High}) = T$, $facets(\mathsf{Low}) = U$ and (within this abstraction) $LowData = HighData = Data = V$, *i.e.* initially we give both High and Low access to all data. We give High and Low access to all data because we're focusing here only on the covert information flow properties of the Data-Diode pattern; its overt information flow properties (which can be expressed as safety properties) can be generalised using the techniques of Chapter 4. Then $System_{T,U,V}$ and all of its components are data-independent in T, U and V and satisfy **NoEqT** for each.

We then define the sets $H_{T,U,V}$ and $L_{T,U,V}$ of high and low events in $System_{T,U,V}$ as follows.

$$H_{T,U,V} = \{|t.\mathsf{DDReader}, \mathsf{DDReader}.t, t.t' \mid t, t' \in T|\},$$
$$L_{T,U,V} =$$
$$\{|u.\mathsf{DDWriter}.\mathsf{Call}.d, \mathsf{DDWriter}.u.\mathsf{Return}.\mathsf{null}, u.u' \mid u, u' \in U, d \in V \cup \{\mathsf{null}\}|\}.$$

To apply Corollary 5.5.4, we first need to show that $H_{T,U,V}$ and $L_{T,U,V}$ partition the effective alphabet of $System_{T,U,V}$, *i.e.* that

$$STOP \sqsubseteq_T System_{T,U,V} \setminus (H_{T,U,V} \cup L_{T,U,V})$$

for all non-empty, disjoint T, U and V. Theorem 2.3.5 implies that 1 is a sufficient threshold for each set to prove this. Instantiating each set as a singleton set containing a fresh element and performing the test in FDR reveals that it holds as required.

To verify Weakened RCFNDC, Corollary 5.5.4 suggests thresholds for T, U and V of 1, 2 and 2 respectively. This implies we need to carry out the test for Weakened RCFNDC in FDR 4 times in order to show that it holds for all non-empty disjoint T, U and V. The most expensive of the 4 tests implied by these thresholds examines about 6.3 million state-pairs, taking less than 4 minutes to compile and complete on a desktop PC; the others are far cheaper. All tests pass, thus generalising our analysis of the covert information flow properties of the improved Data-Diode implementation.

5.6 Related Work

Information Flow in the Take-Grant Model In [Bis96], Bishop considers information flow in the Take-Grant protection model, which models object-capability systems. This considers only how information can flow in object-capability systems under the assumption that all objects are maximally hostile, since the Take-Grant model doesn't take into account the behaviour of trusted security-enforcing objects. For this reason, it cannot be used to reason about the information flow properties of object-capability patterns, which are (by definition) implemented by trusted security-enforcing objects.

Overt Flow and Object-Capability Patterns As we've said earlier, Spiessens' work on the Scoll formalism [SV05, Spi07] was the first to examine the security properties of object-capability patterns by taking into account the behaviour of trusted objects. As part of this work, Spiessens looked at overt information flow, including [Spi07, Section 8.4] a scenario that is very similar to the system depicted in Figure 5.1 that we've analysed here. In this chapter, we've mainly focused on the covert information flow properties of object-capability patterns, since overt information flow can be readily captured via safety properties, which were examined in detail in Chapter 3.

It should be noted that Scoll could also be used to reason about covert information flow. However, doing so would require the person who is constructing the model of a system to include in that model all of the means by which information can propagate covertly within the system. Our approach has the advantage that no such *a priori* knowledge of the mechanisms for covert information propagation is necessary, because these mechanisms are implicitly encoded in the information flow property being applied [ML09a].

Architectural Refinement and Information Flow We briefly discussed van der Meyden's work on information flow and architectural refinement [vdM09] in Section 4.4 and noted that van der Meyden's notion of architectural refinement shares some similarities with the idea of aggregation (which we've adapted from Spiessens [Spi07]). In particular, saying that $\mathcal{A}_1$

is an architectural refinement of $\mathcal{A}_2$ is similar to saying that $\mathcal{A}_2$ is an aggregation of $\mathcal{A}_1$. Hence, one might say that architectural refinement is similar to the idea that is the reverse of aggregation, which we'll call *disaggregation* for lack of a better term.

van der Meyden considers the problem of whether architectural refinement preserves information flow policies. That is, if we have architectures $\mathcal{A}_1$ and $\mathcal{A}_2$, where $\mathcal{A}_1$ is a refinement of $\mathcal{A}_2$, and we build a system in accordance with the refined architecture $\mathcal{A}_1$, does that system respect the information flow properties of the original architecture $\mathcal{A}_2$? van der Meyden shows that the answer is "yes" for a range of intransitive noninterference properties. In this sense, van der Meyden has shown that architectural refinement preserves a range of intransitive information flow properties.

In Section 5.5.1, we showed that if *System* is an aggregation of *System*$'$, then any definition for information flow security for object-capability patterns that conforms to our general characterisation (Definition 5.2.2) holds for *System*$'$ if it holds for *System*. In this sense, we showed that information flow security for object-capability patterns is preserved by disaggregation.

This therefore resonates with van der Meyden's proof that intransitive noninterference is preserved by architectural refinement.

5.7 Conclusion

In this chapter, we have seen that CSP can be applied to reason about the information flow properties of object-capability patterns. We saw that, in order to do so, one needs to make the necessary assumption that the only objects affected by a message exchange are those who partake in it. We found that this assumption could be encoded into a general definition (Definition 5.2.2) for information flow security for object-capability patterns. This definition is parameterised by an information flow property *Prop* and encodes the assumption above by deeming a system to be secure just when *Prop* holds for all of the system's deterministic componentwise refinements.

We saw that when choosing a transitive refinement-closed noninterference property to substitute for *Prop* to instantiate this general definition, that there are really only two effectively different choices that one can make: namely RCFNDC (equivalently Lazy Independence) or RCNIC. We found that one can mechanically derive a testable characterisation $W\phi$ of this definition when it is instantiated with such a refinement-closed noninterference property ϕ. This is done by weakening ϕ to ignore all pairs of inconsistent behaviour. We also saw that a refinement check for $W\phi$ could be derived from the refinement check for ϕ, therefore allowing the general definition to be instantiated and then automatically tested by refinement checking.

We demonstrated this by defining the property Weakened RCFNDC for Compositions, which instantiates the general definition with RCFNDC, and deriving a refinement check for Weakened RCFNDC from that for RCFNDC.

The same could also be done with RCNIC to a produce a weaker property that detects only when high events cause low events to occur, whose definition and associated refinement check are a simplification of those for Weakened RCFNDC. We found that the automatic test for Weakened RCFNDC was particularly useful for detecting and helping to correct covert channels in an implementation of the Data-Diode pattern. We expect that these tests should be widely applicable to other object-capability patterns too.

Finally, we saw that these kinds of analyses could be generalised to arbitrary-sized systems by adapting the techniques presented in Chapter 4. We proved a result that allows one to easily derive data-independence thresholds for these analyses for systems that satisfy **NoEqT**. Since the majority of object-capability patterns don't use EQ, we expect CSP models of them to satisfy **NoEqT**, allowing their information flow analyses to be easily generalised in this way. We leave the generalisation of these kinds of analyses for systems that don't satisfy **NoEqT** as future work.

While this chapter has focused on transitive noninterference properties, we believe that the same approach could also be taken to apply intransitive noninterference properties to object-capability patterns. This is because, as argued in Section 5.3.2, we expect most intransitive noninterference properties can be expressed equivalently for deterministic processes as some refinement-closed noninterference property,

As discussed later in Section 8.1 in Chapter 8, the work in this chapter is not directly applicable, without extension, to systems in which objects have access to shared clocks, by which they can exploit possible *timing channels*. Addressing this limitation to allow such channels to be detected is an obvious avenue for future work.

The techniques demonstrated in this chapter are vital, amongst other reasons, for ensuring that patterns designed to enforce certain confidentiality policies do not inadvertently violate those same policies by containing unknown covert channels. They can also be applied to ensure that a pattern properly prevents one set of objects, or one set of actions, from inappropriately interfering with other parts of a system.

All of the properties examined so far in this thesis have been readily expressible as refinement checks, enabling them to be automatically tested by FDR. In this sense, we've been able to stay within the bounds of what can be easily tested using FDR. We leave this realm in the following chapter, where we examine liveness properties under necessary fairness assumptions. We will find that these properties cannot be precisely expressed as refinement checks for FDR, which will force us to test sufficient conditions for them instead.

6 Liveness

In this chapter, we consider how to reason about certain kinds of liveness properties of object-capability patterns. A *liveness* property is one that asserts that something (good) *must* happen [Lam77], as opposed to a safety property that, recall, asserts that something (bad) must not happen. An example of such a property of an object-capability pattern might be that, once invoked, a trusted object must (eventually) Return to its caller, in order to avoid blocking the caller forever. Liveness tests of CSP processes are usually performed in the stable-failures or failures-divergences models, unlike safety properties which are usually tested via traces refinement checks.

Broadly speaking, the notion of liveness has two different interpretations in the context of CSP. Often, liveness is interpreted in the context of CSP to mean any property that asserts the absence of certain stable failures (and sometimes also divergences), *i.e.* any property that must be tested using a stable-failures or failures-divergences refinement check. Under this interpretation, a property like deadlock freedom is normally considered a liveness property. We depart from this convention in this thesis, instead adopting Alpern and Schneider's [AS85] characterisation of liveness, under which deadlock freedom is actually considered a safety property (albeit one that, unlike most other safety properties, must be tested in the stable-failures model rather than the traces model). We do so because Alpern and Schneider's notion of liveness turns out to be most appropriate for our purposes.

In Section 6.1, we introduce Alpern and Schneider's concept of liveness and show how it applies to CSP. We see, under this notion of liveness, that liveness properties are fundamentally different to safety properties because they deal with *infinite* behaviours, *i.e.* those of infinite length that take infinitely long to observe. This makes them more tricky to test for CSP processes than safety properties, most notably because it requires us to apply some kind of *fairness assumption* to the process being checked. We show how to encode traditional notions of fairness in CSP, and how to express liveness properties under these fairness assumptions, using a fragment of LTL and its associated *refusal-traces* semantics borrowed from Lowe [Low08]. We then prove that, in general, it is impossible to express tests for liveness properties under these fairness assumptions via CSP refinement checks for FDR to carry out. Instead, we derive some sufficient conditions, which can be applied in certain circumstances, for certain liveness properties under the *strong*

event fairness assumption. We show that these sufficient conditions can be framed as stable-failures refinement checks, and so be checked automatically by FDR.

In Section 6.2, we apply the results from Section 6.1 to analyse the liveness of the safe Trademarks implementation from Section 3.1, arguing that the results obtained can be easily adapted to similar patterns, including the Sealer-Unsealer implementation of Section 3.2. We consider an intuitive liveness property, expressed in LTL, that we would like the Trademarks pattern to uphold. Using the results from Section 6.1, we use FDR to analyse this liveness property of this pattern. We argue that untrusted capabilities cannot be easily handled whilst ensuring liveness when using blocking invocation only. We model the use of non-blocking invocation as exists in some object-capability operating systems like seL4 and Coyotos, and show how it can be used to ensure a live Trademarks implementation.

In Section 6.3, we then discuss how liveness analyses of object-capability patterns can be generalised to systems of arbitrary size, using the techniques developed earlier in Chapter 4. We illustrate this by showing how to generalise the liveness analysis of the Trademarks pattern from Section 6.2 in this way.

6.1 Liveness in CSP

6.1.1 Testing Liveness Directly in CSP

Under Alpern and Schneider's definitions [AS85], liveness properties differ fundamentally from safety properties because they can be violated by system behaviours of infinite length while not being violated by any behaviour of finite length. A safety property, on the other hand, is violated by an infinite length behaviour b only if it is also violated by some finite behaviour which is a prefix of b [AAH+85, cited in [Kin94]]. This means that, while all safety violations can be detected by observing only finite behaviours, properly detecting liveness violations requires one to observe behaviours of infinite length that take infinitely long to occur[1].

Consider, for instance, the liveness property that asserts that some event e must occur, commonly written "$\Diamond\, e$" when applying (see *e.g.* [PV01, Puh03, Puh05, SLDW08, Low08]) Linear Temporal Logic (LTL) [Pnu77] to CSP. Clearly, $\Diamond\, e$ is violated by any finite behaviour in which e does not occur that ends in deadlock. However, $\Diamond\, e$ is also violated by any behaviour of *infinite* length in which e does not occur. For some processes, such as $S = a \rightarrow S$, the only behaviours they contain that violate $\Diamond\, e$ are traces of infinite length, in this case the infinite trace $\langle a, a, a, \ldots \rangle$.

In general, liveness properties can be expressed and reasoned about only

[1] Notice that under this characterisation of safety and liveness, deadlock-freedom is indeed a safety property because all deadlocks take only a finite amount of time to observe.

in semantic models that accurately capture behaviours of infinite length. Of CSP's three standard denotational semantic models (namely the traces, stable-failures and failures-divergences models) introduced in Section 2.1, only the failures-divergences model records infinite behaviours. The only kind of infinite behaviour recorded in the failures-divergences model is divergence, which occurs when a process performs an infinite amount of internal activity without performing a visible event. Applying the CSP hiding operator to a process P to obtain a process $P \setminus X$, in which all events in the set X are hidden, effectively turns all events $x \in X$ into internal activity. Hence, one way to detect liveness violations in the form of infinite traces is to use hiding to turn these infinite traces into infinite unbroken sequences of internal actions, *i.e.* into divergences, which can then be detected using a refinement test in the failures-divergences model.

As noted previously by Lowe [Low08], to test that $\Diamond\, e$ holds for some divergence-free process P, one simply tests whether

$$e \rightarrow \mathbf{div} \sqsubseteq_{FD} P \setminus (\Sigma - \{e\}). \tag{6.1}$$

This test fails iff P can either: deadlock before performing e, since in this case $P \setminus (\Sigma - \{e\})$ will be able to refuse $\{e\}$ initially which the specification cannot; or perform an infinite sequence of non-e events, since in that case $P \setminus (\Sigma - \{e\})$ will be able to diverge initially which the specification cannot.

6.1.2 Fairness

The applicability of this direct approach to testing liveness properties in CSP tends to be limited in practice. This is because to reason about liveness properties, one usually needs to make some kind of *fairness assumption* [AFK88, LPS81, VVK05]. No such assumption is taken into account by a direct test of a liveness property, such as that from Equation 6.1.

A fairness assumption implicitly restricts the infinite traces that a process can perform by forbidding all of those that would be judged to be *unfair*, *i.e.* all those that violate the fairness assumption. Fairness usually means that "if a choice is possible sufficiently often, then it is sufficiently often taken" [AFK88]. The inclusion of unfair infinite behaviours in a process's semantics can often result in false violations of liveness properties being detected that could not arise in any fair implementation.

Take, for example, the parallel composition $T = S \;|||\; e \rightarrow STOP$, where recall $S = a \rightarrow S$, and consider whether T satisfies $\Diamond\, e$. Under no fairness assumptions, T clearly fails $\Diamond\, e$ because it can perform an infinite sequence of as. Indeed, applying the test from Equation 6.1 would reveal that it does not hold because $T \setminus (\Sigma - \{e\})$ can initially diverge. However, this raises the question as to whether this infinite behaviour of T is fair. Many intuitive notions of fairness would say "no".

A number of different fairness notions (see *e.g.* [LPS81, AFK88, VVK05] for just a fraction) have been proposed that might be applicable to T above.

On the one hand, we might say that any implementation of T that comprises two concurrently executing processes – one of which performs as, the other of which performs the event e – is unfair when one process is scheduled sufficiently often while the second is ready to be scheduled, without the second process ever being scheduled. Different interpretations of "sufficiently often" naturally give rise to stronger and weaker specific notions of fairness. In CSP, a process is ready to be scheduled when it has some event that is *stably available*, meaning that the process is in a stable state in which this event cannot be refused[2]. With this in mind, we frame fairness in general terms without reference to the particular structure of the system in question nor to its alphabet, as follows.

Definition 6.1.1 (A General Characterisation of Fairness)**.** *Unfairness* exists in some behaviour of infinite length when, from some point onwards in that behaviour, some event was stably available sufficiently often but never occurred. A *fair* infinite behaviour is one that contains no unfairness.

As we explain later in Section 6.1.3, it turns out that no standard denotational model for CSP can distinguish between when an event is guaranteed to be stably available (*i.e.* when the process in question is guaranteed to transition to a stable state in which the event is available) and when the event is merely guaranteed to be stably available if the process transitions to a stable state. Hence, in Definition 6.1.1, by "stably available" we really mean "stably available on the assumption that the process in question stabilised at this point". This turns out to have implications, which we discuss later in Section 6.1.5, when judging the fairness of infinite behaviours in which an event may have been stably available sufficiently often but this information was not recorded in the behaviour.

Because of its generality and natural expression within the concepts (namely the occurrence and stable refusal/availability of events within linear execution traces [Ros08]) embodied in the standard denotational semantic models of CSP, we choose to adopt Definition 6.1.1 as a useful general characterisation of fairness for infinite behaviours. However, alternatives exist (see *e.g.* [AFK88, Puh05]) that might be preferable in certain circumstances.

Weak event fairness and *strong event fairness* [Lam00, PV01, SLDW08] are examples of fairness properties that fit naturally within Definition 6.1.1. These concepts specialise the notions of *weak* and *strong fairness* [Lam77] respectively, applying them to the occurrence of events. Weak event fairness asserts that an event that is continually available occurs infinitely often. Strong event fairness asserts that an event that is available infinitely often occurs infinitely often; it naturally implies weak event fairness.

In practice, these kinds of fairness assumptions are upheld in real systems that implement a corresponding fair scheduling policy. However, the corre-

[2]The requirement for stability here is needed, for example, to ensure that the event that is available cannot subsequently become unavailable before the process is scheduled.

spondence between the fairness of a system's scheduling algorithm to one of these fairness assumptions can be shown only by modelling the scheduling algorithm and showing that for all objects that run in that system, that the fairness assumption is upheld. No such proof of any real object-capability system has ever been done, as far as we are aware. Therefore, these kinds of fairness assumptions are used in this chapter not because we are sure that any current object-capability system guarantees them, but simply because in CSP we cannot sensibly talk about the kinds of liveness properties in which we are interested without them.

6.1.3 Liveness under Fairness in LTL

Liveness and fairness properties are usually expressed as formulae in LTL. Lowe [Low08] presents an encoding of a fragment of LTL for CSP processes that is suitable for our purposes here. An LTL formula comprises *atomic formulae* that may be composed together with *temporal operators* and standard boolean operators. We consider a fragment of Lowe's dialect of LTL that uses the following atomic formulae.

- e: For an event $e \in \Sigma$, this formula means that the event e is (guaranteed to be) performed initially.
- $\mathsf{available}\ e$: For an event $e \in \Sigma$, this formula means that the event e cannot be stably refused initially; if stability is observed initially, then e must be stably available.

In our fragment of LTL, the following temporal operators are used to combine atomic formulae together, along with the standard boolean operators other than negation.

- $\Diamond$: For some property ϕ, $\Diamond\,\phi$ asserts that ϕ is eventually satisfied.
- $\Box$: For some property ϕ, $\Box\,\phi$ asserts that ϕ is always satisfied.

The fragment of LTL that we use expresses formulae ϕ that adhere to the following grammar, which has been chosen so as to be the simplest fragment capable of expressing the kinds of liveness and fairness properties in which we are interested.

$$\phi ::= e \mid \mathsf{available}\ e \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \Rightarrow \phi \mid \Diamond\,\phi \mid \Box\,\phi.$$

The properties of strong and weak event fairness are expressed in our fragment of LTL in the usual way (see *e.g.* [Lam00, PV01, SLDW08]).

Definition 6.1.2 (Strong and Weak Event Fairness)**.** The property of *strong event fairness*, denoted *SEF*, is defined in LTL as

$$SEF = \bigwedge_{e \in \Sigma} (\Box \Diamond \mathsf{available}\ e \Rightarrow \Box \Diamond e).$$

The property of *weak event fairness*, denoted WEF, is defined in LTL as

$$WEF = \bigwedge_{e \in \Sigma} (\Diamond \Box \, \mathsf{available}\ e \Rightarrow \Box \Diamond e).$$

Testing a liveness property ϕ, such as the property $\Diamond e$, under one of these fairness assumptions ψ, is equivalent to testing the property encoded by the LTL formula $\psi \Rightarrow \phi$ [Liu09]. So, for instance, the property that asserts that the event e must eventually occur under the assumption of strong event fairness is expressed by the LTL formula $SEF \Rightarrow \Diamond e$.

To see how liveness under these fairness assumptions works, consider the process P, where

$$P = a \rightarrow P \,\Box\, b \rightarrow P.$$

In P, both a and b are always stably available, hence under the assumption of either strong or weak event fairness, both events should occur infinitely often. Hence, writing $P \models \phi$ to mean that P satisfies the property encoded by the LTL formula ϕ, we have

$$P \models WEF \Rightarrow \Box \Diamond b, \qquad P \models SEF \Rightarrow \Box \Diamond b,$$

and similarly for a.

On the other hand, consider the process Q where

$$Q = a \rightarrow Q \sqcap b \rightarrow Q.$$

Suppose the internal choice in Q is always resolved to the right so that b is never stably available in Q. In this case, we see that under either strong or weak event fairness the event b need never occur. Hence, we have that

$$Q \not\models SEF \Rightarrow \Diamond b, \qquad Q \not\models WEF \Rightarrow \Diamond b,$$

and similarly for a. This makes sense since Q is refined by the process $R = a \rightarrow R$ in which b need never occur. It indicates that our notion of fairness does not forbid one branch of an internal choice being ignored forever, since this would invalidate our expectations about refinement.

To see the difference between strong and weak event fairness, consider the process S where

$$S = a \rightarrow c \rightarrow S \,\Box\, b \rightarrow c \rightarrow S.$$

The event b is not stably available continually in S; although it is stably available infinitely often. Hence

$$S \not\models WEF \Rightarrow \Box \Diamond b, \qquad S \models SEF \Rightarrow \Box \Diamond b,$$

and similarly for a.

Finally, we illustrate a (perhaps non-obvious) consequence that arises from giving our fragment of LTL, and hence our fairness properties, a *denotational* interpretation borrowed from Lowe [Low08], *i.e.* that arises from defining the semantics for our fragment of LTL in terms of one of CSP's standard denotational semantic models.

The formulae for strong and weak event fairness from Definition 6.1.2 are syntactically identical to the LTL encodings of strong and weak event fairness that have appeared previously (*e.g.* [PV01, Puh03, Puh05, SLDW08, Liu09]). However, their semantics differ subtly from those of most of their predecessors, which, unlike ours, have been given in terms of an *operational*, rather than denotational, semantics. Giving our fragment of LTL a denotational interpretation avoids problems that can arise with an operational interpretation of liveness and fairness properties, as explained later in Section 6.4. However, it does mean that our notions of strong and weak event fairness behave slightly differently than one might initially expect.

To see why, consider the process T where

$$T = a \rightarrow T \triangleright b \rightarrow T.$$

The operational semantics of the "$\triangleright$" operator imply that T can initially perform a; however, T can also initially perform some internal activity and transition to a stable state where it can perform only b. Because T can perform internal activity in its initial state, from which a is available, T's initial state is *unstable*.

The process T satisfies the property $\mathsf{available}\ b$ because initially b cannot be stably refused. Hence, T satisfies $\Box\, \mathsf{available}\ b$ and so we have that

$$T \models \mathit{WEF} \Rightarrow \Box \Diamond b, \qquad T \models \mathit{SEF} \Rightarrow \Box \Diamond b.$$

Note, of course, that these properties do not hold for a since T doesn't satisfy $\mathsf{available}\ a$.

T satisfying $\mathsf{available}\ b$ may seem surprising if one considers that b is not guaranteed to be initially stably available in T. For instance, in the case that T performs a initially, b never becomes stably available before a occurs. This highlights the fact that $\mathsf{available}\ b$ does *not* imply that b *must* be stably available initially; but only that b must be available whenever the process is observed to stabilise before performing a visible event.

This means that our notions of strong and weak event fairness are perhaps a little stronger than one might otherwise expect them to be. For the case of T above, for instance, both of our fairness properties judge as unfair the infinite behaviour in which T performs only as forever.

This situation arises unavoidably because we choose to give a denotational, rather than an operational, semantics for our fragment of LTL, and in particular the atomic formula $\mathsf{available}\ e$. All of CSP's standard denotational semantic models purposefully choose not to positively record the presence of instability [Ros09]. This means, for instance, that the process $e \rightarrow \mathit{STOP}$

cannot be distinguished from the process $e \rightarrow STOP \triangleright e \rightarrow STOP$ in any standard denotational model, Hence, any denotational interpretation of $\mathsf{available}\ e$ (over a standard denotational model) must necessarily say that the second process satisfies $\mathsf{available}\ e$ because the first one does so trivially.

Hence, no denotational interpretation of $\mathsf{available}\ e$ can assert that e *must* become stably available before any visible event is performed, but only that e must be available in those cases that the process is observed to stabilise before performing a visible event. This is precisely what $\mathsf{available}\ e$ means under our denotational interpretation, which we present shortly.

In practice, this slight oddity of our fairness properties is not problematic and the benefits obtained from adopting a denotational interpretation of our fragment of LTL far outweigh any negative consequences of doing so.

6.1.4 The Refusal-Traces Model

Lowe [Low08] presents a semantic encoding for LTL in the denotational semantic model for CSP known as the *refusal-traces model* [Muk93], which is also called the *refusal-testing model*. Indeed, this model turns out to be the least discriminating one that can accurately capture the semantics of many LTL formulae, including our fairness properties above, for divergence-free processes. This is because these kinds of property require a model that records information about the stable availability of individual events throughout (infinite) execution traces. The refusal-traces model is the least powerful model that does so accurately for divergence-free processes.

The refusal-traces model records finite behavioural traces that have the following forms:

1. $\langle X_1, a_1, X_2, a_2, \ldots, X_n, a_n \rangle$, which is a partial behaviour;
2. $\langle X_1, a_1, X_2, a_2, \ldots, X_n, a_n, \Sigma \rangle$, which is a completed behaviour that has ended in deadlock.

Above, each X_i is either: a set of events that can be stably refused after performing the sequence of events $\langle a_1, \ldots, a_{i-1} \rangle$, such that $a_i \notin X_i$; or the special symbol $\bullet$, used to indicate that no stable refusal was observed in between the events a_{i-1} and a_i occurring.

For example, the process $a \rightarrow STOP \triangleright STOP$ has the refusal traces $\langle \bullet, a, \Sigma \rangle$ and $\langle \Sigma \rangle$, indicating that a can be performed initially without stability being observed, before it deadlocks, or that it may deadlock initially. The process $a \rightarrow STOP \sqcap STOP$ has the same refusal traces except that it also has all of the refusal-traces in the set $\{\langle X, a, \Sigma \rangle \mid X \subseteq \Sigma - \{a\}\}$, indicating that it can initially stabilise and refuse everything except a before performing a. Because both processes can initially stabilise and refuse to perform a, neither satisfies the property "$\mathsf{available}\ a$" nor the (stronger) property "a".

Let $RefToken = \mathbf{P}\,\Sigma \cup \{\bullet\}$ be the set of *refusal tokens*. We treat $\bullet$ as if it were the smallest member under $\subseteq$ of $RefToken$, so that *e.g.* $\bullet \subseteq \{\}$, $a \notin \bullet$ for any $a \in \Sigma$, and $X \cup \bullet = X$.

Let PRT be the set of *partial refusal traces*, namely all those of form 1 above, and let DRT be the set of *deadlocked refusal traces*, namely all those of form 2 above. Then the set of all finite refusal-traces RT is the union of these two sets: $RT = PRT \cup DRT$. A process P is represented in the refusal-traces model by its set $\mathcal{R}[\![P]\!]$ of finite refusal-traces, where $\mathcal{R}[\![P]\!] \subseteq RT$. Like in the other denotational models, the representation $R = \mathcal{R}[\![P]\!]$ of any CSP process P satisfies certain axioms. Each axiom $\mathbf{R}i$ below essentially lifts the corresponding axiom $\mathbf{F}i$ of the stable-failures model (see Section 2.1) to the refusal-traces model. Here, s ranges over PRT, t over RT and A and B over $RefToken$.

R1. R contains the empty sequence $\langle\rangle$ and is prefix-closed.

R2. $s \mathbin{\hat{}} \langle A, a\rangle \mathbin{\hat{}} t \in R \wedge B \subseteq A \Rightarrow s \mathbin{\hat{}} \langle B, a\rangle \mathbin{\hat{}} t \in R$.

R3. $s \mathbin{\hat{}} \langle A, b\rangle \mathbin{\hat{}} t \in R \wedge A \neq \bullet \wedge s \mathbin{\hat{}} \langle A, a\rangle \notin R \Rightarrow s \mathbin{\hat{}} \langle A \cup \{a\}, b\rangle \mathbin{\hat{}} t \in R$.

Note that $\bullet$ is a possible value for B in **R2**.

For a divergence-free process P, we have that any partial behaviour can always be extended either by deadlock or by the performance of an event.

$$\forall\, s \in \mathcal{R}[\![P]\!] \cap PRT \bullet s \mathbin{\hat{}} \langle \Sigma\rangle \in \mathcal{R}[\![P]\!] \vee \exists\, a \bullet s \mathbin{\hat{}} \langle \bullet, a\rangle \in \mathcal{R}[\![P]\!]. \tag{6.2}$$

Refinement, denoted $\sqsubseteq_R$, in the refusal-traces model is defined as usual:

$$P \sqsubseteq_R Q \Leftrightarrow \mathcal{R}[\![Q]\!] \subseteq \mathcal{R}[\![P]\!].$$

Of course, to capture properties like $\Diamond\, e$, we need to be able to reason about infinite behaviours. Lowe [Low08] shows how the refusal-traces model, despite recording only finite behaviours, can capture infinite refusal-traces of the following form[3], where $a_i \notin X_i$ for all i:

$$\langle X_1, a_1, X_2, a_2, \ldots, X_n, a_n, \ldots\rangle.$$

A process P can exhibit an infinite refusal trace t of the above form iff all of t's finite prefixes are present in $\mathcal{R}[\![P]\!]$. Letting IRT denote the set of all infinite refusal traces, the set of P's infinite refusal traces $\mathcal{I}[\![P]\!]$ is

$$\mathcal{I}[\![P]\!] = \{t \in IRT \mid \forall\, s < t \bullet s \in \mathcal{R}[\![P]\!]\}.$$

[3]This result holds only for fragments of CSP, like that used in this thesis, that do not involve unbounded nondeterminism and that use a finite alphabet. Incidentally, the argument that proves this result is the same one that proves that the failures-divergences model can accurately predict divergence – a behaviour of infinite length – in $P \setminus X$ when P is divergence-free, whilst recording only finite traces and failures of P [Ros97].

6.1.5 LTL Semantics

The semantics of LTL formulae ϕ are usually defined over individual linear behaviours s, giving a satisfaction relation $s \models \phi$ that holds for a linear behaviour s if and only if the behaviour s satisfies ϕ. A property ϕ then holds for a process P, written $P \models \phi$, iff $s \models \phi$ for all behaviours s of P.

Lowe [Low08] takes a slightly different approach, instead, for each formula ϕ, defining the set $\mathcal{F}[\![\phi]\!]$ of refusal-traces (both finite and infinite) that satisfy ϕ, so that

$$P \models \phi \Leftrightarrow \mathcal{R}[\![P]\!] \cup \mathcal{I}[\![P]\!] \subseteq \mathcal{F}[\![\phi]\!].$$

Under Lowe's semantics, we may then say that, for an individual refusal-trace s, $s \models \phi \Leftrightarrow s \in \mathcal{F}[\![\phi]\!]$ and $P \models \phi \Leftrightarrow \forall\, s \in \mathcal{R}[\![P]\!] \cup \mathcal{I}[\![P]\!] \bullet s \models \phi$. From this, we translate Lowe's semantics for the temporal operators of our fragment of LTL, originally presented in terms of $\mathcal{F}[\![\phi]\!]$ in [Low08], to the more traditional $s \models \phi$ form.

Definition 6.1.3 (LTL Refusal-Traces Semantics)**.** Let n be a natural number, s be an arbitrary partial refusal-trace of length n, t be an arbitrary deadlocked refusal trace of length $n+1$ and u be an arbitrary infinite refusal trace so that

$$\begin{aligned} s &= \langle X_1, a_1, X_2, a_2, \ldots, X_n, a_n \rangle, \\ t &= \langle X_1, a_1, X_2, a_2, \ldots, X_n, a_n, \Sigma \rangle, \\ u &= \langle X_1, a_1, X_2, a_2, \ldots, X_n, a_n, \ldots \rangle, \end{aligned}$$

where in each case, for all relevant i, $a_i \notin X_i$. For a refusal-trace w, let w^i denote w with the first i refusal-and-event pairs removed. Then the semantics of the temporal operators in our fragment of LTL are defined as follows.

$$\begin{aligned} s &\models a \text{ iff } n = 0 \vee (n > 0 \wedge a_1 = a), \\ t &\models a \text{ iff } n > 0 \wedge a_1 = a, \\ u &\models a \text{ iff } a_1 = a, \\ s &\models \mathsf{available}\ a \text{ iff } n = 0 \vee (n > 0 \wedge a \notin X_1), \\ t &\models \mathsf{available}\ a \text{ iff } n > 0 \wedge a \notin X_1, \\ u &\models \mathsf{available}\ a \text{ iff } a \notin X_1, \\ s &\models \Diamond\, \phi \text{ iff } \mathsf{true}, \\ t &\models \Diamond\, \phi \text{ iff } \exists\, i \in \{0, \ldots, n\} \bullet t^i \models \phi, \\ u &\models \Diamond\, \phi \text{ iff } \exists\, i \in \mathbb{N} \bullet u^i \models \phi, \\ s &\models \Box\, \phi \text{ iff } \forall\, i \in \{0, \ldots, n\} \bullet s^i \models \phi, \\ t &\models \Box\, \phi \text{ iff } \forall\, i \in \{0, \ldots, n\} \bullet t^i \models \phi, \\ u &\models \Box\, \phi \text{ iff } \forall\, i \in \mathbf{N} \bullet u^i \models \phi. \end{aligned}$$

Let v be an arbitrary partial, deadlocked or infinite refusal-trace. Then the semantics of the boolean operators in our fragment of LTL are defined as follows, as one would expect.

$$v \models \phi \wedge \psi \text{ iff } s \models \phi \text{ and } s \models \psi,$$
$$v \models \phi \vee \psi \text{ iff } s \models \phi \text{ or } s \models \psi,$$
$$v \models \phi \Rightarrow \psi \text{ iff if } s \models \phi \text{ then } s \models \psi.$$

For a divergence-free CSP process P, we extend the satisfaction relation above to define when P satisfies an LTL formula ϕ as follows.

$$P \models \phi \Leftrightarrow \forall\, v \in \mathcal{R}[\![P]\!] \cup \mathcal{I}[\![P]\!] \bullet v \models \phi.$$

For any refusal-trace v, we write $v \not\models \phi$ when it is not the case that $v \models \phi$. We do likewise for the relation $P \models \phi$, so that $P \not\models \phi$ iff there exists some refusal-trace $v \in \mathcal{R}[\![P]\!] \cup \mathcal{I}[\![P]\!]$ such that $v \not\models \phi$.

Notice that all partial behaviours s satisfy conditions of the form $\Diamond\, \phi$. This is because any such s, by itself, cannot be said to violate such a condition, since, by Equation 6.2, s can always be extended and any such extension might satisfy the condition[4]. For the same reason, observe that the empty refusal-trace $\langle\rangle$, which is necessarily a partial refusal-trace, satisfies any property that can be expressed in our fragment of LTL.

The intuition that no deadlock refusal trace t can satisfy any formula ϕ of the form $\Box \Diamond \phi_1$ or $\Diamond \Box \phi_1$, for an atomic formula ϕ_1, can be trivially proved by considering $t^n = \langle \Sigma \rangle$ and whether $t^n \models \phi_1$, which it must in order for t to satisfy $\Box \Diamond \phi_1$ or $\Diamond \Box \phi_1$.

The reader can confirm that each of the examples considered in Section 6.1.3 and the judgements made there are consistent with these semantics. As a further example, consider the process

$$P = a \rightarrow d \rightarrow P \triangleright (b \rightarrow P \sqcap c \rightarrow P),$$

which initially can always stabilise and perform either b or c, or perform a from an unstable state. Having performed a, it stabilises and then performs d before returning to its initial state. It turns out that P contains no fair behaviours, under strong event fairness, that violate the liveness property $\Diamond\, b \vee \Diamond\, c$. Hence

$$P \models \mathit{SEF} \Rightarrow (\Diamond\, b \vee \Diamond\, c).$$

To see why, observe that the only behaviours of P that violate the property $\Diamond\, b \ \vee \ \Diamond\, c$ are the infinite refusal-traces from the set $B = \{\langle \bullet, a, X_1, d, \ldots, \bullet, a, X_n, d, \ldots \rangle \ \mid \ \forall\, i \bullet X_i \subseteq \Sigma - \{d\}\}$. However, none of these behaviours satisfy *SEF* since each actually satisfies the property $\Box \Diamond (\mathsf{available}\ b \wedge \mathsf{available}\ c)$, because of the presence of the infinite number of $\bullet$ refusals. This follows from the fact that our semantics, when applied

[4] These kinds of condition are said to be *machine closed* [AL91, VVK05] or, as originally introduced, *feasible* [AFK88].

just to an individual refusal-trace v, deems that $v \models \bigwedge_{e \in \Sigma} \mathsf{available}\ e$ whenever v begins with $\bullet$. While this may seem counter-intuitive, it doesn't cause any problems because we judge satisfaction of an LTL formula across all behaviours of a process.

Indeed our semantics has no choice but to operate in this way for $\bullet$. This is because $\bullet$ means not that instability was positively observed but that stability wasn't observed, even if the process in question did actually stabilise. As explained earlier in Section 6.1.3, no standard denotational model for CSP records positively the presence of instability. This implies that, under a denotational interpretation, $\mathsf{available}\ e$ can mean only that e must be available whenever the process is observed to stabilise before performing a visible event. Under this understanding of $\mathsf{available}\ e$, we see that every refusal-trace v that begins with $\bullet$ *must* satisfy $\bigwedge_{e \in \Sigma} \mathsf{available}\ e$ because v beginning with $\bullet$ means that no stability was observed in v before the first visible event was performed.

While the behaviours from B don't satisfy strong event fairness, some of them, such as the refusal-trace $\langle \bullet, a, \{b, c\}, d, \ldots \rangle$, do satisfy weak event fairness. Hence, we have that

$$P \not\models \mathit{WEF} \Rightarrow (\Diamond\, b \vee \Diamond\, c).$$

Also, while the behaviours in B satisfy $\Box \Diamond (\mathsf{available}\ b \wedge \mathsf{available}\ c)$, P has others that do not. An obvious example is the infinite behaviour $\langle \{a, c, d\}, b, \{a, c, d\}, b, \ldots, \{a, c, d\}, b, \ldots \rangle$. This behaviour satisfies both *SEF* and *WEF*. Hence,

$$P \not\models \mathit{SEF} \Rightarrow \Diamond\, b \wedge \Diamond\, c, \qquad P \not\models \mathit{WEF} \Rightarrow \Diamond\, b \wedge \Diamond\, c.$$

Finally, the definition of the satisfaction relation $P \models \phi$ means that every property ϕ expressible in our fragment of LTL is refinement-closed in the refusal-traces model. This includes, of course, liveness properties under fairness assumptions, like $\mathit{SEF} \Rightarrow \Diamond\, b$. This follows because if $P \models \phi$ then every behaviour of P must satisfy ϕ, and because any behaviour of a refinement Q of P is necessarily a behaviour of P.

6.1.6 Testing for Liveness under Fairness via Refinement

We now consider how we might test liveness properties under fairness assumptions, such as $\mathit{SEF} \Rightarrow \Diamond\, e$, expressible with the fragment of LTL defined above. As usual, we would like to be able to express these kinds of property using refinement checks that can be automatically carried out in FDR. We will prove that no refinement test $F(P) \sqsubseteq_{\mathcal{M}} G(P)$ exists, where $F(\_)$ and $G(\_)$ are CSP contexts and $\mathcal{M}$ is a standard denotational CSP model that FDR might support, that can express $P \models \mathit{SEF} \Rightarrow \Diamond\, e$ (and similarly for *WEF*) and similar properties that express liveness under fairness assumptions. This prevents such properties being directly tested using FDR at present.

In order to prove this, it is enough to show that the result holds when $\mathcal{M}$ is $\mathcal{FL}$ [Ros08, Ros09], the most discriminating denotational model that ignores divergence, and that it also holds when $\mathcal{M}$ is $\mathcal{FL}^{\Downarrow}$ and $\mathcal{FL}^{\#}$, the divergence-recording counterparts to $\mathcal{FL}$ that don't record infinite behaviours other than divergences. We begin by explaining each of these.

In [Ros09] Roscoe considers the different kinds of non-trivial[5] denotational models that can be defined for divergence-free CSP processes and shows that these models form a natural hierarchy in terms of their ability to distinguish processes. The topmost element of this hierarchy, *i.e.* the most discriminating model, is the model $\mathcal{FL}$, which Roscoe further refines in [Ros08]. The bottommost element is the traces model. This hierarchy includes the stable-failures model (which sits directly above the traces model) and the refusal-traces model (which sits in between the stable-failures model and $\mathcal{FL}$). We call all such models *finite linear observation models*.

$\mathcal{FL}$ records finite behavioural sequences of the following form:

$$\langle A_0, a_0, A_1, a_1, \ldots, A_{n-1}, a_{n-1}, A_n \rangle. \tag{6.3}$$

This sequence represents that the process in question can perform the trace of visible events $\langle a_0, a_1, \ldots, a_{n-1} \rangle$, with the events it can stably accept whilst doing so captured naturally by the A_i. Each A_i is a *generalised acceptance*, being either: a set of visible events that can be stably accepted at its point in the trace; or the special symbol $\bullet$, which, like in the refusal-traces model, is used to indicate that no stability was observed at this point in the trace even if the process being observed actually stabilised at this point.

We write $\mathcal{FL}[\![P]\!]$ to denote the set of finite behaviours of the form of Equation 6.3 of a divergence-free process P recorded by $\mathcal{FL}$.

Despite its apparent similarity to the refusal-traces model, one may observe that $\mathcal{FL}$ is strictly more powerful by considering the processes $P = (a \rightarrow STOP \,\Box\, b \rightarrow STOP) \sqcap STOP$ and $Q = a \rightarrow STOP \sqcap b \rightarrow STOP \sqcap STOP$. P and Q have identical semantics in the refusal traces model since both have all initial refusals that are a subset of Σ. However, they are distinguished in $\mathcal{FL}$ since P has the initial acceptances $\{a, b\}$ and $\{\}$, while Q has the initial acceptances $\{a\}$, $\{b\}$ and $\{\}$.

Each finite linear observation model $\mathcal{M}$, in this hierarchy, can be defined in terms of a function that can be applied to $\mathcal{FL}[\![P]\!]$ to give P's representation in $\mathcal{M}$ [Ros09]. Consider the stable-failures model. The pair ($traces(P)$,$failures(P)$), which is the representation of P in the stable-failures model, may be defined in terms of a function from $\mathcal{FL}[\![P]\!]$ as follows. Let the set $traces(P) = \{tr(s) \mid s \in \mathcal{FL}[\![P]\!]\}$, where the function tr gives the sequence of events performed in any finite refusal-trace, and let the set $failures(P) = \bigcup\{f(s) \mid s \in \mathcal{FL}[\![P]\!] \wedge last(s) \neq \bullet\}$, where $f(s) = \{(tr(s), X) \mid X \subseteq \Sigma - last(s)\}$.

[5] A denotational model is trivial when it identifies all processes.

Note that the failures-divergences model cannot be defined in this way, because $\mathcal{FL}$ does not record information about divergence. In fact, at least four separate hierarchies of CSP models exist [Ros08]; here, we concentrate on just three of them that together contain all of the standard CSP models that FDR might support. Two other hierarchies of models also exist that each precisely mimic the first hierarchy of finite linear observation models. These other two hierarchies contain respectively the the *divergence-strict* and *non-divergence-strict divergence-recording* counterpart of every finite linear observation model. The hierarchy of divergence-strict models contains the failures-divergences model.

The divergence-strict counterpart $\mathcal{M}^{\Downarrow}$ of a finite linear observation model $\mathcal{M}$, is obtained by augmenting $\mathcal{M}$ so that it also records the finite traces of a process on which it can diverge while completely ignoring what a process can actually do after it may have diverged. $\mathcal{M}^{\Downarrow}$ treats divergence as catastrophic, by effectively assuming that once a process can diverge, it can do anything. The failures-divergences model is the divergence-strict counterpart of the stable-failures model. The topmost element of the hierarchy of divergence-strict models is the model $\mathcal{FL}^{\Downarrow}$, the divergence-strict counterpart of $\mathcal{FL}$. As one might expect, each divergence-strict model $\mathcal{M}^{\Downarrow}$ can be defined in terms of a function from $\mathcal{FL}^{\Downarrow}$ to $\mathcal{M}^{\Downarrow}$ [Ros08].

The non-divergence-strict divergence-recording counterpart $\mathcal{M}^{\#}$ of a finite linear observation model $\mathcal{M}$ is similar to $\mathcal{M}$'s divergence-strict counterpart, $\mathcal{M}^{\Downarrow}$, in that it augments $\mathcal{M}$ by also recording the finite traces on which a process can diverge, but unlike $\mathcal{M}^{\Downarrow}$, $\mathcal{M}^{\#}$ dispenses with the divergence-strict assumption. That is, $\mathcal{M}^{\#}$ does not assume that once a process can diverge, it can do anything at all. The topmost element of the hierarchy of non-divergence-strict divergence-recording models is the model $\mathcal{FL}^{\#}$, the non-divergence-strict divergence-recording counterpart of $\mathcal{FL}$. Each non-divergence-strict divergence-recording model $\mathcal{M}^{\#}$ can naturally be defined in terms of a function from $\mathcal{FL}^{\#}$ to $\mathcal{M}^{\#}$. While FDR does not currently support refinement-checking in any non-divergence-strict divergence-recording model $\mathcal{M}^{\#}$, there is no reason why it couldn't be extended to do so.

The finite linear observation models $\mathcal{M}$, divergence-strict models $\mathcal{M}^{\Downarrow}$ and non-divergence-strict divergence-recording models $\mathcal{M}^{\#}$ are the only standard models of CSP that FDR might reasonably support. This is because the only other standard denotational models for CSP are ones that become more powerful than these three kinds of model, only when applied to processes that are not finitely nondeterministic [Ros08]. FDR cannot support processes that are not finitely nondeterministic. Hence, FDR can reasonably support refinement checking only in these three kinds of model.

We show that no refinement-test in any finite-linear observation model, divergence-strict model, or non-divergence-strict divergence-recording model can express general liveness properties under strong or weak event fairness. We do so by generalising a proof by Roscoe in [Ros05].

In [Ros05], Roscoe considers the expressive power of failures-divergences refinement, providing a characterisation of a number of classes of properties that can, and cannot, be expressed as refinement checks in this model. We generalise a small part of his work in order to prove our result.

Given a property, to express it in the form of a CSP refinement test in some model $\mathcal{M}$, we need to find two CSP contexts, $F(\_)$ and $G(\_)$, such that for any process P, P satisfies the property iff $F(P) \sqsubseteq_{\mathcal{M}} G(P)$. The following theorem, which is adapted from [Ros05, p. 106], shows how to demonstrate that a particular property cannot be expressed in this way for any finite-linear observation model, divergence-strict model or non-divergence-strict divergence-recording model $\mathcal{M}$.

Theorem 6.1.4. Let the set of all visible events Σ be finite. Let $\mathcal{M}$ be a finite linear observation model, a divergence-strict model, or a non-divergence-strict divergence-recording model, and let *Prop* be some property of CSP processes where we write $Prop(P)$ to mean that *Prop* is satisfied by process P. If there exists an infinite decreasing (under $\sqsubseteq_{\mathcal{M}}$) sequence $\langle B_k \mid k \in \mathbb{N} \rangle$ of trace-equivalent and divergence-free processes, and a single divergence-free process B^* that is the limit of this sequence, where $\forall\, k \in \mathbb{N} \bullet B^* \equiv_T B_k \wedge Prop(B_k)$ but $\neg Prop(B^*)$, then no refinement-test $F(P) \sqsubseteq_{\mathcal{M}} G(P)$ exists that can express $Prop(P)$ for arbitrary P.

Proof. Let $\mathcal{M}[\![P]\!]$ denote P's representation in $\mathcal{M}$. Then we have that $\mathcal{M}[\![B^*]\!] = \bigcup_{k\in\mathbb{N}} \mathcal{M}[\![B_k]\!]$. We use proof by contradiction. Suppose there is a refinement test of the form $F(P) \sqsubseteq_{\mathcal{M}} G(P)$ that expresses $Prop(P)$. We must have $\forall\, k \in \mathbb{N} \bullet F(B_k) \sqsubseteq_{\mathcal{M}} G(B_k)$ but $F(B^*) \not\sqsubseteq_{\mathcal{M}} G(B^*)$. Then $G(B^*)$ must have some behaviour $b \in \mathcal{M}[\![G(B^*)]\!] - \mathcal{M}[\![F(B^*)]\!]$ that $F(B^*)$ does not. Hence, $\forall\, k \in \mathbb{N} \bullet b \notin \mathcal{M}[\![F(B_k)]\!]$, and so

$$\forall\, k \in \mathbb{N} \bullet b \notin \mathcal{M}[\![G(B_k)]\!]. \tag{6.4}$$

Observe that b cannot be a divergence since B^* is trace-equivalent to every B_k and, for any process P, the divergences of $G(P)$ depend only on P's traces and divergences. Consider arbitrary CSP processes P and Q and an arbitrary CSP context $H(\_)$. Then, whenever b is a behaviour of $H(P)$ that is not a divergence, there exists a finite set Φ of behaviours of P such that $b \in \mathcal{M}[\![H(Q)]\!]$ whenever $\Phi \subseteq \mathcal{M}[\![Q]\!]$. It follows that there exists some finite set Φ of B^*'s behaviours that gives rise to the behaviour $b \in \mathcal{M}[\![G(B^*)]\!]$. Because Φ is finite, for some sufficiently large choice of k we must have that $\Phi \subseteq \mathcal{M}[\![B_k]\!]$ and, therefore, $b \in \mathcal{M}[\![G(B_k)]\!]$. This contradicts Equation 6.4. Hence, ϕ is not expressible as $F(P) \sqsubseteq_{\mathcal{M}} G(P)$. □

Corollary 6.1.5. No refinement test in any standard denotational model for CSP that FDR might support can express the properties $SEF \Rightarrow \Diamond\, e$, $WEF \Rightarrow \Diamond\, e$, $SEF \Rightarrow \Box \Diamond\, e$ and $WEF \Rightarrow \Box \Diamond\, e$.

Proof. Let

$$\begin{aligned} B^* &= a \to B^* \sqcap b \to B^*, \\ B_0 &= a \to B_0 \triangleright b \to B_0, \\ B_k &= a \to B_{k-1} \sqcap b \to B_{k-1}, \text{for } k > 0. \end{aligned}$$

Then, in all finite-linear models, divergence-strict models and non-divergence-strict divergence-recording models, B^* is indeed the limit of the decreasing sequence $\langle B_0, B_1, \ldots \rangle$, and B^* is trace-equivalent to each B_k. Observe that $B_0 \models \mathit{WEF} \Rightarrow \Box \Diamond b$ and so $\forall k \bullet B_k \models \mathit{WEF} \Rightarrow \Box \Diamond b$ and, hence, $\forall k \bullet B_k \models \mathit{WEF} \Rightarrow \Diamond b$ too. However, $B^* \not\models \mathit{SEF} \Rightarrow \Diamond b$ because B^* has the refusal-trace $\langle \{b\}, a, \{b\}, a, \ldots \rangle$ that satisfies *SEF* and, hence, $B^* \not\models \mathit{SEF} \Rightarrow \Box \Diamond b$ too. Note finally that if $P \models \mathit{WEF} \Rightarrow \phi$ then $P \models \mathit{SEF} \Rightarrow \phi$ for any process P and formula ϕ from our fragment of LTL, and similarly if $P \not\models \mathit{SEF} \Rightarrow \phi$ then $P \not\models \mathit{WEF} \Rightarrow \phi$. Thus, the result follows by Theorem 6.1.4. □

6.1.7 Sufficient Conditions for Liveness under Fairness

Because we cannot frame liveness under strong or weak event fairness in terms of CSP refinement tests for FDR, we instead consider some sufficient conditions for the properties $\mathit{SEF} \Rightarrow \Diamond e$ and $\mathit{SEF} \Rightarrow \Box \Diamond e$ that can be expressed as refinement tests in the stable-failures model. We show that, for certain kinds of systems, it is sufficient to show the absence of certain stable-failures in order to guarantee these properties.

Recall that the systems analysed in this thesis are all expressed as CSP processes $\mathit{System} = \|_{o \in \mathit{Object}} (\mathit{behaviour}(o), \alpha(o))$ that are the alphabetised parallel composition of processes, one for each object. We therefore restrict our attention to liveness properties applied to arbitrary alphabetised parallel compositions $\mathit{Sys} = \|_{i \in \{1,\ldots,n\}} (P_i, A_i)$. Suppose such a system contains a component process P_j that has some event $e \in A_j$ in its alphabet such that $P_j \setminus \Sigma - \{e\}$ is divergence-free. This means that P_j cannot perform an infinite sequence of non-e events, *i.e.* that each non-e event that P_j performs brings it closer either to deadlocking or to performing e. We would expect, then, that if $\mathit{Sys} \not\models \mathit{SEF} \Rightarrow \Diamond e$, that P_j must become permanently blocked in *Sys* at some point before e is performed. This is clearly true in the case that *Sys* has no infinite behaviours that satisfy *SEF* but fail $\Diamond e$. We prove that this result also holds when *Sys* does have such infinite behaviours.

Lemma 6.1.6. Let *Sys* be an alphabetised parallel composition, $\mathit{Sys} = \|_{i \in \{1,\ldots,n\}} (P_i, A_i)$, of divergence-free processes P_i and e be an event from Σ (which is finite). Let P_j be a component of *Sys* such that $e \in A_j$ and $P_j \setminus A_j - \{e\}$ is divergence-free. Let s be an infinite refusal-trace of *Sys*, so $s \in \mathcal{I}(\mathit{Sys})$. Then $s \models \mathit{SEF}$ and $s \not\models \Diamond e$ only if e never occurs in s and s has an infinite suffix $t = \langle X_1, a_1, X_2, a_2, \ldots \rangle$ such that $\forall i > 0 \bullet A_i \subseteq X_i$.

Proof. Assume the conditions of the lemma. We use proof by contradiction. Suppose s has no such infinite suffix t. Then, because A_j is necessarily finite, there exists some set $A \subseteq A_j$ such that $\forall\, a \in A \bullet s \models \Box \Diamond \mathsf{available}\ a$. Since $s \models \mathit{SEF}$ and $s \not\models \Diamond\, e$, it must be the case that infinitely many events from A occur in s without e occurring. Hence, P_j must be able to perform infinitely many A-events without performing e. However, this clearly contradicts $P \setminus (\Sigma - \{e\})$ being divergence-free. Hence s must have such a suffix t. □

Observe that Lemma 6.1.6 holds when $\Diamond\, e$ is replaced by $\Box \Diamond\, e$ and the consequence that s contains no occurrence of e is relaxed to t containing no occurrence of e.

Lemma 6.1.7. Let *Sys* be an alphabetised parallel composition, $\mathit{Sys} = \|_{i \in \{1,\ldots,n\}} (P_i, A_i)$, of divergence-free processes P_i and e be an event from Σ (which is finite). Let P_j be a component of *Sys* such that $e \in A_j$ and $P_j \setminus A_j - \{e\}$ is divergence-free. Let s be an infinite refusal-trace of *Sys*, so $s \in \mathcal{I}(\mathit{Sys})$. Then $s \models \mathit{SEF}$ and $s \not\models \Box \Diamond\, e$ only if s has an infinite suffix $t = \langle X_1, a_1, X_2, a_2, \ldots \rangle$ such that $\forall\, i > 0 \bullet A_i \subseteq X_i \wedge a_i \neq e$.

With these results, we can now prove sufficient conditions for $\mathit{SEF} \Rightarrow \Diamond\, e$ and $\mathit{SEF} \Rightarrow \Box \Diamond\, e$ respectively.

Corollary 6.1.8. Let *Sys* be an alphabetised parallel composition, $\mathit{Sys} = \|_{i \in \{1,\ldots,n\}} (P_i, A_i)$, of divergence-free processes P_i and e be an event from Σ (which is finite). Let P_j be a component of *Sys* such that $e \in A_j$ and $P_j \setminus A_j - \{e\}$ is divergence-free. If *Sys* has no stable-failure (s, A_j) where $s \upharpoonright \{e\} = \langle\rangle$, then $\mathit{Sys} \models \mathit{SEF} \Rightarrow \Diamond\, e$.

Proof. We prove the contrapositive. So suppose we have some alphabetised parallel composition *Sys* and event e as stated and that $\mathit{Sys} \not\models \mathit{SEF} \Rightarrow \Diamond\, e$. Then there exists some behaviour $t \in \mathcal{R}[\![\mathit{Sys}]\!] \cup \mathcal{I}[\mathit{Sys}]$ such that $t \models \mathit{SEF}$ but $t \not\models \Diamond\, e$. There are two cases to consider, namely t is finite or not.

If t is finite (*i.e.* $t \in \mathcal{R}[\![\mathit{Sys}]\!]$), then it must be a deadlock refusal-trace $\langle X_1, a_1, \ldots, X_n, a_n, \Sigma \rangle$ from *DRT*. Let $s = \langle a_1, \ldots, a_n \rangle$ be the trace of events performed in t. Then $s \upharpoonright \{e\} = \langle\rangle$ and $(s, \Sigma) \in \mathit{failures}(\mathit{Sys})$. Hence, because refusals are subset-closed by Axiom **F2**, $(s, A_j) \in \mathit{failures}(\mathit{Sys})$.

If t is not finite, then t is an infinite behaviour from $\mathcal{I}[\![\mathit{Sys}]\!]$ that satisfies *SEF* and fails $\Diamond\, e$. Then, by Lemma 6.1.6, t necessarily contains infinitely many refusals X_i where for each, $X_i \supseteq A_j$. Let s be the sequence of events in t that precedes the first such refusal X_i. Then $s \upharpoonright \{e\} = \langle\rangle$ and $(s, X_i) \in \mathit{failures}(\mathit{Sys})$. Because refusals are subset-closed by Axiom **F2**, $(s, A_j) \in \mathit{failures}(\mathit{Sys})$ as required. □

By a similar argument, applying Lemma 6.1.7 instead of Lemma 6.1.6, we see that a sufficient condition for $\mathit{SEF} \Rightarrow \Box \Diamond\, e$ is the absence of all stable-failures (s, A_j) for an arbitrary trace s.

Corollary 6.1.9. Let *Sys* be an alphabetised parallel composition, $Sys = \|_{i\in\{1,\ldots,n\}}(P_i, A_i)$, of divergence-free processes P_i and e be an event from Σ (which is finite). Let P_j be a component of *Sys* such that $e \in A_j$ and $P_j \setminus A_j - \{e\}$ is divergence-free. If *Sys* has no stable-failure (s, A_j), then $Sys \models SEF \Rightarrow \Box\Diamond e$.

So let *Sys*, P_j and e be as described in the statements of these results. Each of the sufficient conditions from Corollaries 6.1.8 and 6.1.9 can be readily expressed as stable-failures refinement tests involving *Sys*. To test the sufficient condition for $Sys \models SEF \Rightarrow \Diamond e$, we simply build the most general specification process *Spec* that can never refuse all events from A_j before e has been performed and test whether $Spec \sqsubseteq_F Sys$. *Spec* can be written as shown in Snippet 6.1.

$$
\begin{array}{l}
Spec = \\
\quad (?d : \Sigma - A_j \rightarrow Spec) \triangleright \\
\quad\quad (\$a : A_j \rightarrow \textbf{if } a = e \textbf{ then } CHAOS_\Sigma \textbf{ else } Spec).
\end{array}
$$

Snippet 6.1: A specification for testing a sufficient condition for $SEF \Rightarrow \Diamond e$.

Spec can perform all events from Σ. We use the $\triangleright$ operator to allow *Spec* to refuse all events from $\Sigma - A_j$ but force it to always offer at least one event from A_j. Hence, before performing its first e, *Spec* may refuse any set of events X where $X \cap A_j \subset A_j$, but may never refuse the entirety of A_j. We then have that

$$(CHAOS_\Sigma \sqsubseteq_{FD} (P_j \setminus \Sigma - \{e\}) \wedge Spec \sqsubseteq_F Sys) \Rightarrow Sys \models (SEF \Rightarrow \Diamond e).$$

The first of these refinement checks ensures that $P_j \setminus (\Sigma - \{e\})$ is divergence-free. Having checked this, the second refinement assertion tests the sufficient condition for $SEF \Rightarrow \Diamond e$.

Similarly, the sufficient condition for $\Box\Diamond e$ is tested by replacing "$CHAOS_\Sigma$" in Snippet 6.1 above with "*Spec*", to assert that the system in question must continue to never refuse the entirety of A_j after each occurrence of e.

In the following section, we apply these results to analyse the liveness of the safe Trademarks implementation from Section 3.1 under the assumption of strong event fairness.

6.2 Live Authenticating Trademarks

An obvious drawback of both our Trademarks and Sealer-Unsealer implementations from Chapter 3, is that a specimen, handed to a guard or unsealer respectively, can mount a denial-of-service attack against the guard

or unsealer by never Returning control to it after being Called. This then prevents the guard or unsealer from responding to its caller.

Generally, in an object-capability operating system, an object cannot rely on an untrusted object, to which it holds a capability, to be ready to receive an invocation or to reply to an invocation. In a single-threaded object-capability language, an object cannot rely on an untrusted object, to which it holds a capability, to return from a blocking invocation.

These observations have obvious consequences for implementers of authentication and coercion abstractions like our Trademarks and Sealer-Unsealer implementations. No object-capability language of which the author is aware provides any means to guard against an object that is invoked, using a blocking invocation, from refusing to return control to its invoker. Hence, in such systems, it is impossible to build an authentication or coercion mechanism that is invulnerable to denial-of-service, when that mechanism relies on having to perform blocking invocations on specimens to be authenticated or coerced respectively.

In object-capability languages that provide an *EQ* primitive, one may use the *capability set* approach described at the beginning of Section 3.1 to provide a live implementation of the Trademarks pattern, in which a guard maintains a collection of authentic capabilities, testing a specimen against each for equality in order to authenticate it. This approach can also be adapted to implement the Sealer-Unsealer pattern by using *EQ* to maintain an associative mapping from boxes to their contents (see [Yee99]). However, as previously noted, this solution is problematic for garbage collection and its time and storage complexity can scale linearly with the number of authentic capabilities. In general, we conclude that if authentication or coercion are to be available in an object-capability language for which liveness is of concern for application developers, the language should provide these services primitively.

In the case of object-capability operating systems, the problem of unresponsive objects that must be invoked can be avoided if the operating system includes primitives for inter-process-communication (IPC) that do not require the sender of a message to block indefinitely waiting for the receiver to be ready to receive it. Such facilities are available in systems that implement IPC operations in which a *timeout* can be specified that defines the maximum time that the thread performing the IPC operation is willing to block waiting for it to complete. Examples of object-capability operating systems that provide such a facility include seL4 and Coyotos (see Figure 2.1), which provide non-blocking send primitives that are guaranteed not to block a sender at all, as if the sender had specified a timeout of 0. We model this facility and show how it can be used in these systems to implement a live Trademarks implementation. The strategies employed here and the results obtained also apply to similar patterns, including the Sealer-Unsealer implementation from Section 3.2.

6.2.1 Deriving a Live Trademarks Implementation

A Draft Implementation

We modify the behaviour of a guard so that it uses a non-blocking send when Calling specimens and no longer waits for a Return message before continuing but instead waits a fixed time for the specimen to place a capability to itself inside its slot before continuing.

Adapting the model of a guard, $AGuard(me, slotR, slotW)$ with identity me and slot capabilities $slotR$ and $slotW$ respectively, to follow this strategy gives the new definition which appears in Snippet 6.2.

$$
\begin{aligned}
&AGuard(me, slotR, slotW) = \\
&\quad ?from!me!\mathsf{Call}?specimen : Capability \rightarrow \\
&\quad me!slotW!\mathsf{Call}!\mathsf{null} \rightarrow slotW!me!\mathsf{Return}!\mathsf{null} \rightarrow \\
&\quad \left(\begin{array}{l} me!specimen!\mathsf{Call}!\mathsf{null} \rightarrow AGuard'(me, slotR, slotW, from, specimen) \\ \triangleright\ AGuard'(me, slotR, slotW, from, specimen) \end{array}\right) \\
&AGuard'(me, slotR, slotW, from, specimen) = \\
&\quad me!slotR!\mathsf{Call}!\mathsf{null} \rightarrow slotR!me!\mathsf{Return}?val \rightarrow \\
&\quad \textbf{if } val = specimen \textbf{ then} \\
&\qquad me!from!\mathsf{Return}!me \rightarrow AGuard(me, slotR, slotW) \\
&\quad \textbf{else } me!from!\mathsf{Return}!\mathsf{null} \rightarrow AGuard(me, slotR, slotW)
\end{aligned}
$$

Snippet 6.2: The behaviour of a live guard using non-blocking sends.

When Calling a specimen, *specimen*, we use the CSP timeout operator to allow the guard to continue to its next state, $AGuard'$, if the sending of the Call message cannot complete promptly. Hence, the guard need not block when performing this send. Note that we model the guard so that its next state is identical whether the send completes or not, giving it no way of knowing if the send completed successfully. This models the semantics of the non-blocking send primitives in object-capability operating systems like seL4 and Coyotos [DEE08, SA07]. Once in this next state, the guard avoids waiting for a Return from the specimen and instead goes ahead and reads the slot's value. CSP has no in-built explicit model of time, but in a real implementation it is likely that the guard would wait a fixed amount of time before reading the slot's value.

The basic liveness property that we would like our Trademarks implementation to satisfy is that whenever a guard is Called with a specimen, it *must* return a response to its caller in a finite amount of time. If this property is upheld by a guard with identity *me*, then when that guard performs its first *from*.*me*.Call.*specimen* event when being invoked, it must eventually perform either *me*.*from*.Return.null or *me*.*from*.Return.*me* no matter what happens. Assuming the object, *from*, that has Called the guard is willing to

accept its Return message, their parallel composition should be able to perform one of the Return messages in a finite amount of time after the guard has been Called.

We define the behaviour of an object with facets *facets*, initial capabilities *caps* and data *data*, that is *live* to all Return messages of a guard *guard* – *i.e.* it never refuses to accept a Return message from *guard* – as the process $LiveToGuardsReturn(facets, caps, data, guard)$. This appears in Snippet 6.3.

$$
\begin{array}{l}
LiveToGuardsReturn(facets, caps, data, guard) = \\
\left(\begin{array}{l}
?me : facets?c : caps \cup facets?op?arg : caps \cup data \cup \{\mathsf{null}\} \rightarrow \\
\quad LiveToGuardsReturn(facets, caps, data, guard) \ \Box \\
?from : Capability - facets?me : facets?op?arg \rightarrow \\
\quad \textbf{let}\ C' = \{arg, from\} \cap Capability\ ;\ D' = \{arg\} \cap Data\ \textbf{within} \\
\quad LiveToGuardsReturn(facets, caps \cup C', data \cup D')
\end{array}\right) \\
\triangleright\ guard?me : facets!\mathsf{Return}?arg \rightarrow \\
\quad \textbf{let}\ C' = \{arg, guard\} \cap Capability\ ;\ D' = \{arg\} \cap Data\ \textbf{within} \\
\quad LiveToGuardsReturn(facets, caps \cup C', data \cup D')
\end{array}
$$

Snippet 6.3: An object that cannot refuse Guard's Return messages.

This process has all behaviours of $Untrusted_{OS}(facets, caps, data)$ except that it cannot ever refuse to perform any of the events from the set $\{\!|guard.me.\mathsf{Return} \mid me \in facets|\!\}$. Notice that we use the timeout operator, "$\triangleright$", to ensure that it can never refuse to perform any of the events in this set. *LiveToGuardsReturn* can refuse all other events at any time because its definition is equivalent to one where there is a "$\sqcap\ STOP$" clause on the left-hand side of the "$\triangleright$" operator. This follows because, for any processes P and Q, $(P \sqcap STOP) \triangleright Q \equiv_F P \triangleright Q$.

To analyse the liveness of our Trademarks implementation, we instantiate the system depicted in Figure 6.1, giving us the process *System*. It extends the system depicted earlier in Figure 3.1 with an extra object,

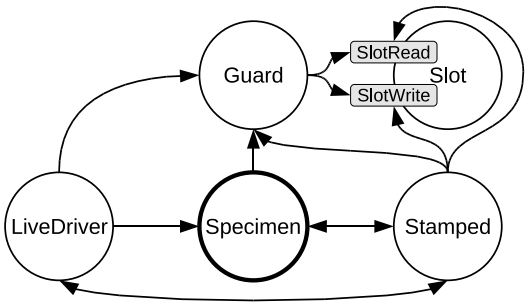

Figure 6.1: Testing the Trademarks implementation for liveness.

LiveDriver, whose behaviour is as follows.

$$\begin{array}{l} \mathit{behaviour}(\mathsf{LiveDriver}) = \\ \quad \mathit{LiveToGuardsReturn}(\{\mathsf{LiveDriver}\}, \mathit{Capability} - \mathit{facets}(\mathsf{Slot}), \mathit{Data}, \mathsf{Guard}) \end{array}$$

The behaviour of the other objects is as before.

The Liveness Specification

The liveness property we expect to be upheld in *System* can be expressed in LTL as follows.

$$\begin{array}{l} \mathit{SEF} \Rightarrow \bigwedge_{\mathit{specimen} \in \mathit{Capability}} \\ \left(\begin{array}{l} \Box\, \mathsf{LiveDriver.Guard.Call}.\mathit{specimen} \Rightarrow \\ \quad \Diamond(\mathsf{Guard.LiveDriver.Return.null} \lor \mathsf{Guard.LiveDriver.Return.Guard}) \end{array} \right) \end{array} \tag{6.5}$$

It asserts that, under the assumption of strong event fairness, for all $\mathit{specimen} \in \mathit{Capability}$, whenever LiveDriver Calls Guard passing *specimen*, eventually Guard should Return to LiveDriver.

Observe that *System* satisfies the liveness property from Equation 6.5 iff, for all traces s of *System* whose last event is from the set $\{\mathsf{LiveDriver.Guard.Call}.\mathit{specimen} \mid \mathit{specimen} \in \mathit{Capability}\}$, we have[6]

$$\begin{array}{l} (\mathit{System} \,/\, s) \models \\ \quad \mathit{SEF} \Rightarrow \\ \quad\quad (\Diamond\, \mathsf{Guard.LiveDriver.Return.null} \lor \Diamond\, \mathsf{Guard.LiveDriver.Return.Guard}). \end{array}$$

Letting $R = \{\mathsf{Guard.LiveDriver.Return.null}, \mathsf{Guard.LiveDriver.Return.Guard}\}$, observe that $\mathit{behaviour}(\mathsf{Guard}) \setminus (\Sigma - R)$ is divergence-free. (Alternatively, FDR can be used to confirm this.) For any process P, $P \setminus X$ being divergence-free implies that $\forall t \in \mathit{traces}(P) \bullet (P \,/\, t) \setminus X$ is divergence-free. Hence, applying Corollary 6.1.8, we see that *System* satisfies the liveness property from Equation 6.5 if

$$\begin{array}{l} \nexists\, s, t, c \bullet c \in C \land t \upharpoonright R = \langle\rangle \land \\ \qquad\qquad (s \mathbin{\hat{}} \langle c \rangle \mathbin{\hat{}} t, \alpha(\mathsf{Guard})) \in \mathit{failures}(\mathit{System}), \end{array} \tag{6.6}$$

where $C = \{\mathsf{LiveDriver.Guard.Call}.\mathit{specimen} \mid \mathit{specimen} \in \mathit{Capability}\}$ is the set of events representing LiveDriver Calling Guard.

We therefore want to test whether *System* can refuse all events from $\alpha(\mathsf{Guard})$ in any stable-state that it can reach in between Guard being Called by LiveDriver (*i.e.* performing an event from C) and Guard subsequently Returning to LiveDriver (*i.e.* performing an event from R). Inspired by the specification in Snippet 6.1, we build a specification *LiveGuardSpec* that exhibits no such refusals and appears in Snippet 6.4. *LiveGuardSpec* is parameterised by the set of events that should not be refused in between Guard being Called by LiveDriver and subsequently Returning.

[6] Recall that $P \,/\, s$ denotes the process that P evolves to after performing the trace s.

$$
\begin{aligned}
&\mathit{LiveGuardSpec}(A) = \\
&\quad \big(?e : \Sigma \to \mathbf{if}\ e \in C \to \mathit{LiveGuardSpec}'(A)\ \mathbf{else}\ \ \mathit{LiveGuardSpec}(A)\big) \\
&\quad \sqcap \mathit{STOP},
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{LiveGuardSpec}'(A) = \\
&\quad ?e : \Sigma - A \to \mathit{LiveGuardSpec}'(A) \triangleright \\
&\qquad \big(\$a : A \to \mathbf{if}\ a \in R\ \mathbf{then}\ \ \mathit{LiveGuardSpec}(A)\ \mathbf{else}\ \ \mathit{LiveGuardSpec}'(A)\big)
\end{aligned}
$$

Snippet 6.4: The specification for testing the liveness of a guard.

Initially, $\mathit{LiveGuardSpec}(A)$ allows arbitrary non-divergent behaviour until an event from the set C of events representing LiveDriver Calling Guard occurs. It then evolves to the process $\mathit{LiveGuardSpec}'(A)$. $\mathit{LiveGuardSpec}'(A)$ is similar to *Spec* from Snippet 6.1 in that it allows all refusals except those in which the entirety of A is refused, until Guard Returns to LiveDriver by performing some event from R. Our liveness property is then upheld if

$$\mathit{LiveGuardSpec}(\alpha(\mathsf{Guard})) \sqsubseteq_F \mathit{System}.$$

Testing this assertion in FDR reveals that it does not hold. Examining the counter-example indicates that the system can perform the trace of events $\langle \mathsf{Stamped.SlotRead.Call.null}, \mathsf{LiveDriver.Guard.Call.Specimen} \rangle$ before deadlocking. Here, before Guard invokes Slot, Stamped invokes Slot's read-facet before choosing to refuse to accept Slot's Return message, thereby preventing anyone else from Calling Slot. This prevents Guard from doing so, causing the system to deadlock. Stamped has, in effect, conspired to mount a denial-of-service attack against Guard.

Refining the Implementation

Preventing this attack requires that each stamped object never refuses to accept a Return message from any of its slots, otherwise it can cause its guards to stop responding.

Some thought (and experimentation with FDR) reveals that this attack is equally possible when a stamped object is permitted to hand out capabilities that refer to its slots to untrusted objects, since this allows an untrusted object to deadlock a slot in the same way that Stamped does above. We refine the behaviour of a stamped object, $\mathit{AStamped}(\mathit{facets}, \mathit{slotW}, \mathit{caps})$, with facets *facets*, slot write capability *slotW* and capabilities *caps*, to include these restrictions by giving it an extra parameter, *slotR*, that is (a capability to) its slot's read-facet. It never refuses to accept a Return message originating from its slot (*i.e.* one for which the reply capability is *slotW* or *slotR*) and never passes *slotW* or *slotR* to any other object. This new behaviour appears in Snippet 6.5.

$$
\begin{array}{l}
AStamped(facets, slotW, slotR, caps, data) = \\
\quad \textbf{let}\ \ caps' = caps - \{slotW, slotR\}\ \textbf{within} \\
\quad \left(\begin{array}{l}
?me : facets?to : caps'?op?arg : caps' \cup data \cup \{\mathsf{null}\} \rightarrow \\
\quad AStamped(facets, slotW, slotR, caps, data)\ \Box \\
?me : facets!slotW!\mathsf{Call}?arg : facets \rightarrow \\
\quad AStamped(facets, slotW, slotR, caps, data)\ \Box \\
?from : Capability - facets?to : facets?op?arg \rightarrow \\
\quad \textbf{let}\ \ C' = \{arg, from\} \cap Capability\ ;\ D' = \{arg\} \cap Data\ \textbf{within} \\
\quad AStamped(facets, slotW, slotR, caps \cup C', data \cup D')\ \Box \\
?from : facets?to : facets?op?arg : caps \cup data \cup \{\mathsf{null}\} \rightarrow \\
\quad AStamped(facets, slotW, slotR, caps, data)
\end{array}\right) \\
\quad \triangleright ?from : \{slotR, slotW\}?me : facets!\mathsf{Return}?arg \rightarrow \\
\quad \textbf{let}\ \ C' = \{arg, from\} \cap Capability\ ;\ D' = \{arg\} \cap Data\ \textbf{within} \\
\quad AStamped(facets, slotW, slotR, caps \cup C', data \cup D')
\end{array}
$$

Snippet 6.5: The behaviour of a live stamped object with multiple facets.

We again use the timeout operator to ensure that this process can never refuse to perform any of the events in the set $\{|from.me.\mathsf{Return} \mid from \in \{slotW, slotR\} \wedge me \in facets|\}$, which models the restriction that a stamped object can never refuse to be Returned to from its slot. In practice, every object in a single-threaded object-capability language automatically obeys this restriction, since in that context it is impossible for any object to refuse to be Returned to. In the case of an object-capability operating system, this restriction can be adhered to simply by ensuring that stamped objects always invoke their slots in a strict send-and-receive sequence.

When performing the check against the system with this new behaviour for Stamped, FDR indicates that it does not hold and returns the following stable-failure of *System*.

$$
\left(\begin{array}{l}
\langle \mathsf{Stamped.SlotWrite.Call.Stamped}, \mathsf{LiveDriver.Guard.Call.Specimen} \rangle, \\
\Sigma - \{\mathsf{SlotWrite.Stamped.Return.null}\}
\end{array}\right).
$$

Here, Guard becomes blocked after it Calls Stamped while waiting for Stamped's prior Call to SlotWrite to Return. While this counter-example looks similar to the previous one, it is actually quite different. Unlike with the previous counter-example, the entire system has not deadlocked here because the event SlotWrite.Stamped.Return.null is not included in the refusal (and FDR indicates that it is in fact available here). This behaviour clearly violates the sufficient condition for our liveness property. However, it isn't clear that this behaviour violates the liveness property itself. For instance, under strong event fairness, so long as Guard isn't blocked continually then our liveness property is not violated by this behaviour.

Given that Guard has become blocked here (perhaps only temporarily) while waiting for Slot to Return to Stamped, an obvious question to ask is whether Guard can ever become blocked in any other circumstance, *i.e.* become blocked while an event representing Slot Returning to Stamped is also refused. Let $D = \{|x.y.\mathsf{Return} \mid x \in \mathit{facets}(\mathsf{Slot}), y \in \mathit{facets}(\mathsf{Stamped})|\}$ be the set of all such events. We can test this by testing whether

$$\mathit{LiveGuardSpec}(\alpha(\mathsf{Guard}) \cup D) \sqsubseteq_F \mathit{System}. \tag{6.7}$$

FDR reveals that this test does indeed hold. Hence, we can conclude that whenever all of Guard's events are refused after it has been Called but before it Returns, that some event from D must be available.

By Lemma 6.1.6, Guard cannot Return due to the presence of some infinite refusal-trace s that satisfies *SEF* but violates our liveness property only if s has an infinite suffix t in which all of Guard's events are refused continually. In t, then, some event from D must be continually available. Since $s \models \mathit{SEF}$, some event from D must occur infinitely often in t. Suppose that we can show that whenever some event from D occurs, no event from D can be immediately stably accepted. Then the existence of s would be impossible.

In this case, we would need only show that the system can't deadlock (*i.e.* stably refuse all of Σ) before Guard Returns. This is already implied, however, by the refinement check above having passed, since this rules out the possibility of $\alpha(\mathsf{Guard}) \cup D$ being refused and refusals are subset-closed by Axiom **F2**.

We can complete our liveness analysis, therefore, by showing that in (all refinements of) *System* whenever some event from D occurs, no event from D can be immediately stably accepted. This is implied by the absence of certain traces, namely those in which two consecutive D events occur. We therefore define the most general process that can never perform two consecutive events from D as follows.

$$\mathit{Spec} = ?x : (\Sigma - D) \rightarrow \mathit{Spec} \,\Box\, ?d : D \rightarrow ?x : (\Sigma - D) \rightarrow \mathit{Spec}$$

We then test simply whether $\mathit{Spec} \sqsubseteq_T \mathit{System}$. FDR reveals that this test holds. From this, we conclude that *System* satisfies our liveness property from Equation 6.5.

Repeating the safety checks performed earlier in Section 3.1, reveals that this implementation of the Trademarks pattern is still safe. Hence, we conclude that non-blocking sends can be used to implement Trademarks that are both safe and live.

6.2.2 Summary

We've shown that FDR can be applied to derive and verify a live Trademarks implementation by testing sufficient conditions for the liveness of that implementation. However, because we could use FDR only to test the stronger

sufficient condition, rather than the liveness property itself, we found that FDR returned counter-examples that violated the sufficient condition but not (necessarily) the liveness property.

We therefore had to go to some extra trouble, by performing some extra checks using FDR, to make sure that these counter-examples didn't represent true violations of the liveness property. Not only did this entail extra effort, but the process was quite ad hoc; it isn't clear that the same strategy that we took could be applied in other examples. Therefore, it is difficult to say whether the overall strategy of using FDR to reason about liveness properties by testing sufficient conditions for them, could be widely applied.

We conclude, therefore, that using FDR to reason about liveness properties in this way requires more effort than one would like, and may not scale well to other object-capability patterns whose structure and behaviour may be vastly different to our Trademarks implementation. We argue that it is probably worthwhile investigating alternative approaches to testing liveness properties under fairness assumptions. One promising kind of approach (see *e.g.* [Ros01, SLDW08, Liu09], all of whose notions of fairness are different to ours as explained later in Section 6.4), examines the operational semantics of a system directly to identify certain strongly connected subgraphs[7] within it that represent infinite fair behaviours that violate some liveness property. We suggest that this approach could be adapted to test liveness properties under our notions of fairness. We briefly discuss how later in Section 8.1.

6.3 Generalising Liveness Analyses

We now consider how we might generalise the liveness analysis performed in this chapter by applying the approach from Chapter 4, which was used there to generalise the safety results obtained in Chapter 3. Note that, unlike in Section 5.5, because the properties we are testing here are all refinement-closed in the stable-failures model, we can apply the approach taken in Chapter 4 directly.

In particular, recall (from Section 4.1.1) that every aggregation is a safe abstraction with regards to properties that are refinement-closed in the traces, stable-failures and failures-divergences models. Since the properties tested with FDR in this chapter are refinement-closed in the stable-failures model, we can therefore be confident that if such a property holds for an aggregation, the property will also hold for every system that the aggregation aggregates. Hence, aggregations are safe abstractions with respect to the properties tested with FDR in this chapter.

[7] A directed graph is *strongly connected* when each node is reachable from all others.

6.3.1 Generalising the Live Trademarks Analysis

We generalise the analysis of the system depicted in Figure 6.1, to all systems that take the form of Figure 6.2. By symmetry, if this system can be proved live, we can conclude that a guard will always return to a caller who is willing to accept the return.

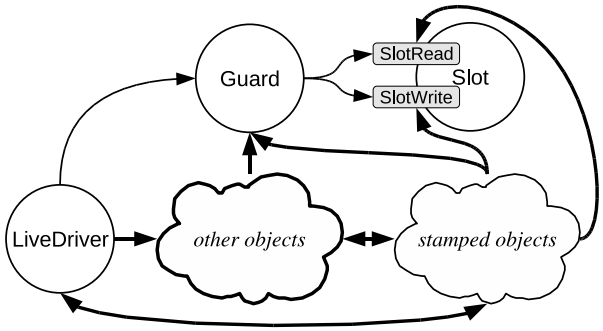

Figure 6.2: Generalising the Trademarks liveness analysis.

Let T denote the set of facets of objects in the "other objects" cloud and U denote the set of facets for the "stamped objects". Then we build a safe abstraction of this system by instantiating the system depicted in Figure 6.1, setting $\mathit{facets}(\mathsf{Specimen}) = T$ and $\mathit{facets}(\mathsf{Stamped}) = U$, using the updated definition of a stamped object's behaviour, *AStamped*, from Snippet 6.5 that is an accurate aggregation of multiple live stamped objects.

Calculating the Thresholds

Recall that this system does not naturally satisfy $\mathbf{NoEqT}_T$ or $\mathbf{NoEqT}_U$ because Guard can perform equality tests on members of both of these types. It can be difficult to find low, tractable, data-independence thresholds for Systems that don't satisfy **NoEqT**. We worked around this problem when generalising the Trademarks safety analysis earlier, in Section 4.1.3, by building an anti-(traces)-refinement of the process *AGuard* that satisfied the weaker property **PosConjEqT**, which allows processes to perform only those equality tests in which they always become $STOP$ whenever a test fails.

To take the same approach here, we would need to build a stable-failures anti-refinement of Guard's behaviour that satisfies the stable-failures version of **PosConjEqT**, which differs from the traces version of **PosConjEqT** used above in that processes must become **div** rather than $STOP$ after a failed equality test. Recall that Equation 4.2 was used to derive a threshold of 2 for the traces refinement check that asserted the safety of the **PosConjEqT** Trademarks implementation. To our knowledge, while we strongly suspect it to be true, the stable-failures analogue of Equation 4.2 has never been proved, however. This prevents us from taking the **PosConjEqT** approach for liveness properties at this time.

Therefore, the best we can do here is to apply standard data-independence theorems that are designed for generalising stable-failures refinement checks performed for systems that satisfy neither **NoEqT** nor **PosConjEqT**. Applying the appropriate theorem (namely Theorem 15.2.4 from [Ros97]) gives data-independence thresholds for T and U of 7.

Making the Refinement Checks Tractable

A little bit of care is needed in order to allow the resulting refinement checks implied by these thresholds to be carried out. In particular, the specification process *LiveGuardSpec* contains few states but many transitions between those states. This means that this process becomes very difficult for FDR to compile as the sizes of T and U increase. In order to remedy this, we express the refinement from Equation 6.7 equivalently using a specification process that avoids this problem.

We instead perform the equivalent check

$$LiveGuardSpec2 \sqsubseteq_F (System \underset{\Sigma-\alpha(\mathsf{Guard})-D}{\parallel} CHAOS_{\Sigma-\alpha(\mathsf{Guard})-D}) \setminus \Sigma - I,$$

where $I = \{|\mathsf{LiveDriver.Guard.Call}, \mathsf{Guard.LiveDriver.Return}|\}$ is the set of events representing communication between LiveDriver and Guard, and the specification *LiveGuardSpec*2 is

$$\begin{array}{l} LiveGuardSpec2 = \\ \quad \mathsf{LiveDriver!Guard!Call?}specimen : Capability \rightarrow \\ \quad \left(\begin{array}{l} \mathsf{Guard!LiveDriver!Return!Guard} \rightarrow LiveGuardSpec2 \sqcap \\ \mathsf{Guard!LiveDriver!Return!null} \rightarrow LiveGuardSpec2 \end{array}\right) \\ \quad \sqcap STOP. \end{array}$$

On the right-hand-side of this refinement, we use *lazy abstraction* [Ros97, Section 12.1.1] to abstract away the occurrence of all events other than those from $\alpha(\mathsf{Guard}) \cup D$ that we want *System* to not be able to refuse in between Guard being Called by LiveDriver and subsequently Returning. We then hide all events not relevant to the specification *LiveGuardSpec*2. The specification asserts that the process on the left-hand-side cannot refuse Guard Returning to LiveDriver after LiveDriver has Called Guard. The laws of hiding in the stable-failures model [Ros97, Section 8.4] imply that such a refusal occurs if and only if *System* can refuse the entire set $\alpha(\mathsf{Guard}) \cup D$ in between Guard being Called and subsequently Returning. Hence, this test is equivalent to that in Equation 6.7.

The process *Spec* that asserts that no two consecutive D-events can occur also has the same problem as *LiveGuardSpec*. Observe, however, that in order to test that no two consecutive D-events can occur, we can partition the set Σ of all events into two equivalence classes: those events that are, and aren't, in D respectively. We can then rename the events according to these

equivalence classes, *i.e.* we rename all D events to some fresh event d and all other events to some fresh event o. Using the CSP renaming operator, this renaming can be expressed as $System[\![\mathsf{d}, \mathsf{o}/d \in D, o \in \Sigma - D]\!]$. We then need to assert that this process cannot perform the event d consecutively, while being sure that the only other event it can perform is o. This is captured by the specification

$$Spec2 = \mathsf{o} \rightarrow Spec2 \,\Box\, \mathsf{d} \rightarrow \mathsf{o} \rightarrow Spec2.$$

Observe that the number of states and transitions in $Spec2$ remains constant as the sizes of T and U increase. We then simply test that

$$Spec2 \sqsubseteq_T System[\![\mathsf{d}, \mathsf{o}/d \in D, o \in \Sigma - D]\!].$$

These tests are certainly within FDR's capabilities for the given thresholds. Carrying out both of these tests when $|T| = |U| = 7$ take about 3 and-a-half hours to complete in total. The other tests, for which $|T| < 7 \vee |U| < 7$, are of course much less expensive to perform. Hence, whilst certainly being sub-optimal, this approach isn't entirely impractical.

6.3.2 Summary

We conclude that with the current body of theory, generalising liveness results for systems that do not satisfy **NoEqT**, including for patterns that make use of EQ, is possible, but may be impractical in some cases.

Note that the situation is much better for systems that satisfy **NoEqT**, such as the Sealer-Unsealer implementation from Section 3.2. In this case, we can obtain data-independence thresholds for each data-independent type of no more than 2 [Ros97, Theorems 15.2.1 and 15.2.2]. A threshold of 2 yields not only systems whose state-spaces are small enough that they are quick for FDR to explore, but also specifications that are simple enough for FDR to compile that refinement checks need not be re-expressed equivalently to make them tractable, as we did with $LiveGuardSpec2$ and $Spec2$ above.

Because the vast majority of object-capability patterns don't make use of EQ, it is likely that most systems under analysis will satisfy **NoEqT**. Hence, we conclude that generalising liveness analyses using the techniques described in this thesis is fairly practical for this most common case, while likely being tractable if not entirely practical otherwise.

6.4 Related Work

Liveness Analyses of Object-Capability Patterns To our knowledge, our work represents the first liveness analysis of an object-capability pattern. Note that Spiessens' Scoll [Spi07] formalism allows one to reason about what are called liveness *possibilities* of object-capability patterns. Unlike liveness

properties, which assert that something must happen, liveness possibilities assert that something is not impossible. Liveness possibilities are, therefore, much weaker than liveness properties. Scoll is unable to reason about liveness properties, unlike our approach.

The Expressiveness of CSP Refinement Theorem 6.1.4 is a generalisation of a result from Roscoe's paper [Ros05] examining the expressiveness of failures-divergences refinement. Here, Roscoe shows that all properties that are not *closed* over the normal topology for CSP processes, cannot be expressed as refinement checks of the form $F(P) \sqsubseteq_{FD} G(P)$ without the use of something like divergence-creating hiding in $G(\_)$ or possibly running multiple copies of P in $F(\_)$[8].

Informally, a property is *closed* when, given a sequence of processes $\langle P_k \mid k \in \mathbb{N} \rangle$ that converges to a process P^*, if each P_k satisfies the property, then so does P^*. Observe that the liveness property $\Diamond e$ is not closed, since it is satisfied by $P_0 = e \rightarrow STOP$ and every $P_k = a \rightarrow P_{k-1}$ for $k > 0$, but is not satisfied by $P^* = a \rightarrow P^*$. This explains why divergence-creating hiding has to be used in order to frame this property as a refinement check (see Equation 6.1).

Expressing LTL Properties as Refinement Checks We saw that for some properties ϕ (like $SEF \Rightarrow \Diamond e$), expressed in our fragment of LTL, one could not express $P \models \phi$ for an arbitrary process P as a refinement check in any standard denotational model for CSP that FDR might support. In [Low08], Lowe shows that for any property ϕ that can be expressed in the *bounded positive* fragment of LTL, that one can express $P \models \phi$ in terms of a refinement-check $Spec \sqsubseteq_R P$ carried out in the refusal-traces model. The bounded-positive fragment of LTL purposefully omits the $\Diamond$ operator.

Liveness and Fairness in CSP Previous approaches (see *e.g.* [PV01, Puh03, Puh05, SLDW08, Liu09]) to testing liveness properties ϕ under fairness assumptions ψ for CSP have generally produced methods in which the resulting properties $\psi \Rightarrow \phi$ are not tested by expressing them as CSP refinement checks, but rather by examining the operational semantics of a system. These approaches have the advantage over ours that one can test a liveness property, like $SEF \Rightarrow \Diamond e$, directly without having to resort to testing sufficient conditions for it instead, as we've shown one must when trying to test these properties using refinement-checking. However, these previous approaches have other drawbacks when compared to ours.

Most of these previous approaches have defined the semantics of LTL over (linear unwindings of) a process's operational semantics. This has two related drawbacks, when compared to our approach. Firstly, as demonstrated by Puhakka *et al.* [PV01, Puh03, Puh05], this can lead to processes

[8] $G(\_)$ must not be *uniformly continuous* or $F(\_)$ must not be *distributive* [Ros05].

that are semantically equivalent being distinguished by LTL properties unless care is taken to ensure that all such properties respect the congruences of the ordinary denotational semantic models of CSP.

Consider, for example, the semantically equivalent processes $P \setminus \{b\}$ and Q, where $P = b \rightarrow a \rightarrow P$ and $Q = a \rightarrow Q$. Despite being semantically equivalent, under most operational characterisations of $\mathsf{available}\ a$ (see *e.g.* [PV01, Puh03, Puh05, SLDW08, Liu09]), $Q \models \mathsf{available}\ a$ while $P \not\models \mathsf{available}\ a$. Hence, $Q \models \mathit{WEF} \Rightarrow \Box\Diamond a$ while $P \not\models \mathit{WEF} \Rightarrow \Box\Diamond a$. Our approach does not suffer from this problem because we borrow our LTL semantics from Lowe [Low08], who defined it over a process's representation in the refusal-traces model. Hence, it necessarily respects the congruences of this model.

Secondly, most previous definitions (notably from those papers besides Lowe's cited in the previous paragraph) of the predicate $\mathsf{available}\ a$, that have been defined in terms of an operational semantics, have omitted any requirement that a be available from a stable state. This then leads to liveness properties under fairness assumptions that give unintuitive results. Consider, for example, the process $R = a \rightarrow R \triangleright b \rightarrow \mathit{STOP}$. Under our definition of $\mathsf{available}\ a$, $R \not\models \mathsf{available}\ a$ because a is available only from an unstable state in R. However, under the previous operational definitions of $\mathsf{available}\ a$, $R \models \mathsf{available}\ a$. Hence, under these previous definitions, $R \models \Box\,\mathsf{available}\ a$.

Under such a definition of $\mathsf{available}\ a$, $R \models \mathit{WEF} \Rightarrow \Diamond a$. However, this is problematic because R is refined, in all standard denotational CSP models, by the process S where $S = b \rightarrow \mathit{STOP}$. Clearly $S \not\models \mathit{WEF} \Rightarrow \Diamond a$ under any sane semantics. This means that these operational interpretations of $\mathsf{available}\ a$ that omit the requirement that a be stably available, yield properties that assert liveness under fairness that are not refinement closed. Our approach avoids this problem by requiring stable availability.

In [SLDW08, Liu09], an algorithm is developed for testing liveness properties under fairness assumptions, like $\mathit{SEF} \Rightarrow \Diamond e$, with *SEF* interpreted under a definition of availability that does not require stability. This algorithm, essentially, tests for the absence of certain strongly connected subgraphs (SCSs) within the operational semantics of a system being analysed. The algorithm identifies those SCSs that represent infinite behaviours that satisfy the fairness assumption (*e.g.* *SEF*) but violate the liveness property (*e.g.* $\Diamond e$) being tested under this fairness assumption.

Roscoe has also considered the problem of testing certain liveness properties under fairness assumptions of CSP systems that model shared variable programs [Ros01]. Roscoe frames tests for liveness as tests that assert the absence of certain stable-failures and certain *fair divergences*. A *fair divergence* with respect to some set of events A is an infinite sequence t of events that can be performed by a process, where t contains infinitely many occurrences of each event $a \in A$.

FDR has been augmented with an algorithm to test for the absence of fair divergence. This algorithm tests for the absence of certain strongly connected components[9] in a process's operational semantics [Ros01], such as those that, for each event $a \in A$, have an a-labelled edge.

Based on the similarity of these two operational approaches for testing liveness properties under fairness assumptions, we conjecture that these approaches might be adapted to test liveness properties under fairness assumptions as defined in this chapter, in order to overcome the limitation that such properties cannot be tested by refinement checking. We briefly discuss how later in Section 8.1.

6.5 Conclusion

In this chapter, we have considered how liveness properties of object-capability patterns can be checked using FDR. We have seen that, as with previous work on liveness, one needs to make fairness assumptions in order to rule out certain kinds of infinite behaviour that would not arise in any fair implementation but would otherwise violate the liveness property in question. We proved that it is impossible to frame certain tests for liveness under reasonable notions of fairness in terms of CSP refinement checks for FDR. In particular, we proved that no refinement check in any standard CSP model that FDR might support can express properties like $SEF \Rightarrow \Diamond e$, where SEF denotes the fairness property of *strong event fairness* (see Definition 6.1.2). Instead, we derived some sufficient conditions for testing properties like $SEF \Rightarrow \Diamond e$ that were stated in terms of the absence of certain stable-failures. These sufficient conditions can be expressed as stable-failures refinement tests for FDR to carry out.

We applied this work to examine the liveness of the Trademarks pattern, arguing that non-blocking communication must be used in order to build a live implementation of this pattern. We modelled the use of such non-blocking communication and defined an intuitive liveness property that asserted that a guard should always respond to a caller in a finite amount of time under the assumption of strong event fairness. We used FDR to derive an implementation of the Trademarks pattern that satisfied this property, by having FDR test sufficient conditions for it. While we could successfully prove that an improved Trademarks implementation satisfied this property, we had to go to some extra trouble when analysing it in order to rule out spurious counter-examples returned by FDR that violated the sufficient condition being tested but not the property itself. This process of ruling out these spurious counter-examples was somewhat ad hoc; it is unclear how well this approach would apply to other object-capability patterns.

We also considered how to generalise liveness results, and showed how to generalise the liveness analysis of the Trademarks pattern to systems of

[9] A *strongly connected component* is a maximal strongly connected subgraph.

arbitrary size. This was made more difficult because this pattern makes use of *EQ*, which prevented the system from satisfying **NoEqT** and resulted in data-independence thresholds being obtained that were higher than we would have liked. We conclude that it is more difficult, although probably still feasible, to generalise liveness results for patterns that make use of *EQ* than those that don't. We expect that liveness results can be easily generalised, however, for the vast majority of patterns, like the Sealer-Unsealer implementation and others analysed in this thesis, that don't use *EQ*.

We conclude that using FDR as described in this chapter can be a workable approach for analysing the liveness properties of some object-capability patterns. However, it requires more effort than one would like and it is unclear how well it could be applied in general. We argue that alternative approaches that examine the operational semantics of a system directly (see *e.g.* [Ros01, SLDW08, Liu09]) should be investigated for testing the liveness properties of object-capability patterns, and that such previous work might be able to be adapted for this purpose. We briefly sketch one such possibility later in Section 8.1.

7 Authority: Exploring Causation

In this second-to-last chapter, we consider the problem of how to reason about authority in object-capability systems. Adopting Miller's definitions [Mil06], an object's *authority* in some system is the collection of effects that it can cause to occur through its overt interactions (*i.e.* through sending and receiving messages) with other objects in that system. Many object-capability patterns are designed to provide certain objects with certain kinds of authority, or to control the authority of certain objects in some way. A full formal analysis of these patterns therefore requires a formal framework for reasoning about authority.

As a simple example, consider Figure 7.1. Here, we see three objects: Alice, Bob and Carol. Suppose that the system is such that initially Alice Calls Bob, which causes Bob to Call Carol. We say then that Alice has the authority to cause Bob to Call Carol. We seek a formal framework in which this statement, and other more complex ones, can be made and proved. Such a framework will need to be able to express properties that capture what it means for one object to cause something to happen, and should ideally be a framework for expressing general properties about causation.

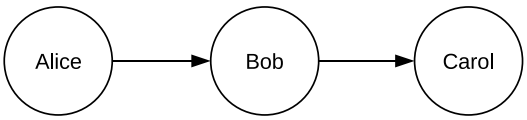

Figure 7.1: A simple example of causation and authority.

A more elaborate example that we will consider later in this chapter is the security-enforcing pattern known as the *Non-Delegatable Authority* [MG08] (NDA). The NDA is a pattern for granting an object o_1 the authority to invoke another o_2, while preventing o_1 from being able to *delegate* to some object o_3 (by passing it a capability) the same authority that o_1 has to invoke o_2. Our framework for reasoning about authority should also allow us to reason about this security property, and others like it.

In this chapter we develop a framework in which properties involving causation and authority can be defined, and consider how such properties can be tested using FDR. We begin in Section 7.1 by giving a general definition for non-causation in object-capability systems, akin to Definition 5.2.2,

that encodes that an object's authority is defined to include only those effects that it can cause through its overt interactions. We then present some simple non-causation properties that might be used to instantiate the general definition and show, with reference to some small examples, how they capture simple elements of an object's authority. Then, in Section 7.2, we distill out the commonalities present in these simple non-causation properties to produce a general framework for expressing non-causation properties for reasoning about authority. Our framework can express complex notions of causation that are useful for reasoning about certain kinds of authority that become relevant in the context of certain object-capability patterns. We demonstrate this in Section 7.3 by showing, through a number of examples, how the framework can capture complex forms of authority, namely delegable, non-delegable, revocable and single-use authority.

We show in Section 7.4 that, in this framework, one can distinguish two primitive kinds of effects, namely the *safety* and *liveness* effects respectively, by identifying effects with refusal-traces *hyperproperties* [CS08, CS10]. Then in Section 7.5, we show how Miller's [Mil06] notions of *defensive correctness* and *defensive consistency* can be captured in our framework as non-causation properties for certain kinds of effects. Defensive correctness and consistency are properties that assert that an object helps protect its clients from each other's misbehaviour by bounding the authority of its clients to interfere with each other's interactions with the object. As has been argued by others [Mil06, MWC10], they are very useful properties for an object-capability pattern to satisfy in order to allow its clients to be mutually suspicious of each other, thereby reducing each client's vulnerability overall. Miller argues that defensive correctness incorporates notions of safety and liveness, while defensive consistency incorporates just safety [Mil06, Section 5.6]. Our formalisation of these ideas makes this point explicit by using the distinction between safety and liveness effects from Section 7.4.

Finally, in Section 7.6, we consider the degree to which a non-causation property can be automatically checked using FDR. This is required in order to check that a pattern, like the NDA, properly confines authority, or that an object is defensively consistent or correct. We show that all safety effects can be expressed as refinement checks. When this check is finite-state, one can use FDR to test for the non-causation of some safety effect, or the effect's opposite, in any finite deterministic system using at most two refinement checks. However, we prove that in general it is not possible to check non-causation of even the most simple safety effects for nondeterministic systems using refinement checking with FDR.

7.1 Simple Non-Causation Properties

Before trying to construct a general framework for expressing causation properties, for reasoning about authority in object-capability systems, we

first present a number of basic examples of different kinds of causation in which one might be interested, in order to shed light on the basic kinds of property we want to express.

The job of each of the causation properties we might want to express is to analyse the behaviours of a process $System = \|_{o \in Object}(behaviour(o), \alpha(o))$ that represents an object-capability system in order to decide whether certain causal relationships exist within that system. For instance, we might wish to know whether one object can cause another to send a message to a third, in order to reason about the first object's authority. To do so, we will define a *non-causation property* ϕ that asserts that no such causation exists.

In line with Miller's definition of authority we restrict our attention to overt causation, *i.e.* causation arising from sending and receiving messages between objects. As in Chapter 5, where we restricted our attention to covert flows of information that arise through overt interactions, this means that when considering the refinements of *System*, we will restrict our attention to its deterministic componentwise refinements (see Definition 5.2.1). This leads to the following general definition for non-causation in an object-capability system.[1]

Definition 7.1.1 (Non-Causation for Object-Capability Systems)**.** An object-capability system captured by the CSP process $System = \|_{o \in Object}(behaviour(o), \alpha(o))$ exhibits none of the causation captured by some non-causation property *Prop* iff $\forall\, System_D \in DCRef(System) \bullet Prop(System_D)$.

Observe that this definition implies that in some system *System*, an object has the authority to cause some effect iff it can cause that effect in any of *System*'s deterministic componentwise refinements $System_D \in DCRef(System)$. Hence, an object's authority in *System* really captures its *potential* authority, which is the union of its actual authority in all such $System_D$.

Recall that each such $System_D$ is a deterministic process. We will consider a number of deterministic example systems that exhibit various kinds of causation and show how the various kinds of causation can be captured by various non-causation properties ϕ that we will define. We will see that each of these properties shares a common structure, which can be distilled to produce a general framework for specifying non-causation properties ϕ that can be used in place of *Prop* in Definition 7.1.1.

7.1.1 Causation as Counterfactual Dependence

Like many other definitions for causation within (models of) computational systems (see *e.g.* [Gro05, CHK08, BBDC+09]), we adopt a notion of causality

[1] Note that the results that we obtain later in Section 7.6 are unchanged if this definition is altered to quantify over all refinements of *System*.

rooted in Lewis' [Lew73] idea of *counterfactual dependence*. Consider some factual scenario in which some effect $\mathcal{B}$ is observed in the presence of some potential cause $\mathcal{A}$, and the question of whether $\mathcal{A}$ causes $\mathcal{B}$ here. Roughly speaking, Lewis argues that we should consider the set of *counterfactual* (*i.e.* not corresponding to the reality we have observed) scenarios in which $\mathcal{A}$ is absent and choose the counterfactual scenario from that set that is most alike to the factual scenario. If $\mathcal{B}$ is absent in this counterfactual scenario, then $\mathcal{B}$ is *counterfactually dependent* on $\mathcal{A}$ in the factual scenario and so $\mathcal{A}$ is a cause of $\mathcal{B}$ in that scenario[2].

Depending on one's choice for the kinds of things that $\mathcal{A}$ and $\mathcal{B}$ represent, one can construct various definitions for different kinds of causation for CSP processes. Our focus on capturing an object's authority, via the things it can overtly cause to occur, means that we will restrict our attention to those $\mathcal{A}$ that represent overt activity by a particular object. Hence, we will define properties that assert that the activity of a particular object cannot cause various effects.

7.1.2 Causing Event-Occurrence

We begin with a very simple example. Consider the system depicted in Figure 7.2. Suppose this is a single-threaded system and that Alice is the object that is initially active and that Alice's behaviour is to simply Call Bob, passing the null argument, and then deadlock. Hence

$$behaviour(\mathsf{Alice}) = \mathsf{Alice.Bob.Call.null} \rightarrow STOP.$$

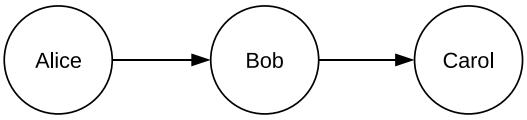

Figure 7.2: A simple example of event causation.

Bob's behaviour is to simply wait to be Called; once Called, Bob Calls Carol, passing the null argument, and then deadlocks. Hence

$$behaviour(\mathsf{Bob}) = \\ ?from : Capability - \{\mathsf{Bob}\}!\mathsf{Bob}!\mathsf{Call}?arg \rightarrow \mathsf{Bob.Carol.Call.null} \rightarrow STOP.$$

Carol simply waits to be Called and then deadlocks, so

$$behaviour(\mathsf{Carol}) = from : Capability - \{\mathsf{Carol}\}!\mathsf{Carol}!\mathsf{Call}?arg \rightarrow STOP.$$

[2]If there are multiple counterfactual scenarios in which $\mathcal{A}$ is absent that are each equally most alike to the factual scenario, then $\mathcal{B}$ must be absent in all of them in order to be counterfactually dependent on $\mathcal{A}$ in the factual scenario.

When composed together to form the depicted system, we see that Alice will simply Call Bob, after which Bob will simply Call Carol. No other behaviours can arise in this system. Hence the system $System1 = \|_{o \in Object}(behaviour(o), \alpha(o))$ will be equivalent to the sequential process in which just these two interactions occur, *i.e.*

$$System1 = \mathsf{Alice.Bob.Call.null} \rightarrow \mathsf{Bob.Carol.Call.null} \rightarrow STOP.$$

It is clear that, in $System1$, Alice's activity causes Bob to Call Carol. Indeed, Bob cannot Call Carol here unless Alice first Calls Bob. Hence, Bob Calling Carol is clearly counterfactually dependent on Alice being present and executing her behaviour.

Able To Occur vs. Must Inevitably Occur

There are two ways in which we might say that Alice causes Bob to Call Carol. We can observe that, in $System1$, Bob *must inevitably* Call Carol. However, in the counterfactual scenario in which Alice doesn't act but the behaviour of the other objects remains unchanged, Bob doesn't inevitably Call Carol. Hence, we might say that Alice causes Bob to *inevitably* Call Carol. Indeed, this counterfactual scenario is captured by the process $System1 \underset{\alpha(\mathsf{Alice})}{\|} STOP$ that is identical to $System1$ except that all of Alice's events are blocked by forcing them to synchronise with the process $STOP$. For any process P and set of events A, let $P\,|_A$ abbreviate $P \underset{A}{\|} STOP$. Observe that $System1\,|_{\alpha(\mathsf{Alice})}$ immediately deadlocks. We capture this kind of causation formally by observing that

$$System1 \models \Diamond\, \mathsf{Bob.Carol.Call.null} \wedge$$
$$System1\,|_{\alpha(\mathsf{Alice})} \not\models \Diamond\, \mathsf{Bob.Carol.Call.null},$$

using the fragment of LTL (see Section 6.1.5) defined over the refusal-traces model (see Section 6.1.4) in Chapter 6. Note that we may often want to incorporate a fairness assumption when deciding whether an event inevitably occurs. For instance, to incorporate the assumption of strong event fairness (see Definition 6.1.2) we would replace the property $\Diamond\, \mathsf{Bob.Carol.Call.null}$ above with $SEF \Rightarrow \Diamond\, \mathsf{Bob.Carol.Call.null}$.

Besides this first kind of causation, we can also observe that in $System1$ the event Bob.Carol.Call.null *is able to* occur at some point, but in the counterfactual scenario $System1\,|_{\alpha(\mathsf{Alice})}$, in which Alice doesn't act, this event can never occur. So we might also say that Alice causes Bob to *be able to* Call Carol. We can capture this sort of causation formally, in the refusal-traces model, by observing that

$$\exists\, s \in \mathcal{R}[\![System1]\!] \cup \mathcal{I}[\![System1]\!] \bullet tr(s) \upharpoonright \{e\} \neq \langle\rangle \wedge$$
$$\nexists\, s \in \mathcal{R}[\![System1\,|_{\alpha(\mathsf{Alice})}]\!] \cup \mathcal{I}[\![System1\,|_{\alpha(\mathsf{Alice})}]\!] \bullet tr(s) \upharpoonright \{e\} \neq \langle\rangle,$$

where $e = \mathsf{Bob.Carol.Call.null}$ and the function $tr(s)$ returns the trace of events performed in a refusal-trace s.

Each kind of causation can be captured simply by a non-causation property that asserts that it cannot occur. For instance, in some system Sys, the object whose alphabet is A does not cause an event e to inevitably occur iff

$$Sys \models \Diamond\, e \Rightarrow Sys\,|_A \models \Diamond\, e. \tag{7.1}$$

Similarly, the object whose alphabet is A does not cause the event e to be able to occur in Sys iff

$$\begin{array}{l} \exists\, s \in \mathcal{R}[\![Sys]\!] \cup \mathcal{I}[\![Sys]\!] \bullet tr(s) \upharpoonright \{e\} \neq \langle\rangle \Rightarrow \\ \quad \exists\, s \in \mathcal{R}[\![Sys\,|_A]\!] \cup \mathcal{I}[\![Sys\,|_A]\!] \bullet tr(s) \upharpoonright \{e\} \neq \langle\rangle. \end{array} \tag{7.2}$$

Note that the presence of neither notion of causation necessarily implies the other. For instance, suppose we have a system comprising four objects, Alice, Bob, Carol and Dave in which Alice, Carol and Dave are initially active. Alice is initially willing only to Call Bob. Bob is initially willing only to be Called by anyone. Carol and Dave initially both try to Call Alice, but each can do so only after Alice has Called Bob. Then the overall behaviour of this system could be captured by the CSP process

$$\begin{array}{l} System2 = \mathsf{Alice.Bob.Call.null} \rightarrow \\ \qquad\qquad \begin{pmatrix} \mathsf{Carol.Alice.Call.null} \rightarrow STOP \,\Box \\ \mathsf{Dave.Alice.Call.null} \rightarrow STOP \end{pmatrix}. \end{array}$$

Here, we see that Bob causes Carol to *be able to* Call Alice, since Carol is not able to Call Alice when Bob doesn't act. However, Bob doesn't cause Carol to *inevitably* Call Alice since $System2$ has the refusal-trace $\langle \bullet, \mathsf{Alice.Bob.Call.null}, \bullet, \mathsf{Dave.Alice.Call.null}, \Sigma \rangle$ that ends in deadlock and so doesn't satisfy $\Diamond\, \mathsf{Carol.Alice.Call.null}$.

Similarly, an object that causes an event to *inevitably* occur need not cause that event to *be able to* occur. Consider a system comprising four objects Alice, Bob, Carol and Dave, in which Alice, Bob and Carol are initially active. Dave is willing to be Called by any object. The object that first Calls Dave determines his subsequent behaviour. If Called initially by Alice, Dave then waits to be Called by Bob and then after that waits to be Called by Carol and then deadlocks. If initially Called by Bob, Dave then waits to be Called by Carol and then deadlocks. If initially Called by Carol, Dave deadlocks immediately. Alice, Bob and Carol each simply try to Call Dave initially. This system could be captured by the CSP process

$$\begin{array}{l} System3 = \\ \quad \mathsf{Alice.Dave.Call.null} \rightarrow \mathsf{Bob.Dave.Call.null} \rightarrow \mathsf{Carol.Dave.Call.null} \rightarrow STOP \\ \quad \Box\ \mathsf{Bob.Dave.Call.null} \rightarrow \mathsf{Carol.Dave.Call.null} \rightarrow STOP \\ \quad \Box\ \mathsf{Carol.Dave.Call.null} \rightarrow STOP. \end{array}$$

Because $System3 \models \Diamond\,\mathsf{Carol.Dave.Call.null}$ and $System3\,|_{\alpha(\mathsf{Bob})} = \mathsf{Alice.Dave.Call.null} \rightarrow STOP \;\Box\; \mathsf{Carol.Dave.Call.null} \rightarrow STOP$ clearly fails to satisfy $\Diamond\,\mathsf{Carol.Dave.Call.null}$, Bob causes Carol to *inevitably* Call Dave. However, since $System3\,|_{\alpha(\mathsf{Bob})}$ also has refusal traces in which Carol.Dave.Call.null does occur, Bob doesn't cause Carol to *be able to* Call Dave here.

7.1.3 Preventing Event-Occurrence

Just as an object can cause some effect to occur, it can also cause that effect to *not* occur. In this case, we say that it *prevents* the effect. Prevention is the natural opposite of causation.

Consider a system comprising three objects Alice, Bob and Carol, in which Alice and Bob both try to Call Carol and Carol is initially willing to be Called by anyone. Suppose that after Calling Carol, both Alice and Bob refuse to send a Return message back to Carol, thereby causing the system to deadlock. This system could be captured by the CSP process

$$System4 = \mathsf{Alice.Carol.Call.null} \rightarrow STOP \;\Box\; \mathsf{Bob.Carol.Call.null} \rightarrow STOP.$$

Then we see that, in $System4$, Alice *prevents* Bob from *inevitably* Calling Carol, since

$$System4 \not\models \Diamond\,\mathsf{Bob.Carol.Call.null} \wedge$$
$$System4\,|_{\alpha(\mathsf{Alice})} \models \Diamond\,\mathsf{Bob.Carol.Call.null}.$$

A non-causation property for this kind of prevention might state that in a system Sys an object with alphabet A cannot prevent an event e from inevitably occurring iff when $Sys \not\models \Diamond\, e$ then $Sys\,|_A \not\models \Diamond\, e$. This can be written equivalently as

$$Sys\,|_A \models \Diamond\, e \Rightarrow Sys \models \Diamond\, e. \tag{7.3}$$

Observe that this equation is very similar to Equation 7.1 except that the direction of the implication has been reversed. Recall also that if one wanted to incorporate a fairness assumption, like strong event fairness, into this definition, then one could replace each occurrence of $\Diamond\, e$ above by *e.g.* $SEF \Rightarrow \Diamond\, e$ (see Definition 6.1.2).

Suppose now that Alice can never refuse to Return to Carol after Calling her and that Carol can never refuse to accept any such Return. Then the system could be captured by the CSP process

$$System5 = \mathsf{Alice.Carol.Call.null} \rightarrow \mathsf{Carol.Alice.Return.null} \rightarrow System5 \;\Box$$
$$\mathsf{Bob.Carol.Call.null} \rightarrow STOP.$$

Suppose we also make the fairness assumption of strong event fairness here. Then $System5 \models (SEF \Rightarrow \Diamond\,\mathsf{Bob.Carol.Call.null})$, *i.e.* under this fairness assumption Bob inevitably Calls Carol in $System5$. Hence, in $System5$, Alice

does not prevent Bob from inevitably Calling Carol under strong event fairness.

This example highlights the difference between this kind of authority and information flow. In any object-capability system that signals an error to Bob when he tries to Call Carol while Carol is busy servicing Alice's Call, Bob will be able to detect that Alice has Called Carol. Hence, we might certainly conclude that Alice can pass information to Bob in *System*5 because she can delay Bob from Calling Carol in a way that is observable to Bob. However, under reasonable fairness assumptions, Alice can delay Bob from being able to Call Carol only for a finite amount of time. Under such fairness assumptions, she is unable, therefore, to prevent Bob from inevitably Calling Carol, despite being able to pass information covertly to Bob through her overt interactions with Carol.

Note that an object can never prevent another from being able to possibly perform some event. Suppose otherwise for a contradiction that in some system *Sys* some object with alphabet A can prevent some event e from possibly being performed. Then $Sys\,|_A$ must have some ordinary trace $s \in traces(Sys\,|_A)$ in which e occurs that is not present in $traces(Sys)$. However, this is clearly impossible since the former is always a traces-refinement of the latter.

7.2 A Framework for Non-Causation Properties

We have shown a number of examples of different kinds of event causation and prevention, and non-causation properties (Equations 7.1, 7.2 and 7.3) designed to capture each. Of course, these non-causation properties are just a fraction of those in which one might be interested when analysing a particular pattern. For many patterns, for instance, one might wish to analyse not whether some object has the authority to cause or prevent the occurrence of an event, but whether it can cause or prevent a second object from causing an event to occur, and so on. (Later, we will see examples of patterns for which this kind of causation is very relevant.)

The non-causation properties that we've seen so far are all very similar to one another. So it seems natural to wonder whether these similarities can be distilled into a framework for specifying other non-causation properties, including those hinted at in the previous paragraph. In this section, we devise such a framework.

7.2.1 Encoding Effects

We begin by considering how to characterise the different effects that our non-causation properties should reason about. We make the straightforward observation that an effect may be characterised by the set of systems (*i.e.* CSP processes) in which it is present. For instance, the effect that is

the inevitable occurrence of the event e is naturally characterised by the set of systems in which e inevitably occurs, *i.e.* the set of systems that satisfy the LTL property $\Diamond\, e$.

A system, in turn, may be characterised by its representation in a CSP denotational semantic model. We saw in Chapter 6 that in order to reason about certain properties, like $SEF \Rightarrow \Diamond\, e$, we needed to use the refusal-traces model. Hence, for the remainder of this chapter, we restrict our attention to the refusal-traces model and stipulate that a system is characterised by its representation within that model.

We may therefore identify each effect with a set E of sets R where each set R is the representation of some divergence-free process P in the refusal-traces model, so that $R = \mathcal{R}[\![P]\!] \cup \mathcal{I}[\![P]\!]$.

It can be shown, however, that any divergence-free process P is captured equally well in the refusal-traces model by its set of *completed* refusal-traces as it is by $\mathcal{R}[\![P]\!] \cup \mathcal{I}[\![P]\!]$. A refusal-trace is completed iff it is finite and ends in deadlock, or it is infinite. The set of all completed refusal-traces is $CRT = DRT \cup IRT$, the union of the sets of deadlocked and infinite refusal-traces respectively. Any divergence-free process P is then captured equally well by the set $\mathcal{C}[\![P]\!] = (\mathcal{R}[\![P]\!] \cup \mathcal{I}[\![P]\!]) \cap CRT$ as it is by $\mathcal{R}[\![P]\!] \cup \mathcal{I}[\![P]\!]$.

The following lemma proves this.

Lemma 7.2.1. For any divergence-free processes P and Q,

$$(\mathcal{R}[\![P]\!] \cup \mathcal{I}[\![P]\!] = \mathcal{R}[\![Q]\!] \cup \mathcal{I}[\![Q]\!]) \Leftrightarrow (\mathcal{C}[\![P]\!] = \mathcal{C}[\![Q]\!]).$$

Proof. Suppose we have two divergence-free processes P and Q. We prove $(\mathcal{R}[\![P]\!] \cup \mathcal{I}[\![P]\!] \neq \mathcal{R}[\![Q]\!] \cup \mathcal{I}[\![Q]\!]) \Leftrightarrow (\mathcal{C}[\![P]\!] \neq \mathcal{C}[\![Q]\!])$. We show the only-if-direction, since the if-direction is trivial. Suppose $\mathcal{R}[\![P]\!] \cup \mathcal{I}[\![P]\!] \neq \mathcal{R}[\![Q]\!] \cup \mathcal{I}[\![Q]\!]$. Without loss of generality, suppose that P has a refusal-trace s that Q does not. Then if $s \in CRT$ the claim follows trivially. If $s \notin CRT$ then $s \in \mathcal{R}[\![P]\!] \cap PRT$ and so, by Equation 6.2, P must have at least one completed extension t of s, where $t \in \mathcal{C}[\![P]\!] \wedge s \leq t$. Recall that $s \notin \mathcal{R}[\![Q]\!]$. By Axiom **R1**, $\mathcal{R}[\![Q]\!]$ is prefix-closed, which implies that $t \notin \mathcal{R}[\![Q]\!] \cup \mathcal{I}[\![Q]\!]$, *i.e.* $t \notin \mathcal{C}[\![Q]\!]$. So $\mathcal{C}[\![P]\!] \neq \mathcal{C}[\![Q]\!]$ as required. □

This means we can identify each effect with a set E that contains sets $\mathcal{C}[\![P]\!]$ of completed refusal-traces of each divergence-free process P in which the effect is present. For instance, letting **CSP** denote the set that contains all divergence-free CSP processes, the effect that is the eventual occurrence of the event e is captured by the set $E_{\Diamond\, e}$ where

$$E_{\Diamond\, e} = \{\mathcal{C}[\![P]\!] \mid P \in \mathbf{CSP} \wedge P \models \Diamond\, e\}. \tag{7.4}$$

The effect that is the possible occurrence of the event e is captured by the set $E_{\mathbf{EF}\, e}$ where[3]

$$E_{\mathbf{EF}\, e} = \{\mathcal{C}[\![P]\!] \mid P \in \mathbf{CSP} \wedge \exists\, s \in traces(P) \bullet s \upharpoonright \{e\} \neq \langle\rangle\}. \tag{7.5}$$

[3]This name is designed to be suggestive of the CTL [CES86] property $\mathbf{EF}\, e$.

We let $\mathit{Effect} = \mathbf{P}\,\{\mathcal{C}[\![P]\!] \mid P \in \mathbf{CSP}\}$ denote the set of all effects. We say that an effect $E \in \mathit{Effect}$ is *present* in a divergence-free process $P \in \mathbf{CSP}$ iff $\mathcal{C}[\![P]\!] \in E$. An effect E is *absent* in P iff $\mathcal{C}[\![P]\!] \notin E$.

The absence of some effect E is itself another effect, which we denote $\overline{E}$, and call the *complement* of E:

$$\overline{E} = \{\mathcal{C}[\![P]\!] \mid P \in \mathbf{CSP} \wedge \mathcal{C}[\![P]\!] \notin E\}. \tag{7.6}$$

7.2.2 Causation and Prevention

From this, we can easily give a general characterisation of non-causation and non-prevention for these kinds of effect.

An object with alphabet $A \subseteq \Sigma$ does not *cause* some effect $E \in \mathit{Effect}$ in system Sys iff

$$\mathcal{C}[\![\mathit{Sys}]\!] \in E \Rightarrow \mathcal{C}[\![\mathit{Sys}\,|_A]\!] \in E. \tag{7.7}$$

An object with alphabet $A \subseteq \Sigma$ does not *prevent* some effect $E \in \mathit{Effect}$ iff it does not cause the absence of that effect, *i.e.* iff $\mathcal{C}[\![\mathit{Sys}]\!] \in \overline{E} \Rightarrow \mathcal{C}[\![\mathit{Sys}\,|_A]\!] \in \overline{E}$, which is equivalent to

$$\mathcal{C}[\![\mathit{Sys}\,|_A]\!] \in E \Rightarrow \mathcal{C}[\![\mathit{Sys}]\!] \in E. \tag{7.8}$$

We refer to any such property obtained by substituting some concrete effect for E in Equation 7.7 or 7.8 as a *non-causation* or *non-prevention property* respectively. Any such property is applied to some system $\mathit{System} = \|_{o \in \mathit{Object}}(\mathit{behaviour}(o), \alpha(o))$ by testing it for all $\mathit{System}_D \in \mathit{DCRef}(\mathit{System})$ in accordance with Definition 7.1.1.

Observe that non-causation (Equation 7.7) may be viewed as just another effect. For instance, the effect E that is the object with alphabet A being unable to cause some effect E_1 may be captured as

$$E = \{\mathcal{C}[\![P]\!] \mid P \in \mathbf{CSP} \wedge \mathcal{C}[\![P]\!] \in E_1 \Rightarrow \mathcal{C}[\![P\,|_A]\!] \in E_1\}.$$

The same also applies to non-prevention (Equation 7.8).

We can also view causation as an effect. The effect E that is the object with alphabet A causing effect E_1 may be defined as

$$E = \{\mathcal{C}[\![P]\!] \mid P \in \mathbf{CSP} \wedge \mathcal{C}[\![P]\!] \in E_1 \wedge \mathcal{C}[\![P\,|_A]\!] \notin E_1\}.$$

Of course we may do likewise for prevention.

As we will see in the following section, treating causation and prevention as effects allows us to reason accurately about some of the more esoteric elements of an object's authority that may be influenced and controlled by various patterns.

7.3 Using the Framework to Capture Authority

One can observe that substituting the definition for the effect $E_{\Diamond e}$ of e inevitably occurring (Equation 7.4) into Equations 7.7 and Equation 7.8, and applying Lemma 7.2.1, would yield equations that are equivalent to Equations 7.1 and 7.3 respectively. The same applies for the effect $E_{\mathbf{EF}\, e}$ (Equation 7.5) of e possibly occurring and Equation 7.2. Hence, the non-causation properties we've encountered so far can be expressed easily in our framework. However, we can also express far more elaborate non-causation properties for reasoning about authority in this framework.

Defining any such property requires us simply to define the effect in which we're interested in detecting whether it can be caused or prevented respectively. We present a number of examples of effects that themselves involve causation, such as detecting when one object o_1 causes another o_2 to cause a third o_3 to perform some action, or when o_1 prevents some effect involving o_2 causing some effect involving o_3.

We show how these subtle kinds of influence manifest themselves in some common patterns for controlling authority in object-capability systems. In doing so, we take the first steps along the path towards formally analysing the ways in which these patterns influence and control the propagation of authority in object-capability systems.

Note that each example system that we will consider is a deterministic process and should be thought of as a deterministic componentwise refinement of the system in which the pattern is instantiated in each case.

7.3.1 Delegable Authority: Capturing the Authority to Delegate One's Authority

Consider the system depicted in Figure 7.3. Here Bob has capabilities to Carol and Dave. Carol and Dave are initially inactive while Bob's behaviour is completely unknown and untrusted. In one deterministic possibility for Bob's behaviour, Bob may repeatedly Call Carol, in which case the entire system could behave equivalently to the deterministic CSP process

$$System6 = \mathsf{Bob.Carol.Call.null} \rightarrow System6.$$

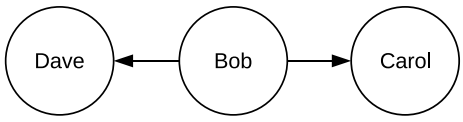

Figure 7.3: Delegable authority.

In this case, we would naturally observe that Bob causes Carol to be repeatedly Called. Indeed, letting $E_{\mathbf{EG}\,\mathbf{EF}\, e}$ denote the effect that is the

event e being able to repeatedly occur, so that

$$E_{\mathbf{EG}\,\mathbf{EF}\,e} = \{\mathcal{C}[\![P]\!] \mid P \in \mathbf{CSP} \wedge \exists\, s \in \mathcal{C}[\![P]\!] \bullet tr(s) \upharpoonright \{e\} = \langle e, e, \ldots\rangle\}, \quad (7.9)$$

we see that

$$System6 \in E_{\mathbf{EG}\,\mathbf{EF}\,\mathsf{Bob.Carol.Call.null}} \wedge System6\,|_{\alpha(\mathsf{Bob})} \notin E_{\mathbf{EG}\,\mathbf{EF}\,\mathsf{Bob.Carol.Call.null}}.$$

Hence, Bob has the authority to cause Carol to be able to be repeatedly Called.

In another possible implementation of Bob, however, Bob's behaviour might instead have him just invoke Dave, passing Dave his capability to Carol, and then deadlock. Suppose that Dave's behaviour is then to repeatedly Call the capability he was passed by Bob. In this case, the system could be captured by the CSP process

$$System7 = \mathsf{Bob.Dave.Call.Carol} \rightarrow System7',$$
$$System7' = \mathsf{Dave.Carol.Call.null} \rightarrow System7'.$$

It is easily confirmed that, in $System7$, Dave causes Carol to be able to be repeatedly Called, since $System7 \in E_{\mathbf{EG}\,\mathbf{EF}\,\mathsf{Dave.Carol.Call.null}} \wedge System7\,|_{\alpha(\mathsf{Dave})} \notin E_{\mathbf{EG}\,\mathbf{EF}\,\mathsf{Dave.Carol.Call.null}}$. Hence, Dave has the authority to cause Carol to be able to be repeatedly Called. However, Dave has this authority here only because Bob passes to Dave a capability to Carol that Dave doesn't initially possess. Hence, we will show that in $System7$, Bob causes Dave to cause Carol to be able to be Called repeatedly, *i.e.* that Bob's authority includes the authority to cause Dave to cause Carol to be able to be repeatedly Called.

We saw above that Bob has the authority to cause Carol to be repeatedly Called. So, we might say that the inclusion in Bob's authority of the authority to cause Dave to cause Carol to be able to be repeatedly Called, captures the idea that Bob's authority includes the ability to *delegate* to Dave the authority over Carol that Bob has. By *delegation*, we mean the simple act in which one object passes one of its capabilities to another object by sending it a message containing that capability [MG08]; however, it should be noted that this is a somewhat restrictive definition of delegation and that other definitions might be more appropriate in other circumstances.

To capture this part of Bob's authority, we first define the effect E_1 that is Dave causing Carol to be able to be repeatedly Called.

$$E_1 = \left\{ \mathcal{C}[\![P]\!] \;\middle|\; P \in \mathbf{CSP} \wedge \begin{array}{l} P \in E_{\mathbf{EG}\,\mathbf{EF}\,\mathsf{Dave.Carol.Call.null}} \wedge \\ P\,|_{\alpha(\mathsf{Dave})} \notin E_{\mathbf{EG}\,\mathbf{EF}\,\mathsf{Dave.Carol.Call.null}} \end{array} \right\}. \quad (7.10)$$

We then capture that Bob causes this effect by noting that

$$\mathcal{C}[\![System7]\!] \in E_1 \wedge \mathcal{C}[\![System7\,|_{\alpha(\mathsf{Bob})}]\!] \notin E_1,$$

since $System7\,|_{\alpha(\mathsf{Bob})} = STOP$.

7.3.2 Non-Delegable Authority

We saw that in the system depicted in Figure 7.3, (1) Bob has the authority to *delegate* to Dave that part of his authority over Carol with which he can cause Carol to be repeatedly Called, by passing his Carol-capability to Dave, and that (2) Bob's authority to delegate this Carol-authority to Dave could be captured in our framework. The ease with which authority may be delegated, simply by passing capabilities, has often been viewed as a negative trait of capability-based systems [MG08]. For instance, some (*e.g.* [WBDF97]) have argued that this means that objects need to be trusted too much, or (*e.g.* [LSM+98]) that it makes *mandatory access controls* [And72] more difficult to enforce, including (as argued in [Kar88]) enforcing *confinement* [Lam73] and (as argued in [Gon89]) the Bell-LaPadula **-property* [BL76].

The *Non-Delegatable Authority* [MG08] (NDA) pattern was proposed to solve this problem. In particular, it can be deployed in the context of the system depicted in Figure 7.3 to allow Bob to Call Carol without allowing him to delegate this authority to Dave. The pattern is deployed by modifying that system to instead have the form of Figure 7.4.

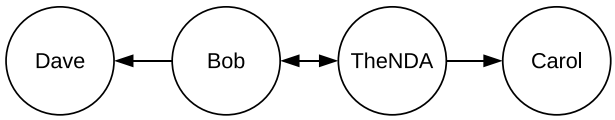

Figure 7.4: Non-delegable authority.

Rather than having direct access to Carol, Bob now has access to TheNDA, an object that implements the NDA pattern and acts as an intermediary between Bob and Carol. We say that Bob is TheNDA's *subject* and that Carol is its *target*. The job of an NDA is to give its subject non-delegable authority to invoke its target. The behaviour of an NDA *me* whose subject is *subject* and whose target is *target* may be modelled by the process $AnNDA(me, subject, target)$, defined in Snippet 7.1.

An NDA waits to be Called by anyone (*e.g.* Bob or any object to whom Bob passes his TheNDA-capability). In response to being invoked, an NDA Calls its subject (in our case Bob). The purpose of this Call is to ask the subject whether it wants the NDA to Call the NDA's target (which, in our case, is Carol). Depending on the result *arg* Returned by the NDA's subject, the NDA may then Call its target. Specifically, an NDA will call its target iff its subject Returns a non-null argument, *i.e.* iff $arg \neq \mathsf{null}$. Having Called its target or not, the NDA then Returns to its original Caller, *from*.

Intuitively, Bob cannot delegate the authority to Call Carol that TheNDA provides to him because this authority is not represented by any capability that Bob possesses, thereby preventing it from being delegable by definition. We can see this as follows. Suppose Bob's behaviour is to simply pass to Dave

$$
\begin{array}{l}
AnNDA(me, subject, target) = \\
\quad ?from : Capability - \{me\}!me!\mathsf{Call}!\mathsf{null} \rightarrow \\
\quad me!subject!\mathsf{Call}!\mathsf{null} \rightarrow subject!me!\mathsf{Return}?arg \rightarrow \\
\quad \textbf{if } arg = \mathsf{null} \textbf{ then} \\
\qquad me!from!\mathsf{Return}!\mathsf{null} \rightarrow AnNDA(me, subject, target) \\
\quad \textbf{else} \\
\qquad me!target!\mathsf{Call}!\mathsf{null} \rightarrow me!from!\mathsf{Return}!\mathsf{null} \rightarrow \\
\qquad\quad AnNDA(me, subject, target)
\end{array}
$$

Snippet 7.1: The behaviour of a Non-Delegable Authority (NDA).[4]

his TheNDA-capability and then refuse to partake in any further interactions with any other object. Dave and Carol behave as before while TheNDA's behaviour is as one would expect, namely $AnNDA(\mathsf{TheNDA}, \mathsf{Bob}, \mathsf{Carol})$. Then the system could be captured by the CSP process

$$System8 = \mathsf{Bob.Dave.Call.TheNDA} \rightarrow \mathsf{Dave.TheNDA.Call.null} \rightarrow STOP.$$

After Dave Calls TheNDA, TheNDA tries to Call Bob, but this call will never succeed since Bob is not willing to be Called by TheNDA. Obviously, since Carol can never be invoked in $System8$, we see that Bob cannot delegate to Dave his authority to cause Carol to be repeatedly invoked here. We have proved that this is the case when each behaves as in $System8$, and we expect this result to hold generally. In order to conclude that Bob has no such authority generally, we would of course need to show that this result holds for all possible behaviours of Bob and Dave, *i.e.* for all deterministic componentwise refinements of this system when Bob and Dave are each an instance of the $Untrusted_{OS}$ process (see Snippet 2.1).

While Bob cannot *delegate* this authority to Dave, he cannot be prevented altogether from sharing it with Dave [Don81, MG08]. Indeed, Bob can still behave in such a way as to allow Dave to cause Carol to be repeatedly invoked. Suppose Bob's behaviour is altered so that initially he passes to Dave his TheNDA-capability and accepts and responds affirmatively to all future invocations. The behaviour of this modified system could then be captured by the CSP process $System9$ defined as follows.

[4] We model a simplified implementation of the NDA (see [MG08] for the full implementation) that doesn't allow Bob to specify the argument that is passed to Carol, nor to receive any value that might be Returned by Carol. These features are unnecessary because Carol accepts only the null argument and doesn't send back Return messages. However, the model could be easily extended to accommodate them if needed.

$$System9 = \mathsf{Bob.Dave.Call.TheNDA} \rightarrow System9',$$

$$System9' = \\ \mathsf{Dave.TheNDA.Call.null} \rightarrow \\ \mathsf{TheNDA.Bob.Call.null} \rightarrow \mathsf{Bob.TheNDA.Return.Bob} \rightarrow \\ \mathsf{TheNDA.Carol.Call.null} \rightarrow \mathsf{TheNDA.Dave.Return.null} \rightarrow \\ System9'.$$

We see that Bob causes Dave to cause Carol to be repeatedly Called in $System9$, since

$$\mathcal{C}[\![System9]\!] \in E_1 \wedge \mathcal{C}[\![System9\,|_{\alpha(\mathsf{Bob})}]\!] \notin E_1,$$

where E_1 is as defined in Equation 7.10 in Section 7.3.1, namely as

$$E_1 = \left\{ \mathcal{C}[\![P]\!] \;\middle|\; P \in \mathbf{CSP} \wedge \begin{array}{l} P \in E_{\mathbf{EG\,EF}\,\mathsf{Dave.Carol.Call.null}} \wedge \\ P\,|_{\alpha(\mathsf{Dave})} \notin E_{\mathbf{EG\,EF}\,\mathsf{Dave.Carol.Call.null}} \end{array} \right\}.$$

The question then arises as to *how* Bob shared this authority with Dave. He certainly can't have shared it by delegating to Dave because Bob delegated to Dave in $System8$ but couldn't share his authority there. So this example doesn't demonstrate that Bob's Carol-authority is delegable. Rather, it hints that in order for Bob to share this authority with Dave, Bob probably needs to repeatedly actively collaborate with Dave or an object acting on his behalf (in our case TheNDA).

This highlights the fundamental difference between delegable and non-delegable authority. Delegable authority can be shared by the single simple irreversible act of delegation, while non-delegable authority can be shared only via repeated collaboration with those objects with whom it is being shared [MG08].

7.3.3 Revocable Authority: Capturing the Authority to Revoke Another's Authority

The previous examples both involved one object having the authority to cause another to cause some effect. We now consider some examples in which one object has the authority to prevent another from causing some effect. The first such example involves one object revoking another's authority. Consider the system depicted in Figure 7.5. It contains the objects Alice, Bob, Carol and Proxy. Proxy is a revocable forwarder or *caretaker* [Red74], designed to grant Bob revocable authority to invoke Carol.

Proxy forwards messages sent to its CForward facet onto Carol. Calling its CRevoke facet causes it to stop forwarding messages to Carol. Once revoked, CForward still continues to process Call messages, however. For simplicity, suppose the behaviour of Alice is to simply Call CRevoke and then deadlock.

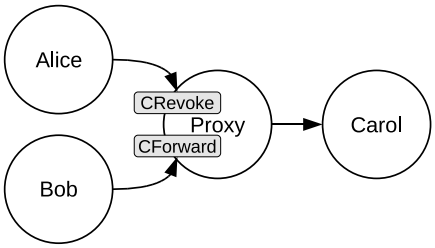

Figure 7.5: Revocable authority with the Caretaker pattern.

Bob's behaviour is to simply repeatedly Call CForward. Carol simply accepts Call messages. Finally, neither Carol nor Proxy send back Return messages when invoked. Then the system could be captured by the CSP process $System10$ where

$$\begin{aligned} System10 &= \mathsf{Alice.CRevoke.Call.null} \rightarrow System10' \,\Box \\ &\quad\; \mathsf{Bob.CForward.Call.null} \rightarrow \mathsf{CForward.Carol.Call.null} \rightarrow System10, \\ System10' &= \mathsf{Bob.CForward.Call.null} \rightarrow System10'. \end{aligned}$$

Here, $System10'$ represents the state that the system evolves to once Proxy has been revoked.

In $System10$, Bob causes Carol to be able to be Called. Indeed, CForward.Carol.Call.null can occur in $System10$ but not in $System10\,|_{\alpha}(\mathsf{Bob})$. We also find that Alice prevents Carol from inevitably being Called, since $System10\,|_{\alpha(\mathsf{Alice})} \models \Diamond\, \mathsf{CForward.Carol.Call.null}$ but $System10 \not\models \Diamond\, \mathsf{CForward.Carol.Call.null}$.

More interesting, however, is that Alice's active presence in $System10$ affects Bob's authority. Without Alice present, Bob *inevitably* causes Carol to be invoked. However, with Alice actively present, it's *possible* for Bob to cause Carol to be invoked but not *inevitable*. We might say, then, that Alice has a subtle kind of authority over Bob, since her active presence modulates Bob's authority. This subtle kind of authority that Alice has over Bob is, of course, a manifestation of Alice's authority to revoke Bob's Carol-authority.

We can capture this in our framework as follows. We say that Bob inevitably causes Carol to be invoked when: (1) Bob causes Carol to be able to be invoked (meaning that Carol cannot be invoked without Bob's presence) and (2) Carol is inevitably invoked in Bob's presence. This effect may be captured as the set E defined as

$$E = \left\{ \mathcal{C}[\![P]\!] \;\middle|\; P \in \mathbf{CSP} \wedge \begin{array}{l} (\mathcal{C}[\![P]\!] \in E_{\mathbf{EF}\,e} \wedge \mathcal{C}[\![P\,|_{\alpha(\mathsf{Bob})}]\!] \notin E_{\mathbf{EF}\,e}) \wedge \\ \mathcal{C}[\![P]\!] \in E_{\Diamond\, e} \end{array} \right\}$$

where $e = \mathsf{CForward.Carol.Call.null}$, using Equations 7.4 and 7.5.

We can then capture that Alice prevents this effect by noting that

$$\mathcal{C}[\![System10]\!] \notin E \wedge \mathcal{C}[\![System10\,|_{\alpha(\mathsf{Alice})}]\!] \in E.$$

$\mathcal{C}[\![System10]\!] \notin E$ since, in $System10$, Carol is not inevitably Called. $\mathcal{C}[\![System10\,|_{\alpha(\mathsf{Alice})}]\!] \in E$ since, in $System10\,|_{\alpha(\mathsf{Alice})}$, Carol is inevitably Called and Bob causes Carol to be able to be Called. Therefore, Alice's authority to revoke Bob's authority over Carol manifests itself as an effect involving Bob causing Carol to be Called that Alice can prevent.

7.3.4 Single-Use Authority

Our final example of preventing an effect involving causation comes from Spiessens [Spi06]. Consider now the system depicted in Figure 7.6, which contains an object OneShot that forwards only the first invocation it receives to Carol; after being invoked once it still accepts future invocations but doesn't forward them to Carol. Carol simply accepts Call messages, while Alice and Bob repeatedly Call OneShot.

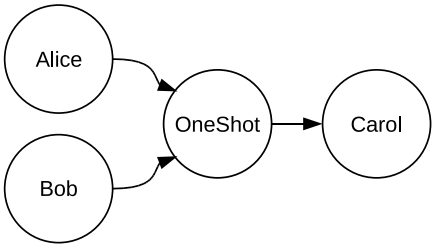

Figure 7.6: A system with a single-use object.

This system could be captured by the CSP process $System11$ where

$$\begin{aligned} System11 = {} & \mathsf{Alice.OneShot.Call.null} \rightarrow \mathsf{OneShot.Carol.Call.null} \rightarrow System11' \\ & \Box \\ & \mathsf{Bob.OneShot.Call.null} \rightarrow \mathsf{OneShot.Carol.Call.null} \rightarrow System11', \\ System11' = {} & \mathsf{Alice.OneShot.Call.null} \rightarrow System11' \,\Box \\ & \mathsf{Bob.OneShot.Call.null} \rightarrow System11'. \end{aligned}$$

This system is similar to $System10$ in that OneShot can be considered as a kind of revocable forwarder that forwards to Carol, which is automatically revoked upon its first use. However, in this system, Bob no longer has the authority to cause Carol to be Called, since $\Diamond\,\mathsf{OneShot.Carol.Call.null}$ holds for both $System11$ and $System11\,|_{\alpha(\mathsf{Bob})}$. This makes sense, since no matter whether Bob acts or not, Carol is inevitably Called. Hence, Bob is powerless to alter whether Carol is inevitably Called and so he should rightly have no authority in this regard.

In $System11$, Alice's active presence is what prevents Bob from having the authority to cause Carol to be invoked. Indeed, if we let E_1 denote the effect that is Bob causing Carol to be able to be Called, so that

$$E_1 = \{\mathcal{C}[\![P]\!] \mid P \in \mathbf{CSP} \wedge \mathcal{C}[\![P]\!] \in E_{\mathbf{EF}\,e} \wedge \mathcal{C}[\![P\,|_{\alpha(\mathsf{Bob})}]\!] \notin E_{\mathbf{EF}\,e}\},$$

where $e = \mathsf{OneShot.Carol.Call.null}$, we see that

$$\mathcal{C}[\![System11\,|_{\alpha(\mathsf{Alice})}]\!] \in E_1 \wedge \mathcal{C}[\![System11]\!] \notin E_1.$$

So Alice prevents Bob from causing Carol to be able to be Called here. Similarly, we could also show that Alice prevents Bob from causing Carol to be inevitably Called here.

However, *System*11 contains a more subtle form of prevention too, which was first noted by Spiessens [Spi06]. This form of prevention captures Alice's authority to revoke Bob's access to Carol, which he has by virtue of OneShot, by Alice using OneShot.

Observe that when Alice Calls OneShot for the first time and does so before Bob has Called OneShot, she does exercise a subtle influence over Bob. Indeed, before Alice Calls OneShot in this way, Bob is able to Call OneShot and have it Call Carol on his behalf. After Alice Calls OneShot, however, Bob's invocation of OneShot has no effect.

This kind of influence can be captured as follows. We note that before Alice Calls OneShot, it is inevitably the case that when Bob Calls OneShot, OneShot then inevitably Calls Carol. After Alice Calls OneShot, this is no longer the case. We capture the effect that is OneShot inevitably Calling Carol in response to Bob's invocation of it as the set E.

$$E = \left\{ \begin{array}{l} \mathcal{C}[\![P]\!] \;\mid\; P \in \mathbf{CSP} \wedge \\ \quad P \models \Diamond(\mathsf{Bob.OneShot.Call.null} \wedge \Diamond\, \mathsf{OneShot.Carol.Call.null}) \end{array} \right\} \tag{7.11}$$

We then note that $\mathcal{C}[\![System11\,|_{\alpha(\mathsf{Alice})}]\!] \in E$ and $\mathcal{C}[\![System11]\!] \notin E$ since all of the infinite refusal-traces of *System*11 that begin with

$$\langle \bullet, \mathsf{Alice.OneShot.Call.null}, \bullet, \mathsf{OneShot.Carol.Call.null}, \bullet, \ldots \rangle$$

do not satisfy $\Diamond(\mathsf{Bob.OneShot.Call.null} \wedge \Diamond\, \mathsf{OneShot.Carol.Call.null})$. Hence, Alice's authority to revoke Bob's access to Carol, by using OneShot, may be captured in our framework in this way as a form of prevention.

The effect that Alice is preventing here is that inevitably Bob Calls OneShot, and when this happens OneShot then inevitably Calls Carol. This effect can be viewed as a weak form of causation, in which Bob Calling OneShot causes it to inevitably Call Carol. However, the notion of causality here is not one based on counterfactual dependence. Whatever this causation is, that is captured by the effect E defined in Equation 7.11, it should not be considered to be part of Bob's authority because, as we noted above, Bob is powerless to affect whether Carol is inevitably invoked here.

This kind of causation is actually rooted in the idea that some potential cause $\mathcal{A}$ causes some effect $\mathcal{B}$ when $\mathcal{A}$ is *sufficient* to produce $\mathcal{B}$, even if $\mathcal{A}$ is not *necessary* for $\mathcal{B}$. Identifying causation with counterfactual-dependence requires that causes be *necessary* rather than *sufficient*[5]. This example

[5]This distinction between sufficiency and necessity comes from [Par03] and [Hal04].

demonstrates that a notion of causation based on necessity rather than sufficiency is possibly the better choice for modelling authority.

7.3.5 Summary

We've seen that a wide range of causation properties can be expressed in our framework that are useful for capturing various elements of an object's authority. These various elements are relevant to a range of object-capability patterns designed to provide or control authority in various ways. In this sense, we have taken the first steps towards formally analysing the ways in which these patterns provide and control authority.

Each system that we've considered here should be thought of as a deterministic componentwise refinement of some (possibly nondeterministic) system in which the pattern being analysed is instantiated in each case. A full analysis of these patterns would require us to be able to test these non-causation properties for all deterministic componentwise refinements of each system, in accordance with Definition 7.1.1. We discuss the prospects for doing so using FDR later in Section 7.6.

A Note on Authority and Information Flow

Observe that in *System*10 in Section 7.3.3 and *System*11 in Section 7.3.4, Alice is unable to pass any information to Bob covertly through her overt actions in the system, even though Alice has influence over Bob's authority and, hence, some kind of authority over Bob. These examples indicate, then, that one object can have authority over another without necessarily being able to pass information to it.

Recall that in *System*5 in Section 7.1.3, we saw that one object could pass information to another without having certain kinds of authority over it. Taken together, these examples indicate that authority and information flow are somewhat independent of each other, despite both having natural formulations in terms of counterfactual causation. (Noninterference properties, for instance, assert that the presence of High activity doesn't affect Low, and naturally imply a counterfactual comparison of Low between the scenarios in which High does and does not act.)

7.4 Safety and Liveness Effects

We now show that, in our framework, two different primitive kinds of effect can be distinguished, namely the *safety* and *liveness* effects respectively. As we will see later, this distinction is integral to our formalisation of Miller's notions of defensive correctness and consistency, and is also useful for understanding which kinds of non-causation and non-prevention properties can be more easily tested using FDR. We make this distinction with the aid of Clarkson and Schneider's *hyperproperties* framework [CS08, CS10], and

adapt one of their results to show that every effect can be expressed as the intersection of a safety effect and a liveness effect.

Observe that what we call an effect above might more generally be called simply a *refusal-traces property*. An effect E is identified with the refusal-traces property *Prop* that holds for a process P iff the effect is present in E, so that $\forall P \in \mathbf{CSP} \bullet Prop(P) \Leftrightarrow \mathcal{C}[\![P]\!] \in E$.

It turns out that each effect E corresponds to a unique element from the class of refusal-traces *hyperproperties* [CS08, CS10], and vice-versa. Clarkson and Schneider recently introduced the notion of a hyperproperty to capture a broad class of security properties, across a range of formalisms, that includes traditional safety and liveness properties, as well as information flow properties and various other kinds of property. The concept of a hyperproperty is, therefore, necessarily very abstract; its abstract definition is made meaningful by instantiating it within a particular semantic framework. Here, we present the abstract definitions and then show how they may be instantiated in the context of the refusal-traces model. When instantiated this way, each hyperproperty corresponds to a unique effect and vice-versa. We then adopt Clarkson and Schneider's distinction between safety and liveness hyperproperties to distinguish safety and liveness effects.

In the abstract definition of a hyperproperty, a system is represented by a non-empty set of infinite sequences of *states* σ. We call each of these sequences an *execution* of the system. Any completed finite execution of a system that ends in some state σ is represented by the infinite sequence obtained from the finite one by infinitely stuttering the final state σ. The set of all such infinite-length executions is denoted Ψ_{inf}. The set of all partial (incomplete) executions, which are finite sequences of states σ, is denoted Ψ_{fin}.

Certain constraints may be imposed on the representation of a valid system. Hence, the set *Rep* denotes the set containing all valid system representations. Each member of *Rep* is therefore a non-empty set of infinite executions that represents a valid system.

A *hyperproperty HProp* for system representation *Rep* is then a set of systems from *Rep*, namely just those that satisfy the condition that *HProp* represents. The set of all hyperproperties for system representation *Rep* is then $\mathbf{P}\, Rep$. The hyperproperty true is of course *Rep* and the hyperproperty false is $\{\}$.

Each effect E is trivially mapped onto a corresponding hyperproperty by mapping each completed refusal-trace $s \in \mathcal{C}[\![P]\!] \in E$ onto a corresponding infinite execution. Each state σ of an infinite execution is either:

- a pair (X, a) where $a \in \Sigma$ and $(X \subseteq \Sigma \wedge a \notin X) \vee X = \bullet$, or
- the symbol dl (for "deadlock").

Then each completed refusal trace is mapped onto an infinite sequence of states σ as follows.

- A deadlocked refusal trace $\langle X_1, a_1, X_2, a_2, \ldots, X_n, a_n, \Sigma\rangle$ corresponds to the infinite execution $\langle (X_1, a_1), (X_2, a_2), \ldots, (X_n, a_n), \mathsf{dl}, \mathsf{dl}, \ldots\rangle$, in which the final dl state is stuttered infinitely, and vice-versa.
- An infinite refusal-trace $\langle X_1, a_1, X_2, a_2, \ldots, X_n, a_n, \ldots\rangle$ corresponds to the infinite execution $\langle (X_1, a_1), (X_2, a_2), \ldots, (X_n, a_n), \ldots\rangle$, and vice-versa.

The set *Rep* of valid system representations is simply those sets of infinite executions that correspond to $\mathcal{C}[\![P]\!]$ for some process $P \in \mathbf{CSP}$. Hence, every effect $E \in \mathit{Effect}$ corresponds to a hyperproperty for this system representation *Rep* and vice-versa. We call such a hyperproperty a *refusal-traces hyperproperty*. Since each refusal-traces hyperproperty corresponds to an effect and vice-versa, we will therefore use the words "effect" and "hyperproperty" interchangeably from now on, noting that by "hyperproperty" we mean a refusal-traces hyperproperty.

Clarkson and Schneider distinguish two kinds of hyperproperty, namely the *safety* and *liveness* ones. The definition of each kind captures the natural intuitions about safety and liveness properties to which we have appealed in Chapters 3 and 6, for example. Recall that a safety property asserts that something (bad) never occurs, while a liveness property asserts that something (good) must occur [Lam77, AS85]. Liveness properties have the intuitive characteristic that any incomplete observation of a system can always be possibly extended so as to satisfy any liveness property [VVK05, AFK88, AL91]. Each of these intuitions is captured in the context of hyperproperties as follows.

Clarkson and Schneider define the set *Obs* of all *observations* that could be made of any system in a finite amount of time, while allowing the observer to restart the system at any point while it is being observed to observe multiple finite executions of the system[6]. Each observation is therefore a finite set of finite executions from Ψ_{fin}.

$$\mathit{Obs} = \mathbf{P}^{\mathit{fin}}\, \Psi_{\mathit{fin}},$$

where $\mathbf{P}^{\mathit{fin}}\, X$ denotes the set of all finite subsets of X.

Given an observation $M \in \mathit{Obs}$ and a set T of finite or infinite executions, we say that M is a prefix of T, written $M \leq T$, when the observation M can be made of T, *i.e.*

$$M \leq T \Leftrightarrow (\forall\, s \in M \bullet \exists\, t \in T \bullet s \leq t).$$

Under this definition, T can of course contain new executions not in M as one would expect.

[6] Equivalently, allowing the observer to run multiple copies of the system in parallel.

The set $Obs(Rep)$ contains all observations that could be made of any valid system. Hence,

$$Obs(Rep) = \{M \mid M \in Obs \land \exists\, Sys \in Rep \bullet M \leq Sys\}.$$

Then a *safety hyperproperty* is one that asserts that something bad can never happen. This bad thing is necessarily an observation $M \in Obs(Rep)$. Once this bad thing has occurred, the property is violated forever; no further action by the system can undo the violation. This leads naturally to the following definition from [CS10], which parallels the standard definition for safety properties [AS85].

Definition 7.4.1 (Safety Hyperproperty for system representation Rep)**.** A hyperproperty $HProp$ is a *safety hyperproperty for system representation* Rep iff

$$\forall\, Sys \in Rep \bullet Sys \notin HProp \Rightarrow \left(\begin{array}{l} \exists\, M \in Obs(Rep) \bullet M \leq Sys \land \\ \qquad (\forall\, Sys' \in Rep \bullet M \leq Sys' \Rightarrow Sys' \notin HProp) \end{array} \right).$$

Observe that each finite (*i.e.* partial or deadlocked) refusal-trace can be trivially mapped onto a finite sequence of states σ in which dl is always the last element if it is present. Each member of the set $Obs(Rep)$ of observations of valid systems then simply corresponds to a set $M \subseteq \mathcal{R}[\![P]\!]$ of partial and deadlocked refusal traces that can be exhibited by some process $P \in \mathbf{CSP}$. Then some observation $M \in Obs(Rep)$ is a prefix of a system $Sys \in Rep$ iff the finite set M' of finite refusal-traces that corresponds to M can be exhibited by the system $Sys' \in \mathbf{CSP}$ for which $\mathcal{C}[\![Sys']\!]$ corresponds to Sys, *i.e.*

$$M \leq Sys \Leftrightarrow M' \subseteq \mathcal{R}[\![Sys']\!].$$

Hence, some effect $E \in Effect$ corresponds to a safety hyperproperty iff

$$\forall\, Sys \in \mathbf{CSP} \bullet \mathcal{C}[\![Sys]\!] \notin E \Rightarrow \left(\begin{array}{l} \exists\, M \bullet |M| \in \mathbb{N} \land M \subseteq \mathcal{R}[\![Sys]\!] \land \\ \qquad (\forall\, Sys' \in \mathbf{CSP} \bullet M \subseteq \mathcal{R}[\![Sys']\!] \Rightarrow \mathcal{C}[\![Sys']\!] \notin E) \end{array} \right). \tag{7.12}$$

We therefore call such an effect a *safety effect*.

All of the safety properties considered so far in this thesis can be expressed as safety effects. This can be seen by simply considering the bad thing that each of these properties asserts cannot arise and observing that it can be represented by a finite set M of finite refusal-traces.

The most extreme example, which demonstrates the power of safety hyperproperties for expressing interesting security properties, would be the refinement-closed noninterference properties (see Definition 5.3.12) from Chapter 5, each of which can be expressed as a safety hyperproperty. The bad thing in this case is necessarily two refusal-traces s_1 and s_2, related by

(a suitable adaptation of) *Pred* (from traces to refusal-traces) such that e follows one but is stably refused after the other. Any weakened refinement-closed noninterference property (see Definition 5.3.18) can also be expressed as a safety hyperproperty, although of the composition *WSys* (see Snippet 5.2) that allows us to observe both system-level and individual component refusals.

A *liveness hyperproperty* is one such that any incomplete observation can always be extended so as to satisfy that property. This is captured by the following definition [CS10], which parallels the standard definition for liveness [AS85].

Definition 7.4.2 (Liveness Hyperproperty for system representation *Rep*)**.** A hyperproperty *HProp* is a *liveness hyperproperty for system representation Rep* iff

$$\forall\, M \in \mathit{Obs}(\mathit{Rep}) \bullet \exists\, \mathit{Sys}' \in \mathit{Rep} \bullet M \leq \mathit{Sys}' \land \mathit{Sys}' \in \mathit{HProp}.$$

Note that this definition allows M to contain observations that end in deadlock. However, observe that any such $M \in \mathit{Obs}(\mathit{Rep})$ cannot be extended so as to satisfy the liveness property $\Box \Diamond e$. Hence, in order to ensure that $\Box \Diamond e$ is a liveness refusal-traces hyperproperty, we need to restrict our attention to those M in the above definition that contain only executions that correspond to partial refusal-traces from PRT.

Any effect $E \in \mathit{Effect}$ corresponds to a liveness hyperproperty under this restriction iff

$$\begin{aligned} &\forall\, M \bullet \forall\, \mathit{Sys} \in \mathbf{CSP} \bullet |M| \in \mathbb{N} \land M \subseteq \mathcal{R}[\![\mathit{Sys}]\!] \cap \mathit{PRT} \Rightarrow \\ &\quad \exists\, \mathit{Sys}' \in \mathbf{CSP} \bullet M \subseteq \mathcal{R}[\![\mathit{Sys}']\!] \land \mathcal{C}[\![\mathit{Sys}']\!] \in E. \end{aligned} \tag{7.13}$$

Naturally, we call such an effect a *liveness effect*.

As one might expect, each of the effects $E_{\Diamond e}$, $E_{\mathbf{EF}\, e}$ and $E_{\mathbf{EG}\, \mathbf{EF}\, e}$ from Equations 7.4, 7.5 and 7.9 respectively are liveness effects. Also, all of the liveness properties considered so far in this thesis can be expressed as liveness effects. This includes liveness properties like $\mathit{SEF} \Rightarrow \Diamond e$ and more complicated ones like that in Equation 6.5. Finally, one can observe that the property of deadlock-freedom is both a safety and a liveness effect.

Clarkson and Schneider show that every hyperproperty for system representation *Rep* can be expressed as the intersection of a safety and liveness hyperproperty for *Rep* respectively. This parallels the well-known analogue of this result for the standard definitions of safety and liveness [AS85]. Theorem A.0.7, which appears in Appendix A, is a straightforward adaptation of their result, which cannot be applied directly because of our slightly specialised definition of a liveness effect. It states that for every effect $E \in \mathit{Effect}$ there exists a safety effect E_S and a liveness effect E_L where $E = E_S \cap E_L$.

7.5 Defensive Correctness and Consistency

In this section, we show how Miller's notions of *defensive correctness* [Mil06, Section 5.5] and *defensive consistency* [Mil06, Section 5.6] can be formalised in our framework. These are intuitive security properties that have often been applied informally in the past when informal analyses of object-capability patterns and systems have been carried out, and are widely regarded amongst those who build object-capability systems as being very important properties for patterns to uphold (see *e.g.* [MWC10]). We present here the first attempt to capture these properties formally. We begin by explaining and motivating each of them.

Suppose we have some pattern implemented by an object, such as the Guard object that implements the Trademarks pattern in the systems depicted in Figures 3.1 and 6.1 (the second of which we repeat here for convenience as Figure 7.7). Suppose Guard has a number of clients that want

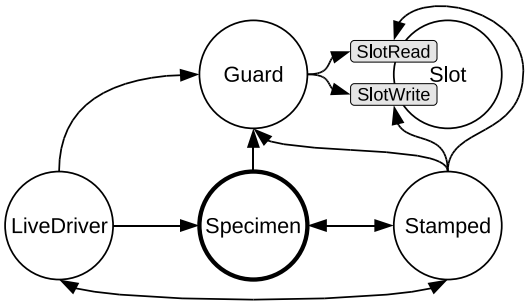

Figure 7.7: Defining defensively correct Trademarks.

to use its service. If Guard is defensively correct then none of Guard's clients needs to rely on any of Guard's other clients in order for Guard to provide it with correct service. Each of Guard's clients that wants correct service will still have to interact with Guard in a way that allows Guard to be able to provide it with correct service, however. An object o is *defensively correct* when [Mil06, MWC10],

> under the assumption that every object that o relies upon for its correctness is correct, o provides correct service to each of its clients c when c interacts with o in such a way as to allow o to provide it with correct service, irrespective of the behaviour of o's other clients c'.

The assumption that every object that o relies upon for its correctness is correct is required because if o provides incorrect service to c because of a bug in some service that o relies upon, then this bug is rightly considered a fault of the relied upon service and not of o [Mil06, Section 5.5]. Similarly, the assumption that c interacts with o in such a way that o can provide it correct service is necessary because o's inability to provide c correct service

because c doesn't respect o's preconditions should not be considered a fault of o. This is why, for instance, when analysing the liveness of the Trademarks implementation in Section 6.2, we tested that Guard would eventually Return to only those clients (namely just LiveDriver) that could be guaranteed to accept the Return message.

Defensive correctness is a very useful property because it absolves each client of o from having to rely on the correctness of o's other clients. When o is defensively correct, each client of o need only rely on those things that o (transitively) needs to rely on in order to expect correct service from o. This greatly reduces the degree to which each client of o is vulnerable to the misbehaviour of other objects in the system.

Widely used objects should therefore ideally be verified to be defensively correct in order to ensure that their presence doesn't inadvertently make large collections of objects vulnerable to each other's misbehaviour.

Defensive consistency is a weaker form of defensive correctness. Miller notes that correctness is a combination of safety and liveness. Defensive consistency incorporates safety only [Mil06, Section 5.6]. Therefore, an object o is *defensively consistent* when,

> under the assumption that every object that o relies upon for its correctness is correct, whatever service that o provides to each of its clients c is never incorrect when c interacts with o in such a way as to allow o to avoid providing it incorrect service, *although o may be prevented from giving c any service*, irrespective of the behaviour of o's other clients c'.

The clients of an object that is defensively consistent may be able to cause it to not provide service to other clients, but they cannot cause the service it does provide to be incorrect. A defensively consistent object may, therefore, be vulnerable to denial-of-service but should still be incorruptible by its clients [Mil06, Section 5.6].

Observe that an object that is defensively correct or consistent limits the authority of its clients to interfere with each other, and that the clients of a defensively correct (respectively consistent) object o cannot cause o to give incorrect or no service (respectively give service that is not correct) to a client c when c behaves so as to allow o to be able to render it such service and the objects on which o relies are correct, *i.e.* when o would otherwise give such service to c. Hence, defensive correctness and defensive consistency may each be framed as a non-prevention property that asserts that for each client c of o, the other clients don't have the authority to prevent the effect involving:

- o always giving service that is correct to c, for defensive correctness, or
- o never giving incorrect service to c, for defensive consistency,

when that effect would have otherwise been present. Concurring with Miller's observation [Mil06, Section 5.6], the effect in the case of defensive correctness naturally incorporates both a safety component (o never gives incorrect service to c) and a liveness component (o always gives service to c), while in the case of defensive consistency the effect incorporates only the safety component.

Suppose we can frame each kind of effect and let $E_{Corr}(o, c)$ be the effect in the case of defensive correctness and $E_{Con}(o, c)$ be the effect in the case of defensive consistency. $E_{Corr}(o, c)$ asserts that o never gives incorrect service to c and always gives service to c, while $E_{Con}(o, c)$ asserts that o never gives incorrect service to c. $E_{Con}(o, c)$ is naturally a safety effect, while $E_{Corr}(o, c)$ is the intersection (*i.e.* conjunction) of $E_{Con}(o, c)$ and a liveness effect that asserts that o always gives service to c.

Then consider some system $System$ containing (at least) o, c and some other clients c' of o, and let C be the set of all clients of o including c. Because defensive correctness and consistency are both properties involving authority and because authority is calculated over just those deterministic componentwise refinements of $System$, we have that o is defensively correct or consistent in $System$ when it is defensively correct or consistent respectively in each $System_D \in DCRef(System)$ in accordance with Definition 7.1.1.

So consider some $System_D \in DCRef(System)$ and consider the process $System_D^c = System_D \,|_{\bigcup_{c' \in C - \{c\}} \alpha(c')}$ in which the activity of all clients of o other than c has been blocked. Then $\mathcal{C}[\![System_D^c]\!] \in E_{Corr}(o, c)$ precisely when the objects on which o relies to provide c correct service are behaving correctly and c is behaving in such a way as to allow o to provide it correct service, since if either was not true o could not provide c correct service and so we would have $\mathcal{C}[\![System_D^c]\!] \notin E_{Corr}(o, c)$. o fails to provide c correct service in $System_D$ of course when $\mathcal{C}[\![System_D]\!] \notin E_{Corr}(o, c)$. It follows o is not defensively correct in some $System_D$, then, precisely if $\exists\, c \in C \bullet \mathcal{C}[\![System_D^c]\!] \in E_{Corr}(o, c) \wedge \mathcal{C}[\![System_D]\!] \notin E_{Corr}(o, c)$. Therefore, o is defensively correct in $System$ precisely when for all for all $c \in C$ and $System_D \in DCRef(System)$, we have that

$$\mathcal{C}[\![System_D^c]\!] \in E_{Corr}(o, c) \Rightarrow \mathcal{C}[\![System_D]\!] \in E_{Corr}(o, c). \tag{7.14}$$

The same is true for defensive consistency, of course, when we replace $E_{Corr}(o, c)$ by $E_{Con}(o, c)$. So o is defensively consistent in $System$ precisely when for all $c \in C$ and all $System_D \in DCRef(System)$, we have that

$$\mathcal{C}[\![System_D^c]\!] \in E_{Con}(o, c) \Rightarrow \mathcal{C}[\![System_D]\!] \in E_{Con}(o, c). \tag{7.15}$$

Note that because the counterfactual comparison is performed in each case against $System_D^c$, in which the events of all clients other than c have been blocked, these definitions allow one to detect when the combined efforts of two or more clients affect c's interactions with o. This means these definitions therefore cover the case in which two or more clients must conspire in order to affect c.

Hence, it appears as if defensive correctness and consistency can each be expressed as the conjunction (over C) of multiple applications of Definition 7.1.1 in which *Prop* is replaced by Equations 7.14 and 7.15 respectively.

Of course, the conjunction over C is not necessary when *System* is symmetric in C, since in this case we can test the property for one such $c \in C$ and conclude by symmetry that it must hold for the others.

We see that defensive consistency thus asserts the non-prevention of some safety effect $E_{Con}(o, c)$ (that asserts that o never gives incorrect service to c), while defensive correctness asserts the non-prevention of some effect $E_{Corr}(o, c)$ that is the conjunction of $E_{Con}(o, c)$ and a liveness effect (that asserts that o always gives service to c). This formalises Miller's observation in our context that defensive correctness involves both safety and liveness while defensive consistency involves only safety.

7.5.1 Defining Defensively Correct Trademarks

We demonstrate these ideas in the context of the Trademarks analyses performed earlier in Sections 3.1 (safety) and 6.2 (liveness). In Section 3.1, we tested a safety property of the Trademarks pattern. From this we can easily describe a safety effect $E_{Con}(\mathsf{Guard}, c)$ that asserts that the Guard object from the Trademarks pattern, instantiated as *e.g.* in Figure 7.7 above, never gives incorrect service to client c. $E_{Con}(\mathsf{Guard}, c)$ is the safety effect in which $\mathsf{Guard}'s$ behaviour towards c is trace-equivalent to the process $SafeGuard_c(\mathsf{Guard}, \{\mathsf{Stamped}\})$ where $SafeGuard_c(me, sObjs)$ is the behaviour of a guard me, whose set of stamped objects is $sObjs$, that is always safe for client c. It is defined as follows.

$$
\begin{array}{l}
SafeGuard_c(me, sObjs) = \\
\quad c!me!\mathsf{Call}?specimen : Capability \rightarrow \\
\quad \textbf{if } specimen \in sObjs \textbf{ then} \\
\qquad \left(\begin{array}{l} me!c!\mathsf{Return}!me \rightarrow SafeGuard_c(me, sObjs) \sqcap \\ me!c!\mathsf{Return}!\mathsf{null} \rightarrow SafeGuard_c(me, sObjs) \end{array}\right) \\
\quad \textbf{else } \; me!c!\mathsf{Return}!\mathsf{null} \rightarrow SafeGuard_c(me, sObjs)
\end{array}
$$

The safety effect $E_{Con}(\mathsf{Guard}, c)$ is then defined as

$$
E_{Con}(\mathsf{Guard}, c) = \left\{ \mathcal{C}[\![P]\!] \;\middle|\; \begin{array}{l} P \in \mathbf{CSP} \wedge \\ \forall\, s \in \mathcal{R}[\![P]\!] \bullet tr(s) \upharpoonright \alpha(\mathsf{Guard}) \cap \alpha(c) \in \\ \quad traces(SafeGuard_c(\mathsf{Guard}, \{\mathsf{Stamped}\})) \end{array} \right\}.
$$

So Guard is defensively consistent in some system *System* when for all of Guard's clients c and for all $System_D \in DCRef(System)$, Equation 7.15 holds.

To assert defensive correctness, we also define the liveness effect $E_{c,L}$ that asserts that c is always given service by Guard. $E_{c,L}$ is defined by adapting the liveness property (Equation 6.5) tested for the Trademarks pattern in

Section 6.2. Client c is always given service, under the assumption of strong event fairness, when the system satisfies the liveness property ϕ defined as follows.

$$\phi = SEF \Rightarrow \bigwedge_{specimen \in Capability} \left(\begin{array}{l} \Box \Diamond c.\mathsf{Guard.Call}.specimen \wedge \\ \left(\begin{array}{l} \Box\, c.\mathsf{Guard.Call}.specimen \Rightarrow \\ \quad \Diamond(\mathsf{Guard}.c.\mathsf{Return.null} \vee \mathsf{Guard}.c.\mathsf{Return.Guard}) \end{array} \right) \end{array} \right).$$

This property asserts not only that whenever c Calls Guard, Guard must eventually Return, but also that c can always eventually Call Guard and so can always get service. The liveness effect $E_{c,L}$ is then defined as

$$E_{c,L} = \{\mathcal{C}[\![P]\!] \mid P \in \mathbf{CSP} \wedge P \models \phi\}.$$

Guard gives correct service to client c in some $System_D$ then when $\mathcal{C}[\![System_D]\!] \in E_{Con}(\mathsf{Guard}, c) \cap E_{c,L}$. Hence let $E_{Corr}(\mathsf{Guard}, c) = E_{Con}(\mathsf{Guard}, c) \cap E_{c,L}$. Guard is defensively correct in some system $System$ then when for all clients c of Guard and for all $System_D \in DCRef(System)$, Equation 7.14 holds.

7.5.2 Discussion

We have seen that the informal notions of defensive correctness and consistency can be formalised in our framework and that Miller's intuitions regarding these properties and safety and liveness are naturally captured in terms of safety and liveness effects.

We expect that these properties should be very useful for analysing object-capability patterns. Applying them properly would require one to be able to judge non-prevention for all deterministic componentwise refinements of a system. We discuss the prospects for doing so with FDR in the following section.

It should be noted, however, that the assumptions encoded in the definitions of defensive correctness and consistency mean that they cannot be used in place of testing ordinary safety and liveness properties for object-capability patterns. For instance, Guard is defensively consistent when the service it gives (if any) to all clients c is correct despite arbitrary activity from other clients *only in those circumstances in which each of the objects on which* Guard *relies is correct*. We saw in Section 3.1 that Guard relies on Stamped not to divulge its capability to its slot's write-facet in order to remain safe. Hence, testing that Guard is defensively consistent would not detect that Guard is unsafe when Stamped divulges its slot's write-facet capability. Hence, ordinary safety and liveness properties are still required in order to check that the objects on which a pattern relies are correct.

Testing for these properties considers whether o gives expected service to some client c only in those cases in which c behaves in such a way as to allow

o to give it such service. Hence, unlike when we analysed the Trademarks pattern for liveness in Section 6.2, testing for these properties does not require one to manually construct implementations of clients, like LiveDriver, whose behaviour doesn't prevent the property being tested from being violated. The assumption that any client that expects some kind of service will behave in a way so as to allow it to be provided with that service, is automatically encoded in the definitions of defensive defensive correctness and consistency.

The same applies of course to the objects on which o relies, as noted above. This means that when testing whether an object is defensively consistent or correct, one can model all of its clients and all of the objects on which it relies as instances of the most general process, *e.g.* as instances of $Untrusted_{OS}$ (see Snippet 2.1). One expects that all such objects could be aggregated into a single $Untrusted_{OS}$ object. This reflects the intuitive expectation that the judgement about whether an object is defensively correct or consistent respectively can be made independently of its clients and the objects on which it relies, *i.e.* independently of its environment [MWC10].

7.6 Testing Non-Causation and Non-Prevention

We now consider to what degree FDR can be applied to allow one to check Definition 7.1.1 when *Prop* is replaced by some non-causation or non-prevention property, *i.e.* a property of the form of Equations 7.7 or 7.8 for an arbitrary effect E. We need to be able to test this, for instance, in order to be able to check automatically that a pattern, like the NDA, properly confines authority or that an object is defensively consistent or correct.

7.6.1 Deterministic Systems

We begin by considering the simpler problem of testing for non-causation (or similarly non-prevention) of some effect E for deterministic systems, which naturally have no proper deterministic componentwise refinements. Let E be some effect and *System* be some deterministic system so that $DCRef(System) = \{System\}$. Then, by Definition 7.1.1 and Equation 7.7, testing for non-causation of E by an arbitrary object whose alphabet is A in *System* is equivalent to testing $\mathcal{C}[\![System_D]\!] \in E \Rightarrow \mathcal{C}[\![System_D \,|_A]\!] \in E$ for all $System_D \in DCRef(System)$, which just amounts to testing that

$$\mathcal{C}[\![System]\!] \in E \Rightarrow \mathcal{C}[\![System \,|_A]\!] \in E. \tag{7.16}$$

For prevention, recall that by Equation 7.8, the direction of this implication is simply reversed.

Suppose, for an arbitrary process P, we can test whether $\mathcal{C}[\![P]\!] \in E$ using a refinement check in FDR. This refinement check will naturally be of the form $F(P) \sqsubseteq_{\mathcal{M}} G(P)$ for CSP contexts $F(\_)$ and $G(\_)$ and some

CSP model $\mathcal{M}$ that FDR might support. Then we can easily test for non-causation or non-prevention of the effect E by performing at most two of these refinement checks: one with *System* in place of P, the other with $System\,|_A$ in place of P.

This check will be easiest for FDR to carry out when the CSP context $F(\_)$ used on the left-hand side of this refinement is just some constant process *Spec*, *i.e.* is a context that makes no use of its argument P whatsoever [Ros05]. It turns out that when E is some safety effect, deciding whether E is present in some arbitrary process P, *i.e.* whether $\mathcal{C}[\![P]\!] \in E$, is equivalent to testing such a refinement in the refusal-traces model

$$Spec \sqsubseteq_R G(P).$$

Theorem 7.6.1. Let E be a safety effect. Then there exists a fixed specification process *Spec* and a CSP context $G(\_)$ that makes no use of divergence-creating hiding such that for all divergence-free processes $P \in \mathbf{CSP}$

$$\mathcal{C}[\![P]\!] \in E \Leftrightarrow Spec \sqsubseteq_R G(P).$$

Proof. We extend Roscoe's proof from [Ros05, Theorem 3.2].

From Equation 7.12, E may be characterised by a set S of finite sets M of finite refusal-traces such that each set $M \in S$ corresponds to some process $P \in \mathbf{CSP}$ for which $\mathcal{C}[\![P]\!] \notin E$, $M \subseteq \mathcal{R}[\![P]\!]$ and $\forall Q \in \mathbf{CSP} \bullet M \subseteq \mathcal{R}[\![Q]\!] \Rightarrow \mathcal{C}[\![Q]\!] \notin E$. Given such an S and an arbitrary process P, we have that

$$\mathcal{C}[\![P]\!] \in E \Leftrightarrow \forall M \in S \bullet M \not\subseteq \mathcal{R}[\![P]\!].$$

We will build a refinement check from S that asserts exactly this.

Consider some M of S and some process P. We first build a refinement check that asserts that $M \not\subseteq \mathcal{R}[\![P]\!]$. We then simply take the conjunction of this test across all $M \in S$, which can itself be expressed as a refinement check as we'll see.

Let s be a member of M, *i.e.* s is a finite refusal-trace that is either a partial or deadlocked refusal-trace. Consider the process $T(s)$ defined as follows.

$$\begin{aligned} T(\langle \bullet, a \rangle \mathbin{^\frown} t) &= \mathsf{pong2} \rightarrow a \rightarrow T(t), \\ T(\langle X, a \rangle \mathbin{^\frown} t) &= (?x : X \rightarrow \mathsf{ping} \rightarrow STOP) \,\Box\, (\mathsf{pong} \rightarrow a \rightarrow T(t)), \\ T(\langle \Sigma \rangle) &= \mathsf{dotest} \rightarrow ?x : \Sigma \rightarrow \mathsf{ping} \rightarrow STOP, \\ T(\langle\rangle) &= \mathsf{dotest} \rightarrow STOP. \end{aligned}$$

We define the process $Test(s, P)$ which places P and $T(s)$ in parallel. $T(s)$ acts as a *testing* process and is designed so that $Test(s, P)$ exhibits certain behaviours iff P can perform s, *i.e.* iff $s \in \mathcal{R}[\![P]\!]$. Letting $\Sigma^- = \Sigma - \{\mathsf{ping}, \mathsf{pong}, \mathsf{pong2}, \mathsf{dotest}\}$, we have

$$Test(s, P) = (T(s) \underset{\Sigma^-}{\parallel} P) \setminus \Sigma^-.$$

Consider $Test(\langle X, a\rangle \mathbin{^\frown} t, P)$. We see that it can stably refuse $\{\mathsf{ping}\}$ initially iff P can stably refuse X initially. When P stably refuses X initially, $Test(\langle X, a\rangle \mathbin{^\frown} t, P)$ can make progress after performing pong only if P can perform a. Hence, we have that $\langle X, a\rangle \in \mathcal{R}[\![P]\!]$ iff $\langle \{\mathsf{ping}\}, \mathsf{pong}\rangle \mathbin{^\frown} u \in \mathcal{R}[\![Test(\langle X, a\rangle \mathbin{^\frown} t, P)]\!]$ where $u \notin \{\langle\Sigma\rangle, \langle\rangle\}$. For $Test(\langle \bullet, a\rangle \mathbin{^\frown} t, P)$, we see that $\langle \bullet, a\rangle \in \mathcal{R}[\![P]\!]$ iff $\langle \bullet, \mathsf{pong2}\rangle \mathbin{^\frown} u \in \mathcal{R}[\![Test(\langle \bullet, a\rangle \mathbin{^\frown} t, P)]\!]$ where $u \notin \{\langle\Sigma\rangle, \langle\rangle\}$.

Let $U = \{\langle X_1, a_1, \ldots, X_n, a_n\rangle \mathbin{^\frown} \langle \Sigma - \{\mathsf{dotest}\}, \mathsf{dotest}, \{\mathsf{ping}\}\rangle \mid n \in \mathbb{N} \wedge \forall\, i \in \{1, \ldots, n\} \bullet (X_i = \{\mathsf{ping}\} \wedge a_i = \mathsf{pong}) \vee (X_i = \bullet \wedge a_i = \mathsf{pong2})\}$. Then we have that

$$U \cap \mathcal{R}[\![Test(s, P)]\!] \neq \{\} \Leftrightarrow s \in \mathcal{R}[\![P]\!].$$

We define a specification *Spec* that can exhibit none of the refusal-traces from U. One way that *Spec* can be written is as follows.

$$\begin{aligned} &Spec = \\ &\quad (\mathsf{pong} \to CHAOS_\Sigma \triangleright \mathsf{ping} \to CHAOS_\Sigma) \sqcap \\ &\quad (\mathsf{dotest} \to \mathsf{ping} \to STOP) \sqcap \\ &\quad (\mathsf{pong} \to Spec) \sqcap (\mathsf{pong2} \to Spec) \sqcap STOP \end{aligned}$$

Spec allows arbitrary behaviour following any occurrence of pong from a state in which ping is not refused. *Spec* also allows arbitrary behaviour following any ping occurrence. This is because, in either case, once this has happened, the behaviour being exhibited cannot be from U. *Spec* allows ping to be refused always except after a dotest event. Hence, we then have that $Spec \sqsubseteq_R Test(s, P) \Leftrightarrow s \notin \mathcal{R}[\![P]\!]$.

We need to extend this test to assert that $M \not\subseteq R[P]$. We may do so by defining the process $Combine(M, P)$ as

$$Combine(M, P) = \|_{\{\mathsf{dotest}\}}\ _{s \in M}\ Test(s, P).$$

For each $s \in M$, $Combine(M, P)$ runs a copy of $Test(s, P)$ in parallel to all others forcing them all to synchronise on the occurrence of the dotest event. This is sound because M is finite. The net effect is that $Combine(M, P)$ can exhibit a refusal-trace from U iff each $Test(s, P)$ can. Hence, we have

$$Spec \sqsubseteq_R Combine(M, P) \Leftrightarrow M \not\subseteq \mathcal{R}[\![P]\!].$$

Finally, to assert that $\forall\, M \in S \bullet M \not\subseteq \mathcal{R}[\![P]\!]$ (*i.e.* to express $\mathcal{C}[\![P]\!] \in E$ for some process P), consider the process $\bigsqcap_{M \in S} Combine(M, P)$ that can exhibit all of the refusal-traces that can be exhibited by each $Combine(M, P)$ for all $M \in S$. Then we have that

$$Spec \sqsubseteq_R \bigsqcap_{M \in S} Combine(M, P) \Leftrightarrow \forall\, M \in S \bullet M \not\subseteq \mathcal{R}[\![P]\!].$$

Let $G(\_)$ be the CSP context that is defined as $G(P) = \bigsqcap_{M \in S} Combine(M, P)$. We see that

$$Spec \sqsubseteq_R G(P) \Leftrightarrow \mathcal{C}[\![P]\!] \in E.$$

Note finally that $G(\_)$ uses no divergence-creating hiding, as required, since the hiding used in $Test(s, P)$ cannot cause divergence. □

By Theorem 7.6.1, one can judge whether $\mathcal{C}[\![P]\!] \in E$ for some safety effect E by testing a refinement of the form $Spec \sqsubseteq_R G(P)$ for some CSP context $G(\_)$. In many cases, the context $G(\_)$ is just the identity function and the specification *Spec* is constructed so that it never exhibits the bad thing that the safety effect in question asserts cannot arise. In other cases, for instance, the context $G(\_)$ may run multiple copies of its argument P when the bad thing (characterised by the set M from Equation 7.12) is a set of multiple behaviours. A notable concrete example of a safety effect of this kind is any refinement-closed noninterference property. Recall that it is exactly this technique that is used to test these properties in FDR (see Section 5.3).

We illustrate the simplest case. Suppose we define the safety effect $E_{\mathsf{not}\ e}$ that asserts that the event e can never occur.

$$E_{\mathsf{not}\ e} = \{\mathcal{C}[\![P]\!] \mid P \in \mathbf{CSP} \wedge \forall\, s \in traces(s) \bullet s \upharpoonright \{e\} = \langle\rangle\}. \tag{7.17}$$

Then testing that $P \in \mathcal{C}[\![P]\!]$ is equivalent to testing that

$$CHAOS_{\Sigma-\{e\}} \sqsubseteq_R P.$$

From Equation 7.16, to test non-causation of this effect in some deterministic system *System* by some object with alphabet A, we first test if $CHAOS_{\Sigma-\{e\}} \sqsubseteq_R System$. If this refinement doesn't hold, we can conclude that this effect is not caused in *System*. If this refinement holds, however, we then test whether $CHAOS_{\Sigma-\{e\}} \sqsubseteq_R System\,|_A$. If this check doesn't hold, then the object with alphabet A causes the effect $E_{\mathsf{not}\ e}$ in *System*; otherwise, it doesn't. So we can test non-causation of this safety effect with at most two refinement checks.

We can, of course, do similarly to test non-prevention of this effect (by performing the tests in the reverse order). Observe, however, that non-prevention of the effect $E_{\mathsf{not}\ e}$ is equivalent to non-causation of the effect $\overline{E_{\mathsf{not}\ e}}$ (see Equation 7.6). Note also that $\overline{E_{\mathsf{not}\ e}} = E_{\mathbf{EF}\ e}$ from Equation 7.5. Hence, testing non-prevention of $E_{\mathsf{not}\ e}$ is equivalent to testing non-causation of $E_{\mathbf{EF}\ e}$. So, in general, we can also test non-causation or non-prevention of those liveness effects whose complement is a safety effect[7] that can be expressed by a finite-state refinement check.

Note that the refinement check $Spec \sqsubseteq_R G(P)$ may not always be finite-state, meaning that it may not always be possible for it to be checked automatically by FDR. However, to our knowledge, the vast majority of useful safety hyperproperties can be expressed as finite-state refinement checks. This includes all of the safety hyperproperties (such as the safety properties

[7] The complement of a safety property is called an *observable* property [Abr91, CS10].

from Chapter 3 and the refinement-closed information flow properties from Chapter 5) considered in this thesis.

We conclude, therefore, that one can usually check non-causation or non-prevention of some safety effect, or its complement, for a deterministic system *System* by performing at most two refinement checks in FDR.

Testing non-causation/-prevention for deterministic systems of non-safety effects, whose complement is not a safety effect, is more difficult. This is because such effects necessarily involve observing infinite behaviours. Some non-safety effects, like the effect $E_{\Diamond e}$ from Equation 7.4, can be expressed as refinement checks (see the refinement check given in Equation 6.1), by mapping the infinite behaviours that they involve onto divergences using hiding. Others, like the effect $E = \{\mathcal{C}[\![P]\!] \mid P \in \mathbf{CSP} \wedge P \models SEF \Rightarrow \Diamond e\}$ that asserts that e eventually occurs under the assumption of strong event fairness, cannot be expressed as refinement checks for FDR, as shown by Corollary 6.1.5. Non-causation of these effects may be able to be tested, however, in certain cases by using refinement checks that express sufficient conditions for the effect (or its absence), as we did when testing for liveness properties under notions of event fairness in Chapter 6. However, it is unclear how well this approach scales to arbitrary systems.

We leave open the question as to how to test non-causation and non-prevention of arbitrary non-safety effects for deterministic systems via refinement-checking. We briefly discuss some possibilities in Section 8.1.

7.6.2 Nondeterministic Systems

We now consider to what degree one can test non-causation or non-prevention of some effect E for all deterministic componentwise refinements of a nondeterministic system using FDR[8]. We've seen that for deterministic systems, safety effects are generally easier to test via refinement checks. Therefore, it seems natural to consider just safety effects first.

It turns out that even for the simplest of the safety effects, one cannot in general test for their non-causation/-prevention over all deterministic componentwise refinements of a nondeterministic system by using refinement checking in FDR. Furthermore, this result is unchanged even if we consider testing non-causation/-prevention over *all* refinements of a nondeterministic system.

Consider the safety effect $E_{\mathsf{not}\ e}$ from Equation 7.17 and the system depicted in Figure 7.8, and suppose we wish to decide whether Alice prevents the effect $E_{\mathsf{not\ Bob.Emma.Call.null}}$ (equivalently, whether Alice causes Bob to be able to Call Emma).

[8] For an effect that can be expressed as a finite-state refinement check, one can of course test whether the effect is caused or prevented in some nondeterministic system itself (without considering that system's refinements) by performing at most two refinement checks. However, the refinement paradox (see Section 5.2) makes this approach problematic.

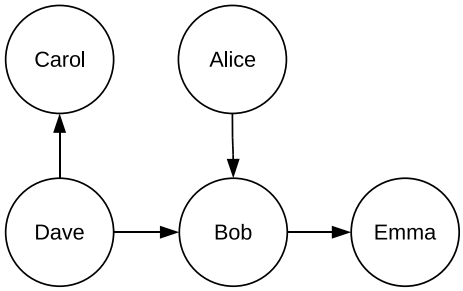

Figure 7.8: A simple example of event causation.

$$\begin{aligned}
&\mathit{behaviour}(\mathsf{Alice}) = \mathsf{Alice.Bob.Call.null} \rightarrow \mathit{STOP},\\
&\mathit{behaviour}(\mathsf{Bob}) =\\
&\quad ?\mathit{from} : \mathit{Capability} - \{\mathsf{Bob}\}!\mathsf{Bob}!\mathsf{Call}!\mathsf{null} \rightarrow \mathsf{Bob.Emma.Call.null} \rightarrow \mathit{STOP},\\
&\mathit{behaviour}(\mathsf{Carol}) =\\
&\quad ?\mathit{from} : \mathit{Capability} - \{\mathsf{Carol}\}!\mathsf{Carol}!\mathsf{Call}!\mathsf{null} \rightarrow \mathit{behaviour}(\mathsf{Carol}),\\
&\mathit{behaviour}(\mathsf{Dave}) =\\
&\quad \mathsf{Dave.Carol.Call.null} \rightarrow \mathit{behaviour}(\mathsf{Dave}) \sqcap \mathsf{Dave.Bob.Call.null} \rightarrow \mathit{STOP},\\
&\mathit{behaviour}(\mathsf{Emma}) = ?\mathit{from} : \mathit{Capability} - \{\mathsf{Emma}\}!\mathsf{Emma}!\mathsf{Call}!\mathsf{null} \rightarrow \mathit{STOP}.
\end{aligned}$$

Snippet 7.2: A system for which non-prevention cannot be tested by refinement-checking.

Letting *Object* and *facets* be defined naturally as one would expect from Figure 7.8, suppose the behaviour of each object is given as in Snippet 7.2, yielding the CSP process $\mathit{System} = \|_{o \in \mathit{Object}} (\mathit{behaviour}(o), \alpha(o))$ that captures the entire system per Definition 2.3.1.

Then consider the deterministic componentwise refinement System_D of this system in which Dave's behaviour is given by the deterministic process b_{Dave}, defined as

$$b_{\mathsf{Dave}} = \mathsf{Dave.Carol.Call.null} \rightarrow b_{\mathsf{Dave}}.$$

System_D is equivalent to the process that behaves like

$$\mathsf{Alice.Bob.Call.null} \rightarrow \mathsf{Bob.Emma.Call.null} \rightarrow \mathit{STOP} \;|||\; b_{\mathsf{Dave}}.$$

Alice clearly causes Bob to be able to Call Emma here. Because $\mathit{System}_D \in \mathit{DCRef}(\mathit{System})$, by Definition 7.1.1, Alice prevents the effect $E_{\mathsf{not\ Bob.Emma.Call.null}}$ in *System*.

Now consider the (infinite) sequence $\langle \mathit{System}_k \mid k \in \mathbb{N} \rangle$ of systems that are identical to *System* except that, for each $k \in \mathbb{N}$, in System_k,

$behaviour(\mathsf{Bob}) = P_k$ where

$$P_0 = \mathsf{Dave.Carol.Call.null} \rightarrow P_0 \triangleright \mathsf{Dave.Bob.Call.null} \rightarrow STOP,$$
$$P_k = \mathsf{Dave.Carol.Call.null} \rightarrow P_{k-1} \sqcap \mathsf{Dave.Bob.Call.null} \rightarrow STOP, \text{for } k > 0.$$

Observe that P_0 can never refuse to perform the event Dave.Bob.Call.null. Hence, in all failures-divergences refinements of each P_k, this event can occur. Hence, in all deterministic componentwise refinements of each $System_k$, the event Bob.Emma.Call.null can occur without Alice acting. Therefore, under Definition 7.1.1, in every $System_k$, Alice is unable to cause Bob to be able to Call Emma (equivalently Alice is unable to to prevent the effect $E_{\mathsf{not\ Bob.Emma.Call.null}}$).

Note that the sequence $\langle System_k \mid k \in \mathbb{N} \rangle$ of processes is strictly decreasing (since each $System_{k+1} \sqsubseteq System_k$ in all standard denotational models of CSP) and that $System$ is the limit of this sequence. Also, each $System_k$ is trace-equivalent to all others, as well as to $System$. So, because all processes here are divergence-free, we may apply Theorem 6.1.4 to conclude that no refinement check exists, in any CSP model that FDR might support, that is equivalent to testing non-prevention of the effect $E_{\mathsf{not}\ e}$ (equivalently, non-causation of the effect $E_{\mathbf{EF}\ e}$ that asserts that e can occur) for all deterministic componentwise refinements of a nondeterministic system. Note that this argument does not rely on any fairness assumptions.

This implies that in general, even for the most simple non-causation and non-prevention properties, Definition 7.1.1 cannot be tested by refinement checking with FDR, and so cannot in general be a safety hyperproperty when instantiated with a non-causation or non-prevention property for $Prop$. In fact, we see that these conclusions would remain unchanged even if we altered Definition 7.1.1 to quantify over all refinements of the system being examined, rather than only its deterministic componentwise refinements. This is because Alice is unable to prevent the effect $E_{\mathsf{not\ Bob.Emma.Call.null}}$ in all refinements of each $System_k$.

We are forced to conclude, therefore, that refinement checking is not very well suited to testing non-causation and non-prevention for nondeterministic systems, and that alternative testing methodologies should be investigated. We briefly consider some possibilities later in Section 8.1.

7.7 Related Work

Counterfactual Causality We based our definition of causation upon Lewis' notion of counterfactual-dependence [Lew73]. This has also been the basis of a number of other definitions for causation, including those of Pearl *et al.* [HP03, Gro05, HP05, HP07]. Unlike ours, these other definitions modify the basic notion of counterfactual dependence in order to arrive at a definition that captures so-called "commonsense" notions of causality.

The following example is often used in the literature to motivate why, and originally appeared in [Hal04].

Consider two children, Billy and Suzy. Suzy and Billy both picked up rocks and threw them at a bottle. Suzy's rock got there first, shattering the bottle. Both throws were perfectly accurate and occurred simultaneously. In this scenario, the bottle shattering is *not* counterfactually dependent on Suzy having thrown her rock since if she hadn't thrown, Billy's rock would have hit the bottle and caused it to shatter. However, it has been repeatedly argued (see *e.g.* [HP03, Hal04, HP05, HP07, CHK08]) that concluding that Suzy's throw did not cause the bottle to shatter defies commonsense reasoning. This argument then leads to the conclusion that counterfactual dependence is not, on its own, a good measure of causation here.

However, this conflates two questions about causation, namely (1) "Did Suzy's throw cause the bottle to shatter?" and (2) "Did Suzy have the authority to cause the bottle to shatter?" While counterfactual dependence might not be entirely appropriate for deciding the first question, it is certainly appropriate for deciding the second. In this scenario, no matter what Suzy chose to do on her own, Billy still would have thrown. Suzy had no real power, therefore, to alter whether the bottle was going to shatter or not. Hence, she rightly had no authority in this regard.

Another approach that, like ours, applies Lewis' notion of counterfactual dependence directly as a definition for causation is that of Groce [Gro05]. Groce's definition of causation considers linear event traces and the question of whether one event $\mathcal{A}$ causes a subsequent event $\mathcal{B}$ to occur in a particular trace. Groce's definition considers all counterfactual traces (*i.e.* ones that differ from the observed trace) of the system in question in which $\mathcal{A}$ doesn't occur and ranks them according to how alike each is to the observed trace. If $\mathcal{B}$ is absent in all of the most alike traces to the observed one, then $\mathcal{A}$ is judged to be the cause of $\mathcal{B}$ in the observed trace. In this way, Groce's definition precisely encodes Lewis' notion of counterfactual dependence.

The ranking of the counterfactual traces according to how alike they are to the observed one is achieved straightforwardly via a *distance metric*. In our approach, no such distance metric is required since there is only one counterfactual scenario to consider, namely $Sys\,|_A$ (see *e.g.* Equation 7.7).

Authority Analysis In earlier work [ML07] trying to capture authority via causation, we devised a different definition for causation based on Lewis' notion of counterfactual dependence. This definition, like Groce's, considered causation within individual traces. It states that in some trace $s^\frown\langle e\rangle$ of a deterministic system Sys, that some object with alphabet A doesn't cause the event e to occur in this trace iff $s \setminus A^\frown\langle e\rangle \in traces(Sys)$. This definition for non-causation was extended to nondeterministic systems by taking its refinement-closure, which can be checked automatically in FDR by using a refinement test that runs two copies of the system being analysed.

Unfortunately, this definition does not always yield correct answers. For instance, consider the system Sys that is defined as

$$Sys = a \rightarrow b \rightarrow STOP \;|||\; e \rightarrow STOP,$$

in which the occurrence of the event a clearly never causes the event e to occur. However, the definition from [ML07] says that a causes e to occur in the trace $\langle a, b, e\rangle$ of Sys since $\langle a, b, e\rangle \setminus \{a\} = \langle b, e\rangle$ is not a trace of Sys. For this reason, we consider this definition of causation to be far less useful than one would like.

These problems are avoided by the definitions used in this thesis which avoid considering individual execution traces, and instead ask whether the presence of an object can cause an effect (like the possible or inevitable occurrence of an event) to occur in some system. Of course, the approach taken in this thesis is also much more general, since it can reason not only about causing event occurrence but also a range of other effects in which one might be interested, as demonstrated in Section 7.3.

Analysing Authority Propagation In [Spi07, Chapter 9] and [SQV06], Spiessens *et al.* consider the use of directed graphs, called *authority flow graphs*, to model and reason about the propagation of authority in systems of interacting components. The nodes of an authority flow graph are the components of a system and its edges capture how authority may flow between the system's components. The definition of a component's authority, used by Spiessens *et al.* here, is defined somewhat informally as its ability "to directly or indirectly induce an effect" [SQV06]. This definition is not, therefore, explicitly wedded to any specific notion of causation, unlike our notion of authority which is defined in terms of counterfactual dependence.

The transitive closure of an authority flow graph captures the reachable authority in a system [SQV06, Spi07]. Given a system and some kind of authority, whose propagation in that system we are interested in, one naturally defines constraints on the propagation of this authority by imposing constraints on the corresponding authority flow graph and its transitive closure. Spiessens *et al.* show how finding authority flow graphs that capture systems in which certain authority flow constraints are satisfied, can be captured in terms of the *bounded transitive closure* problem, and how this problem can be solved automatically by using Quesada's *DomReachability* [QVDC06] graph constraint solver.

This work differs to ours because it is not concerned with detecting a component's possible authority, in terms of counterfactual causation, but is instead concerned with finding configurations of components, and the flows of authority between them, under which certain authority flow constraints are satisfied.

CSP Specification Slicing In [LLO+09], Leuschel *et al.* consider the problem of CSP specification *slicing* and present two static analysis techniques known as *must be executed before* (MEB) and *could be executed before* (CEB). Each of these techniques operates over the syntax of a CSP process. Given an event that is mentioned in a process's syntax, the MEB and CEB analyses return those parts of the process's syntax that *must be* and *could be* executed respectively before this event occurs.

For instance, in the process $P = a \rightarrow c \rightarrow STOP \sqcap b \rightarrow c \rightarrow STOP$, for the event c, the CEB analysis would report that the underlined syntax in "$\underline{a \rightarrow} \; c \rightarrow STOP \; \underline{\Box} \; \underline{b \rightarrow} \; c \rightarrow STOP$" could be executed before c occurs. The MEB analysis would report that the underlined syntax in "$a \rightarrow c \rightarrow STOP \; \underline{\Box} \; b \rightarrow c \rightarrow STOP$" must be executed before c occurs; this doesn't include the events a and b because P has executions in which c can occur without a occurring and similarly for b.

There is a sense in which this slicing information could be useful for inferring information about causation. However, the analysis techniques presented by Leuschel *et al.* fail to distinguish between external and internal choice, *i.e.* they treat "$\Box$" and "$\sqcap$" identically. This makes the results obtained from these analyses less useful than one might like for nondeterministic processes. In particular, P above has the refinement Q where $Q = a \rightarrow c \rightarrow STOP$. In Q, a *must* clearly occur for c to occur; this would be reflected in the MEB analysis of Q. However, this information is not reflected in the MEB analysis for P.

Hence, these slicing techniques, whilst sharing some intuitive similarities with our ideas of counterfactual causation, cannot be used directly to infer useful information about event causation for nondeterministic processes.

Non-Counterfactual Causality Causation has also been studied in the context of other process algebras. However, the kind of causation typically examined in these other contexts is not based on counterfactual dependence and is therefore less useful than one would like for our purpose of reasoning about authority.

For instance, Sewell and Vitek consider causality in the context of their *box-π calculus* [SV03]. In this approach, the influence of each principal in a system is tracked through a *colouring* semantics, in which each principal in the system is assigned a colour and the edges in the system's operational semantics (*i.e.* in its labelled transition system) are labelled with these colours as the system evolves to reflect those principals whose actions may have influenced the system's current execution.

For instance, consider a system that comprises the four principals Alice, Bob, Carol and Dave in which Alice and Bob each initially try to Call Carol. Carol waits to be called, at which point she Calls Dave. This system could

be captured by the CSP process $System12$ defined as

$$System12 = \mathsf{Alice.Carol.Call.null} \rightarrow \mathsf{Carol.Dave.Call.null} \rightarrow STOP \;\Box \\ \mathsf{Bob.Carol.Call.null} \rightarrow \mathsf{Carol.Dave.Call.null} \rightarrow STOP.$$

Here, the first Carol.Dave.Call.null-event would carry the colours of Alice and Carol, while the second would carry the colours of Bob and Carol.

This analysis says that Alice can cause the the first Carol.Dave.Call.null-event and Bob can cause the second. However, neither has the authority to cause Dave to be called in this system under a definition based on counterfactual dependence, which we've shown is most useful in this chapter.

Similar problems exist with other process algebra semantics that primitively incorporate causal information. These include so-called "true" concurrency semantics, which usually incorporate information about the causal relationship between events. *Event structures* [Win89] are a notable example with many variations.

Under typical event structure encodings of CSP-like languages, such as van Glabbeek and Vaandrager's *bundle event structures* encoding from [vGV03], event a causes event b in system trace s when a comes before b in all system traces containing the same events as s [LBK97]. Under such an encoding, the event Alice.Carol.Call.null would be identified as a unique cause of the event Carol.Dave.Call.null in the trace $\langle \mathsf{Alice.Carol.Call.null}, \mathsf{Carol.Dave.Call.null} \rangle$ of $System12$ above. However, this identification is not very useful for reasoning about authority here.

Responsiveness In [RSR04, RRS05], Reed *et al.* consider the problem of determining when, given two processes P and Q whose alphabets are αP and αQ respectively, such that $\alpha Q \subseteq \alpha P$, whether composing some refinement R_Q of Q with some refinement R_P of P to form $R_P \;{}_{\alpha P}\|_{\alpha Q}\; R_Q$, can cause R_P to block on some set of events $A \subseteq \alpha P \cap \alpha Q$ when R_P otherwise would not have on its own. The absence of this causation is captured by the property $RespondsToLive_A(R_P, R_Q)$ holding for all refinements R_P and R_Q of P and Q respectively.

$$RespondsToLive_A(R_P, R_P) = \\ \forall\, s \bullet (s, A) \in failures(R_P \;{}_{\alpha P}\|_{\alpha Q}\; R_Q) \Rightarrow (s, A) \in failures(R_P).$$

Observe how similar this property is in spirit to the non-causation properties defined in this chapter. Unlike the properties in this chapter, the counterfactual comparison in responsiveness compares R_P composed with R_Q, which naturally restricts the behaviours of R_P, against R_P on its own, where no such restrictions exist. In this way, responsiveness asserts that the addition of a component to a system that *restricts* the possible behaviours of the rest of the system doesn't cause another component to be blocked. Our non-causation properties, on the other hand, assert that the addition

of a component to a system that *enables* more behaviours in the rest of the system doesn't cause certain effects.

Unlike our non-causation properties, responsiveness can be expressed as a CSP refinement check and so automatically tested by FDR. It would be interesting to investigate further the similarities and differences between our non-causation properties and the notion of responsiveness.

Fault Tolerance Finally, our non-causation properties, including our formulations of defensive correctness and consistency, seem to share some intuitive similarities with certain formulations of *fault tolerance* [Ros97, Section 12.3]. These formulations of fault tolerance involve a counterfactual comparison between two scenarios involving the same system: one in which errors and faults can be introduced arbitrarily, and another when all such faults are absent. It would be worth investigating further the connections between these ideas.

7.8 Conclusion

In this chapter, we have considered how one might reason about authority in object-capability systems, modelled in CSP. We developed a framework for expressing general non-causation properties that is able to express various kinds of effects, whose causation is defined simply in terms of counterfactual dependence. In our framework, an effect is encoded by the corresponding (completed) refusal-traces property that holds for all divergence-free processes in which that effect is present. The flexibility of this approach is evident in the various kinds of authority that our framework can capture, including delegable, non-delegable, revocable and single-use authority.

We saw that one could distinguish those effects involving safety from those that involve liveness, by identifying each effect with an equivalent refusal-traces hyperproperty [CS10]. The safety and liveness effects are then simply those that correspond to safety and liveness hyperproperties respectively. We found that Miller's notions of defensive correctness and consistency can be expressed straightforwardly within our framework. We argued that defensive correctness naturally asserts the non-prevention of an effect that involves both a safety component and a liveness one, whilst defensive consistency asserts the non-prevention of just the safety component.

We also found, however, that the flexibility our approach makes it difficult to test non-causation properties using automatic refinement-checking. We saw that every safety effect can be expressed as a CSP refinement test. We concluded that this allows one to judge non-causation of those safety effects, and their complements, whose associated refinement check is finite-state, for deterministic systems by performing at most two refinement checks in FDR. This may not be possible for other effects however, particularly those that cannot be expressed in the form of CSP refinement checks for

FDR. An example is the effect of some event e inevitably occurring under strong event fairness, which is equivalent to the LTL property $SEF \Rightarrow \Diamond e$, which we saw in Chapter 6 cannot be expressed as a refinement check for FDR.

For a nondeterministic system, we argued that one must judge non-causation by considering all of its deterministic componentwise refinements. We found that even for the simplest of safety effects, which can be easily expressed as CSP refinement checks, one cannot always express their non-prevention (equivalently non-causation of their complements) for all deterministic componentwise refinements of some systems as a refinement check for FDR. We showed that the same is also true if one wants to instead test this for all refinements of a system. We are forced to conclude, therefore, that refinement checking is ill-suited to testing non-causation properties of nondeterministic systems. We argue that alternative testing methodologies should be investigated for this purpose. We discuss some possibilities later in Section 8.1.

8 Conclusion

In this thesis, we have examined the use of the process algebra CSP, and its automatic refinement-checker FDR, for analysing security properties of object-capability patterns.

We've seen that CSP is naturally very expressive, and can be used to model most, if not all, of the wide variety of features and differences that exist between current object-capability systems (see Table 2.1) with ease and accuracy. For instance, we've seen that CSP can express both concurrent and single-threaded systems (see Section 2.3.5), as well as recursively invocable objects (see *e.g.* Snippet 3.10), non-blocking invocation (see Section 6.2) and the *EQ* primitive (see *e.g.* Snippet 3.2). CSP's expressiveness is also an asset when formalising security properties as refinement checks to be automatically carried out by FDR. In Chapter 3, we saw that complex safety properties, like safe coercion (see Snippet 3.9), can be easily expressed in CSP as traces refinement checks by defining suitable specification processes. We showed in that chapter that CSP's ability to express interesting systems and safety properties allows one to automatically detect vulnerabilities in patterns that arise due to recursive and concurrent invocation.

CSP's rich body of semantic theory has also been a major asset. In Chapter 4, we saw that CSP's theory of data-independence is particularly useful for allowing one to generalise the results of analysing small fixed-sized systems to systems of arbitrary size. We showed that this can be done by treating a fixed-sized system as a safe abstraction of a set of arbitrary-sized systems, in which the behaviour of multiple objects has been aggregated into a single one, and that the theory of data-independence could be applied to generalise this analysis to all such arbitrary-sized systems. This also allowed us to model and reason about patterns that make use of unbounded object creation. In particular, we showed that this approach allowed one to detect subtle differences in the revocation property upheld by a revocable Membrane implementation when deployed in single-threaded and concurrent systems respectively.

In Chapter 5, CSP's denotational theory of refinement was used to encode the necessary and intuitive assumption that must be made when analysing the information flow properties of an object-capability pattern, namely that each object can directly influence the others only through its overt interactions with them. We saw that this assumption could be

easily expressed using CSP's theory of refinement by defining that a system is secure under some information flow property iff that property holds for all of the system's deterministic componentwise refinements (see Definition 5.2.2). We showed that traditional noninterference properties can be adapted to take this assumption into account, producing the class of weakened refinement-closed noninterference properties (see Definition 5.3.18). We saw that these properties can be readily tested using FDR by expressing them as CSP refinement checks. We showed that this approach allows one to diagnose covert channels present in an object-capability pattern, and to generalise the analysis of such patterns to systems of arbitrary size with a slight extension of the theory developed earlier in Chapter 4.

CSP's rich diversity of semantic models was also invaluable for allowing us to formally state liveness properties of object-capability patterns under fairness assumptions in Chapter 6. We saw that such properties could be encoded in a fragment of LTL, whose semantics (from Lowe [Low08]) was defined over the refusal-traces model. This allowed us to express an intuitive liveness property to be analysed of the Trademarks pattern. We showed that this liveness property could be verified by performing certain stable-failures refinement checks in FDR that together constituted a sufficient condition for the property.

We also saw in Chapter 7 that the refusal-traces model could be used as a base on which to build a flexible framework for expressing general non-causation properties. We showed how the framework that we developed in that chapter is capable of expressing interesting and complicated elements of an object's authority, such as non-delegable and single-use authority, as well as the intuitive notions of defensive correctness and consistency. These results indicate that CSP, and its associated denotational semantic models, are apt for formalising complicated security properties.

However, we also found that CSP refinement checking with FDR, whilst being incredibly useful for reasoning about safety and information flow properties of object-capability patterns, is not powerful enough to be used to analyse the full range of security properties in which one might be interested. In Chapter 6 we saw, that while CSP's semantic models are more than adequate for expressing liveness properties under fairness assumptions, that such properties cannot always be precisely expressed as refinement checks in any such model that FDR might support. Similarly in Chapter 7, while we showed that non-causation of safety effects and their complements can be expressed as CSP refinement checks for deterministic systems, we also saw that no refinement check, in any standard CSP model that FDR might support, could express certain very simple non-causation properties for nondeterministic systems. We argue, therefore, that alternative approaches should be investigated for verifying these kinds of properties.

In summary, we conclude that CSP and FDR are, together, a very useful combination for analysing the security properties of object-capability pat-

terns. However, more work is required in order to extend the reach of this approach beyond safety and information flow properties to include *e.g.* liveness and non-causation properties as well. Fortunately, CSP's strong foundation of rich semantic models provides the perfect base from which such further work can proceed.

8.1 Future Work

We conclude this thesis by considering avenues for future work.

Automatic Analysis of Liveness Properties In Chapter 6, we saw that liveness properties like $SEF \Rightarrow \Diamond e$ cannot be expressed as CSP refinement checks for FDR, preventing FDR from being able to automatically check them directly. We argue that alternative automatic testing approaches should be examined for these kinds of property. In particular, we conjecture that existing work (such as [Ros01, SLDW08, Liu09]) on testing liveness properties by examining a system's operational semantics could probably be adapted to allow one to automatically verify the kinds of liveness properties considered in Chapter 6. We sketch one such possibility, based on the work in [Liu09].

The standard explicit-state, automata-based approach [VW86] to testing liveness properties expressed as LTL formulae ϕ against a system's operational semantics $\mathcal{A}$ involves first constructing [WVS83] a Büchi automaton [Büc62] $\mathcal{B}$ that corresponds to the LTL formula $\neg\phi$, *i.e.* a Büchi automaton that accepts those and only those infinite behaviours that violate the liveness property ϕ. One then constructs the *product* $\mathcal{A} \times \mathcal{B}$ of $\mathcal{A}$ and $\mathcal{B}$, which is a Büchi automaton that accepts all infinite behaviours that can be exhibited by $\mathcal{A}$ that are accepted by $\mathcal{B}$. Then the liveness property ϕ is satisfied by the system iff the language recognised by $A \times B$ is empty. This occurs precisely when $\mathcal{A} \times \mathcal{B}$ contains no reachable non-trivial strongly connected subgraph (SCS) that is *accepting*, *i.e.* an SCS that contains a node $(s_{\mathcal{A}}, s_{\mathcal{B}})$, where naturally $s_{\mathcal{A}}$ and $s_{\mathcal{B}}$ are states of $\mathcal{A}$ and $\mathcal{B}$ respectively, for which $s_{\mathcal{B}}$ is an accepting state of $\mathcal{B}$.

Testing a liveness property ϕ_L (*e.g.* $\Diamond e$) under a fairness assumption ϕ_F (*e.g. SEF*) is equivalent to testing the property $\phi_F \Rightarrow \phi_L$ (*e.g.* $SEF \Rightarrow \Diamond e$). Hence, one way to test liveness under fairness using this standard approach involves building a Büchi automaton $\mathcal{B}$ that corresponds to the formula $\neg(\phi_F \Rightarrow \phi_L)$ and then computing $\mathcal{A} \times \mathcal{B}$ as usual. However, the construction of $\mathcal{B}$ usually scales poorly with the size of the LTL formula to which it corresponds (in the worst case, scaling exponentially). Recall that the fairness assumptions that we've used in this thesis, namely *SEF* and *WEF* from Definition 6.1.2, involve a conjunction over every event in Σ. Previous work by others [SLDW08, Liu09] examining the application of this approach using the SPIN model checker [Hol03] with similar fairness

assumptions, indicate that it is unlikely to work well when Σ contains more than a few events, and is therefore infeasible for us.

This problem is avoided in [Liu09, Chapter 4] by constructing $\mathcal{B}$ to correspond to just the formula $\neg\phi_L$. The accepting SCSs of $\mathcal{A}\times\mathcal{B}$ then correspond to all behaviours of $\mathcal{A}$ that violate ϕ_L, whether fair or unfair. Unfair behaviours that violate ϕ_F are then *pruned* away by algorithmically identifying the *fair* SCSs of the product that (correspond to behaviours that) satisfy the fairness assumption ϕ_F. The system then satisfies $\phi_F \Rightarrow \phi_L$ iff none of these fair SCSs are accepting.

Adapting this approach therefore requires one to be able to identify whether the behaviours captured by an SCS of some product $\mathcal{A} \times \mathcal{B}$ satisfy our fairness assumptions *SEF* and *WEF*. We briefly sketch how to do so.

Let S be a non-trivial SCS in the product $\mathcal{A}\times\mathcal{B}$ of a system's operational semantics $\mathcal{A}$ and a Büchi automaton $\mathcal{B}$. Let N_S and E_S be the sets of states and edges respectively of S. When $s_\mathcal{A}$ and $s'_\mathcal{A}$ are states of $\mathcal{A}$, we write $s_\mathcal{A} \xrightarrow{x} s'_\mathcal{A}$ to mean that from state $s_\mathcal{A}$ the system can transition to state $s'_\mathcal{A}$ by performing the event x from $\Sigma \cup \{\tau\}$, where τ is the special event used to represent internal activity [Ros97, Chapter 7]. We also write $s_\mathcal{A} \xrightarrow{x}$ to mean that there exists a state $s'_\mathcal{A}$ of $\mathcal{A}$ such that $s_\mathcal{A} \xrightarrow{x} s'_\mathcal{A}$. Then for each node $(s_\mathcal{A}, s_\mathcal{B}) \in N_S$, let

$$\mathit{stableStates}((s_\mathcal{A}, s_\mathcal{B})) = \{s'_\mathcal{A} \mid s_\mathcal{A}(\xrightarrow{\tau})^* s'_\mathcal{A} \wedge s'_\mathcal{A} \not\xrightarrow{\tau}\}$$

denote the set of states $s'_\mathcal{A}$ reachable from $s_\mathcal{A}$ in the system's operational semantics under zero or more τ-transitions such that each $s'_\mathcal{A}$ is stable.

Then for each $(s_\mathcal{A}, s_\mathcal{B}) \in N_S$, let

$$\begin{aligned}&\mathit{availableEvents}((s_\mathcal{A}, s_\mathcal{B})) = \\ &\quad \{e \mid e \in \Sigma \wedge \forall\, s'_\mathcal{A} \in \mathit{stableStates}((s_\mathcal{A}, s_\mathcal{B})) \bullet s'_\mathcal{A} \xrightarrow{e}\}\end{aligned}$$

denote the set that contains those events e that are available from every τ-reachable stable state $s'_\mathcal{A}$ in the system's operational semantics from $s_\mathcal{A}$. Then, let

$$\begin{aligned}\mathit{sometimesAvailableEvents}(S) &= \textstyle\bigcup_{(s_\mathcal{A},s_\mathcal{B})\in N_S} \mathit{availableEvents}((s_\mathcal{A}, s_\mathcal{B})),\\ \mathit{alwaysAvailableEvents}(S) &= \textstyle\bigcap_{(s_\mathcal{A},s_\mathcal{B})\in N_S} \mathit{availableEvents}((s_\mathcal{A}, s_\mathcal{B})),\end{aligned}$$

be the sets that contain those events that are sometimes and always respectively stably available at some point during S. Let $\mathit{performedEvents}(S)$ be the set of events performed in S, *i.e.*

$$\begin{aligned}&\mathit{performedEvents}(S) = \\ &\quad \{e \mid e \in \Sigma \wedge \exists (s_\mathcal{A}, s_\mathcal{B}), (s'_\mathcal{A}, s'_\mathcal{B}) \in N_S \bullet ((s_\mathcal{A}, s_\mathcal{B}), e, (s'_\mathcal{A}, s'_\mathcal{B})) \in E_S\}.\end{aligned}$$

Then we conjecture that S satisfies *SEF* iff

$$\mathit{sometimesAvailableEvents}(S) \subseteq \mathit{performedEvents}(S).$$

Similarly, we conjecture that S satisfies WEF iff

$$alwaysAvailableEvents(S) \subseteq performedEvents(S).$$

For example, consider the process $P = a \rightarrow P \triangleright b \rightarrow STOP$. Under our LTL semantics from Chapter 6, $P \models WEF \Rightarrow \Diamond b$ and similarly for SEF. P's operational semantics $\mathcal{A}$ has two transitions from its initial state: one labelled with a, which is a self-loop, and another labelled with τ that leads to a state from which the only transition available is labelled with b and leads to a terminal state with no outgoing transitions. The product $\mathcal{A} \times \mathcal{B}$ of $\mathcal{A}$ and the Büchi automaton $\mathcal{B}$ that corresponds to the formula $\neg \Diamond b$, has just a single state that has a single transition, namely a self-loop labelled with a. $\mathcal{A} \times \mathcal{B}$ has just one non-trivial SCS, which we denote S; S is, in fact, the entire automaton. S is accepting.

By the above definitions, we have that $sometimesAvailableEvents(S) = alwaysAvailableEvents(S) = \{b\}$, but that $performedEvents(S) = \{a\}$. We see that $\{b\} \not\subseteq \{a\}$. This indicates that, under the conjectures above, none of the behaviours captured by this SCS that violate the liveness property $\Diamond e$, satisfy either of these fairness assumptions. Hence, under these conjectures, none of the behaviours present in $\mathcal{A}$ that violate $\Diamond e$ satisfy SEF or WEF. This is, of course, consistent with P satisfying $WEF \Rightarrow \Diamond b$ and $SEF \Rightarrow \Diamond b$.

Analysing Non-Causation Properties In Chapter 7, we concluded that refinement-checking is not well suited for testing non-causation properties of nondeterministic systems. One obvious avenue for future work involves identifying ways in which these kinds of property can be mechanically verified.

One potential avenue would be to analyse these properties by using mechanised logical proof directly over the denotational semantic models in which they are defined (*i.e.* directly over the refusal-traces model) with the aid of mechanical theorem proving technologies. The CSP-Prover [IR05, IR08] tool could be particularly useful here.

CSP-Prover is a collection of theory libraries and proof tactics for the Isabelle [Pau94] proof assistant. CSP-Prover encodes many of CSP's standard denotational semantic models as Isabelle theories, allowing one to write CSP processes and prove semantic refinement between them using Isabelle's interactive theorem proving interfaces. CSP-Prover has also been used to reason about CSP's denotational semantic models themselves [IR06, SRI09].

We conjecture that one could extend CSP-Prover to allow one to state and prove non-causation properties of CSP processes. Doing so would first require the refusal-traces model to be formalised in CSP-Prover. Given [IR06, SRI09] that a number of other models, including the stable-failures and *stable-revivals* [Ros09] models, have been formalised in CSP-Prover, we expect that formalising the refusal-traces model should be a relatively straightforward task of adapting these existing encodings. Having

formalised the refusal-traces model, one would then likely identify a number of key lemmas and results to be proved, regarding non-causation properties, that would assist generally in proving non-causation properties of nondeterministic processes.

Besides this approach based on interactive theorem proving, it remains to be seen whether mechanisms for the automatic verification of non-causation properties by model-checking may be found. It can be observed that all of the properties that we have proved in this thesis cannot be tested by automatic refinement checking, involve detecting the presence of certain infinite behaviours that are not infinite traces. This can be seen by examining Theorem 6.1.4 and noting that what sets B^* there apart from each B_k must be some infinite behaviour that is not an infinite trace (since B^* is trace-equivalent to each B_k). As a concrete example, in Section 7.6.2, what sets *System* (where causation does exist) apart from each $System_k$ (in which causation doesn't exist), is the set of infinite refusal-traces in which the event Dave.Bob.Call.null is always stably refused. These refusal-traces are present in *System* but not in any $System_k$.

One way to understand why these properties cannot be tested by refinement checking is to observe that the only way to test for infinite behaviours using CSP refinement checking is to map those infinite behaviours onto corresponding divergences using hiding, and then assert the absence of such divergences. Recall that it is precisely this approach that is taken to test the liveness property $\Diamond\, e$ under no fairness assumption (see Equation 6.1). An infinite behaviour that is not an infinite trace cannot be mapped in this way onto a corresponding divergence, because the hiding ignores the extra refusal/acceptance information that is necessarily present in the behaviour.

As implied by the discussion above in the context of liveness properties, model-checking algorithms based on identifying certain SCSs within the operational semantics of a system are appropriate for testing properties that involve detecting certain infinite behaviours (see *e.g.* [Ros01]). Hence, we conjecture that it might be possible to adapt pre-existing SCS-based techniques to detect infinite behaviours that cannot be observed using refinement-checking, *i.e.* those infinite behaviours that are not infinite traces. This might allow one to model-check properties that cannot otherwise be tested using refinement-checking, including non-causation properties.

Timed Information Flow In Chapter 5, we showed that information flow properties of object-capability patterns should be tested under the assumption that the only way for objects to influence each other directly is by exchanging messages. This assumption was naturally encoded by asserting that an information flow property holds for a system iff that property holds for all deterministic componentwise refinements of that system (see Definition 5.2.2).

While useful, this assumption may be overly restrictive in some cases.

In particular, it assumes that objects do not have access to a global clock or other sources of timing information, and therefore makes it difficult to assert the absence of possible *timing channels* in object-capability patterns.

Huang and Roscoe [HR06] have considered the problem of expressing noninterference properties for *timed systems* that contain a global clock and in which each entity may have access to timing information. They show how to adapt traditional noninterference properties, including Roscoe's Lazy Independence (see Proposition 5.3.3), to this timed setting.

One promising avenue of future work therefore involves extending Huang and Roscoe's work to provide a characterisation of *timed weakened refinement-closed noninterference properties*, which would adapt our notion of a weakened refinement-closed noninterference property (see Definition 5.3.18) to this timed setting. Doing so should allow us to analyse object-capability patterns to detect possible timing channels within them.

Analysing Source Code Finally, perhaps the most interesting area for future work involves applying the approaches developed in this thesis to the analysis of object-capability patterns expressed directly in source code. This source code would either be C code, in the case of a pattern deployed in an object-capability operating system, or code in some object-capability language like E or Cajita.

The most straightforward way to apply the work in this thesis to the analysis of such source code would be to translate the source code into an appropriate representation in CSP. This translation would produce a CSP system, similar to those crafted by hand in this thesis, that models the pattern and its environment. One could then apply the analysis techniques developed in this thesis to the automatically generated CSP representation of the pattern to analyse the pattern's security properties.

CSP has often been used as a target language into which higher-level languages can be translated (or *compiled*) in order to formally analyse systems expressed in the higher-level language. This basic approach has been applied to the analysis of cryptographic protocols expressed in a high-level domain specific language [Low98], shared-variable multithreaded programs expressed in a C-like language [Ros01, RH07] and web services protocols expressed in SOAP [Kle08] to name a few. Hence, we believe that this approach would be likely to yield useful results. A good first step would involve investigating to what degree Roscoe *et al.*'s SVA [Ros01, RH07] tool could be applied directly to this problem.

In order to ensure that the approach is sound, one could produce (or borrow, see *e.g.* [Nor98]) a formal semantics for the source language and prove that the translation from the source language to CSP preserves the security properties being analysed. Having done so, one could then be confident that the translation does not hide real vulnerabilities in the source nor introduce spurious ones in its CSP translation.

Bibliography

[AAH+85] Mack W. Alford, Jean-Pierre Ansart, Günter Hommel, Leslie Lamport, Barbara Liskov, Geoff P. Mullery, and Fred B. Schneider. *Distributed systems: methods and tools for specification. An advanced course.* Springer-Verlag, New York, NY, USA, 1985. 129

[Abr91] Samson Abramsky. Domain theory in logical form. *Annals of Pure and Applied Logic*, 51(1-2):1–77, 1991. 192

[AFK88] Krzysztof R. Apt, Nissim Francez, and Shmuel Katz. Appraising fairness in languages for distributed programming. *Distributed Computing*, 2(4):226–241, 1988. 130, 131, 138, 181

[AL91] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991. 138, 181

[And72] James P. Anderson. Computer security technology planning study, Volume 2. Technical Report ESD-TR-73-51, Electronic Systems Division, Air Force Systems Command, Hanscom Field, Bedford MA, USA, October 1972. 173

[APW86] M. Anderson, R. D. Pose, and C. S. Wallace. A password capability system. *The Computer Journal*, 29(1):1–8, 1986. 14

[AS85] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985. 128, 129, 181, 182, 183

[BBDC+09] Ilan Beer, Shoham Ben-David, Hana Chockler, Avigail Orni, and Richard Trefler. Explaining counterexamples using causality. In *Proceedings of the 21st International Conference on Computer Aided Verification (CAV '09)*, volume 5643 of *Lecture Notes in Computer Science*, pages 94–108. Springer-Verlag, 2009. 163

[Bis96] Matt Bishop. Conspiracy and information flow in the take-grant protection model. *Journal of Computer Security*, 4(4):331–360, 1996. 125

[BL76] David E. Bell and Leonard J. LaPadula. Secure computer system: Unified exposition and MULTICS interpretation. Technical Report MTR-2997 Rev. 1, The MITRE Corporation, Bedford, MA, USA, March 1976. 92, 173

[Boy09] Andrew Boyton. A verified shared capability model. In *Proceedings of the 4th Workshop on Systems Software Verification (SSV '09)*, volume 254 of *Electronic Notes in Theoretical Computer Science*, pages 25–44, 2009. 51

[Bri07] Marcus Brinkmann. Membrane implementations?, January 2007. E-mail communication to the `cap-talk` mailing list, available at: `http://www.eros-os.org/pipermail/cap-talk/2007-January/007342.html`. 76

[Bro01] Philippa J. Broadfoot. *Data independence in the model checking of security protocols*. D.Phil. thesis, Oxford University Computing Laboratory, September 2001. 88

[Bry05] Jeremy Bryans. Reasoning about XACML policies using CSP. In *Proceedings of the 2005 workshop on Secure Web Services (SWS '05)*, pages 28–35. ACM Press, 2005. 3

[Büc62] J. R. Büchi. On a decision method in restricted second order arithmetic. In *Proceedings of the 1st International Congress on Logic, Methodology, and Philosophy of Science*, pages 1–11. Stanford University Press, 1962. 204

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986. 169

[CHK08] Hana Chockler, Joseph Y. Halpern, and Orna Kupferman. What causes a system to satisfy a specification? *ACM Transactions on Computational Logic*, 9(3):1–26, 2008. 163, 196

[Clo06] Tyler Close. Credit transfer for market-based infrastructure. In *Proceedings of the 10th International Conference on Financial Cryptography and Data Security (FC '06)*, volume 4107 of *Lecture Notes in Computer Science*, pages 160–165. Springer, 2006. 30

[Clo08] Tyler Close. The `ref_send` API, 2008. Available at: `http://waterken.sourceforge.net/`. 16

[CS08] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF '08)*, pages 51–65, 2008. 162, 179, 180

[CS10] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, 2010. To appear. Preprint available at: `http://www.cs.cornell.edu/fbs/publications/Hyperproperties.JCS.pdf`. 5, 162, 179, 180, 182, 183, 192, 200, 227

[DEE08] Philip Derrin, Dhammika Elkaduwe, and Kevin Elphinstone. *seL4 Reference Manual*. NICTA, 2008. Available at: `http://www.ertos.nicta.com.au/research/sel4/sel4-refman.pdf`. 1, 15, 16, 147

[DH66] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–154, March 1966. 1

[Don76] James E. Donnelley. A distributed capability computing system. In *Proceedings of the Third International Conference on Computer Communication*, pages 432–440, 1976. 72

[Don81] James E. Donnelley. Managing domains in a network operating system. In *Proceedings of the Local Networks and Distributed Office Systems Conference*, May 1981. 174

[DY08] P. Dinges and N. Yonezaki. Structural operational semantics for an idealised object-capability programming language. In *Proceedings of the 25th Convention of the Japan Society for Software Science and Technology*, 2008. 15

[EKE08] Dhammika Elkaduwe, Gerwin Klein, and Kevin Elphinstone. Verified protection model of the seL4 microkernel. In *Proceedings of the 2nd International Conference on Verified Software: Theories, Tools, Experiments (VSTTE '08)*, pages 99–114. Springer, 2008. 1, 15, 51

[FG95] Riccardo Focardi and Roberto Gorrieri. A classification of security properties for process algebras. *Journal of Computer Security*, 3(1):5–33, 1995. 93, 99, 102

[Foc96] Riccardo Focardi. Comparing two information flow security properties. In *Proceedings of the 9th IEEE Computer Security Foundations Workshop (CSFW '96)*, pages 116–122. IEEE Computer Society, 1996. 93, 99

[For99] Richard Forster. *Non-Interference Properties for Nondeterministic Processes*. D.Phil. thesis, Oxford University Computing Laboratory, 1999. 93, 99

[GCS91] John Graham-Cumming and J. W. Sanders. On the refinement of non-interference. In *Proceedings of the 4th IEEE Computer Security Foundations Workshop (CSFW '91)*, pages 35–42. IEEE Computer Society, 1991. 93

[GGH+05] Paul Gardiner, Michael Goldsmith, Jason Hulance, David Jackson, Bill Roscoe, Bryan Scattergood, and Philip Armstrong. *Failures-Divergences Refinement: FDR2 User Manual*. Formal Systems (Europe) Ltd, 2005. 2

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. 2

[GM82] Joseph A. Goguen and José Meseguer. Security policies and security models. In *Proceedings of the 1982 IEEE Symposium on Security and Privacy (SP '82)*, pages 11–20, 1982. 93

[GMO+07] Duncan Grove, Toby Murray, Chris Owen, Chris North, Jeremy Jones, M. R. Beaumont, and B. D. Hopkins. An overview of the Annex system. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC '07)*, pages 341–352, 2007. 1, 15

[Gon89] Li Gong. A secure identity-based capability system. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy (SP '89)*, pages 56–65. IEEE Computer Society, 1989. 173

[GR09] Thomas Gibson-Robinson. On the refinement-closure of information flow properties. Undergraduate project report submitted to the Oxford University Computing Laboratory, 2009. 102

[Gro05] Alex David Groce. *Error Explanation and Fault Localization with Distance Metrics*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, 2005. 163, 195, 196

[Hal04] Ned Hall. Two concepts of causation. In *Causation and Counterfactuals*, chapter 9. MIT Press, 2004. 178, 196

[Har85] Norman Hardy. The KeyKOS architecture. *Operating Systems Review*, 19(4):8–25, 1985. 15, 71

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985. 2

[Hol03] Gerard J. Holzmann. *The SPIN model checker: Primer and reference manual*. Addison-Wesley, 2003. 204

[HP03] Mark Hopkins and Judea Pearl. Clarifying the usage of structural models for commonsense causal reasoning. In *Proceedings of the AAAI Spring Symposium on Logical Foundations of Commonsense Reasoning*, 2003. 195, 196

[HP05] J. Y. Halpern and J. Pearl. Causes and explanations: A structural-model approach. Part I: Causes. *British Journal for the Philosophy of Science*, 56(4):843–888, 2005. 195, 196

[HP07] Mark Hopkins and Judea Pearl. Causality and counterfactuals in the Situation Calculus. *Journal of Logic and Computation*, 17(5):939–953, 2007. 195, 196

[HR06] Jian Huang and A. W. Roscoe. Extending noninterference properties to the timed world. In *Proceedings of the 2006 ACM symposium on Applied computing (SAC 2006)*, pages 376–383. ACM, 2006. 208

[HRU76] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8):461–471, 1976. 50

[HY86] J. Thomas Haigh and William D. Young. Extending the noninference model of MLS for SAT. In *Proceedings of the 1986 IEEE Symposium on Security and Privacy (SP '86)*, pages 232–239. IEEE Computer Society Press, 1986. 99, 103, 104

[IR05] Y. Isobe and M. Roggenbach. A generic theorem prover of CSP refinement. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, page 108. Springer Verlag, 2005. 206

[IR06] Yoshinao Isobe and Markus Roggenbach. A complete axiomatic semantics for the CSP stable-failures model. In *Proceedings of the 17th International Conference on Concurrency Theory (CONCUR '06)*, volume 4137 of *Lecture Notes in Computer Science*, pages 158–172. Springer, 2006. 206

[IR08] Y. Isobe and M. Roggenbach. CSP-Prover: A proof tool for the verification of scalable concurrent systems. *Journal of Computer Software, Japan Society for Software Science and Technology (JSSST)*, 25(4):85–92, 2008. 206

[JSV05] Yves Jaradin, Fred Spiessens, and Peter Van Roy. SCOLL: A language for safe capability based collaboration. Research

Report INFO-2005-10, Université catholique de Louvain, 2005. 2

[Kar88] Paul Ashley Karger. *Improving Security and Performance for Capability Systems*. PhD thesis, Computer Laboratory, University of Cambridge, 1988. Also published as University of Cambridge Computing Laboratory Technical Report No. 149. 173

[Kin94] Ekkart Kindler. Safety and liveness properties: A survey. *EATCS-Bulletin*, 53:268–272, June 1994. 129

[Kle08] Eldar Kleiner. *A Web Services Security Study using Casper and FDR*. D.Phil. thesis, Oxford University Computing Laboratory, 2008. 88, 208

[KN06] Eldar Kleiner and Tom Newcomb. Using CSP to decide safety problems for access control policies. Research Report RR-06-04, Oxford University Computing Laboratory, University of Oxford, January 2006. 3

[Kos08] Matej Košík. Taming of Pict. In *Proceedings of the 34th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2008)*, volume 4910 of *Lecture Notes in Computer Science*, pages 610–621, 2008. 15

[Kos09] Matej Košík. *Toward Robust Software Systems Composed from Defensively Correct Components*. PhD thesis proposal, Slovak University of Technology in Bratislava, 2009. Available at: `http://www.altair.sk/mediawiki/upload/2/22/Kosik-thesis.pdf`. 71

[KT09] Maxwell Krohn and Eran Tromer. Non-interference for a practical DIFC-based operating system. In *Proceedings of the 2009 IEEE Symposium on Security and Privacy (SP '09)*, pages 61–76, 2009. 3

[Lam73] Butler Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973. 30, 173

[Lam77] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, March 1977. 30, 128, 131, 181

[Lam00] Leslie Lamport. Fairness and hyperfairness. *Distributed Computing*, 13(4):239–245, 2000. 131, 132

[Lan09] Charles Landau. CapROS: The Capability-based Reliable Operating System, 2009. Available at: `http://www.capros.org`. 15

[Laz99] Ranko S. Lazić. *A Semantic Study of Data Independence with Applications to Model Checking*. D.Phil. thesis, Oxford University Computing Laboratory, 1999. 3, 4, 27, 54, 65, 70, 120, 122

[LBK97] Rom Langerak, Ed Brinksma, and Joost-Pieter Katoen. Causal ambiguity and partial orders in event structures. In *Proceedings of the 8th International Conference on Concurrency Theory (CONCUR '97)*, pages 317–331. Springer-Verlag, 1997. 199

[Lew73] David Lewis. Causation. *Journal of Philosophy*, 70(17):556–567, 1973. 164, 195

[Liu09] Yang Liu. *Model Checking Concurrent and Real-Time Systems: The PAT Approach*. PhD thesis, National University of Singapore, 2009. Draft available at: `http://www.comp.nus.edu.sg/~liuyang/thesis/thesis.pdf`. 5, 133, 134, 153, 157, 158, 160, 204, 205

[LLO+09] Michael Leuschel, Marisa Llorens, Javier Oliver, Josep Silva, and Salvador Tamarit. The MEB and CEB static analysis for CSP specifications. In *Revised Selected Papers from Proceedings of the 18th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2008)*, volume 5438 of *Lecture Notes in Computer Science*, pages 103–118. Springer-Verlag, 2009. 198

[Low96] Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '96)*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer-Verlag, 1996. 3

[Low98] Gavin Lowe. Casper: A compiler for the analysis of security protocols. *Journal of computer security*, 6(1):53–84, 1998. 208

[Low07] Gavin Lowe. On information flow and refinement-closure. In *Proceedings of the 7th International Workshop on Issues in the Theory of Security (WITS '07)*, 2007. 13, 93, 94, 95, 99, 100

[Low08] Gavin Lowe. Specification of communicating processes: temporal logic versus refusals-based refinement. *Formal Aspects of Computing*, 20(3):277–294, 2008. 128, 129, 130, 132, 134, 135, 136, 137, 157, 158, 203

[Low09] Gavin Lowe. 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)*, volume 250 of *Electronic Notes in Theoretical Computer Science*, pages 153–170, 2009. 99, 110, 111

[LPS81] Daniel J. Lehmann, Amir Pnueli, and Jonathan Stavi. Impartiality, justice and fairness: The ethics of concurrent termination. In *Proceedings of the 8th Colloquium on Automata, Languages and Programming (ICALP 1981)*, volume 115 of *Lecture Notes in Computer Science*, pages 264–277. Springer-Verlag, 1981. 130

[LR99] Ranko Lazić and A. W. Roscoe. Data independence with generalised predicate symbols. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, pages 319–325. CSREA Press, June 1999. 89

[LS77] R. J. Lipton and L. Snyder. A linear time algorithm for deciding subject security. *Journal of the ACM*, 24(3):455–464, 1977. 51

[LSM+98] Peter A. Loscocco, Stephen D. Smalley, Patrick A. Muckelbauer, Ruth C. Taylor, S. Jeff Turner, and John F. Farrell. The inevitability of failure: The flawed assumption of security in modern computing environments. In *Proceedings of the 21st National Information Systems Security Conference*, pages 303–314, 1998. 173

[MG08] Toby Murray and Duncan Grove. Non-delegatable authorities in capability systems. *Journal of Computer Security*, 16(6):743–759, December 2008. 161, 172, 173, 174, 175

[Mil00] Mark Miller. Grant Matcher Puzzle, 2000. Available at: `http://www.erights.org/elib/equality/grant-matcher/index.html`. 90

[Mil06] Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control.* PhD thesis, Johns Hopkins University, 2006. 1, 5, 14, 15, 30, 72, 94, 115, 161, 162, 184, 185, 186

[ML07] Toby Murray and Gavin 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)*, pages 113–130, 2007. iv, 196, 197

[ML09a] Toby Murray and Gavin Lowe. Analysing the information flow properties of object-capability patterns. In *Proceedings of the Sixth International Workshop on Formal Aspects of Security and Trust (FAST 2009)*, 2009. To appear. iii, iv, 125

[ML09b] Toby Murray and Gavin Lowe. On refinement-closed security properties and nondeterministic compositions. In *Proceedings of the Eighth International Workshop on Automated Verification of Critical Systems (AVoCS '08)*, volume 250 of *Electronic Notes in Theoretical Computer Science*, pages 49–68, 2009. iii, iv, 99

[Mor73] James H. Morris, Jr. Protection in programming languages. *Communications of the ACM*, 16(1):15–21, 1973. 30, 40

[MSL+07] Mark S. Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. Caja: Safe active content in sanitized JavaScript (draft), 2007. Draft from December 31, 2007, available at: `http://google-caja.googlecode.com/files/caja-2007.pdf`. 41, 49

[MSL+08] Mark S. Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. Caja: Safe active content in sanitized JavaScript (draft), 2008. Draft from June 7, 2008, available at: `http://google-caja.googlecode.com/files/caja-spec-2008-06-07.pdf`. 1, 15, 50

[Muk93] Abida Mukarram. *A Refusal Testing Model for CSP*. D.Phil. thesis, University of Oxford, 1993. 135

[Mur08] Toby Murray. 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)*, pages 177–194, 2008. 24, 46, 52, 89

[MW08] Adrian Matthew Mettler and David Wagner. The Joe-E language specification, version 1.0. Technical Report EECS-2008-91, University of California, Berkeley, August 2008. 15

[MWC10] Adrian Mettler, David Wagner, and Tyler Close. Joe-E: A security-oriented subset of Java. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS 2010)*, 2010. To appear. Available at: `http://www.eecs.berkeley.edu/~daw/papers/joe-e-ndss10.pdf`. 1, 15, 162, 184, 189

[Nor98] Michael Norrish. *C Formalised in HOL*. PhD thesis, Computer Laboratory, University of Cambridge, 1998. Also published as University of Cambridge Computing Laboratory Technical Report No. 453. 208

[Par03] James D. Park. Causes and explanations revisited. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 154–162. Morgan Kaufman, 2003. 178

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994. 206

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57, November 1977. 129

[Puh03] Antti Puhakka. Using fairness in process-algebraic verification. Technical Report 24, Institute of Software Systems, Tampere University of Technology, 2003. 129, 134, 157, 158

[Puh05] Antti Puhakka. Using fairness constraints in process-algebraic verification. In *Proceedings of the Second International Colloquium on Theoretical Aspects of Computing (ICTAC 2005)*, volume 3722 of *Lecture Notes in Computer Science*, pages 546–561. Springer, 2005. 129, 131, 134, 157, 158

[PV01] Antti Puhakka and Antti Valmari. Liveness and fairness in process-algebraic verification. In *Proceedings of the 12th International Conference on Concurrency Theory (CONCUR '01)*, volume 2154 of *Lecture Notes in Computer Science*, pages 202–217. Springer, 2001. 129, 131, 132, 134, 157, 158

[QVDC06] L. Quesada, P. Van Roy, Y. Deville, and R. Collet. Using dominators for solving constrained path problems. In *Proceedings of the 8th International Symposium on Practical Aspects of Declarative Languages (PADL '06)*, volume 3819 of *Lecture Notes in Computer Science*, page 73. Springer, 2006. 197

[RA03] Peter Ryan and Ragni Ryvold Arnesen. A process algebraic approach to security policies. In *Proceedings of the Sixteenth International Conference on Data and Applications Security (DBSec '02)*, volume 256 of *IFIP Conference Proceedings*, pages 301–312. Kluwer, 2003. 3

[Raj89] Susan A. Rajunas. The KeyKOS/KeySAFE system design. Technical Report SEC009-01, Key Logic, Inc., March 1989. See `http://www.cis.upenn.edu/~KeyKOS`. 72

[RB99] A. W. Roscoe and P. J. Broadfoot. Proving security protocols with model checkers by data independence techniques. *Journal of Computer Security*, 7(2-3):147–190, 1999. 65, 66, 83, 87, 88, 89

[Red74] David D. Redell. *Naming and Protection in Extendable Operating Systems*. PhD thesis, University of California, Berkeley, 1974. Published as Project MAC Technical Report TR-140, Massachusetts Institute of Technology. 73, 175

[RG99] A. W. Roscoe and M. H. Goldsmith. What is intransitive noninterference? In *Proceedings of the 12th IEEE Computer Security Foundations Workshop (CSFW '99)*, page 228. IEEE Computer Society, 1999. 99, 100, 103, 104

[RGG+95] A. W. Roscoe, Paul H. B. Gardiner, M. H. Goldsmith, J. R. Hulance, D. M. Jackson, and J. B. Scattergood. Hierarchical compression for model-checking CSP or how to check 10^{20} dining philosophers for deadlock. In *Proceedings of the First International Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS '95)*, pages 133–152, London, UK, 1995. Springer-Verlag. 2, 19

[RH07] A. W. Roscoe and David Hopkins. SVA, a tool for analysing shared-variable programs. In *Proceedings of AVoCS 2007*, pages 177–183, 2007. Available at: `http://web.comlab.ox.ac.uk/oucl/work/bill.roscoe/publications/119.pdf`. 208

[RL05] Gordon Thomas Rohrmair and Gavin Lowe. Using data-independence in the analysis of intrusion detection systems. *Theoretical Computer Science*, 340(1):82–101, 2005. 3, 88

[Ros94] A. W. Roscoe. Model-checking CSP. In A. W. Roscoe, editor, *A Classical Mind: Essays in Honour of C. A. R. Hoare*, pages 353–378. Prentice-Hall, 1994. 2

[Ros95] A. W. Roscoe. CSP and determinism in security modelling. In *Proceedings of the 1995 IEEE Symposium on Security and Privacy (SP '95)*, page 114. IEEE Computer Society, 1995. 93

[Ros97] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, Upper Saddle River, NJ, USA, 1997. Available at: `http://www.comlab.ox.ac.uk/people/bill.roscoe/publications/68b.pdf`. 2, 6, 10, 27, 65, 70, 95, 98, 99, 106, 136, 155, 156, 200, 205

[Ros01] A. W. Roscoe. Compiling shared variable programs into CSP. In *Proceedings of the 2001 PROGRESS Workshop*,

2001. Available at: http://web.comlab.ox.ac.uk/oucl/work/bill.roscoe/publications/82.ps. 5, 153, 158, 159, 160, 204, 207, 208

[Ros05] A. W. Roscoe. On the expressive power of CSP refinement. *Formal Aspects of Computing*, 17(2):93–112, August 2005. 3, 98, 110, 141, 142, 157, 190

[Ros08] A. W. Roscoe. The three platonic models of divergence-strict CSP. In *Proceedings of the 5th International Colloquium on Theoretical Aspects of Computing (ICTAC 2008)*, volume 5160 of *Lecture Notes in Computer Science*, pages 23–49. Springer-Verlag, 2008. 13, 131, 140, 141

[Ros09] A. W. Roscoe. Revivals, stuckness and the hierarchy of CSP models. *Journal of Logic and Algebraic Programming*, 78(3):163–190, 2009. 134, 140, 206

[RRS05] J. N. Reed, A. W. Roscoe, and J. E. Sinclair. Machine-verifiable responsiveness. In *Proceedings of the 5th International Workshop on Automated Verification of Critical Systems (AVoCS 2005)*, volume 145 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2005. 199

[RS01] P. Y. A. Ryan and S. A. Schneider. Process algebra and noninterference. *Journal of Computer Security*, 9(1/2):75–103, 2001. 93

[RSG+00] Peter Ryan, Steve Schneider, Michael Goldsmith, Gavin Lowe, and Bill Roscoe. *Modelling and Analysis of Security Protocols: the CSP Approach*. Addison-Wesley, 2000. 3, 87

[RSR04] J. N. Reed, J. E. Sinclair, and A. W. Roscoe. Responsiveness of interoperating components. *Formal Aspects of Computing*, 16(4):394–411, 2004. 199

[Rus89] John Rushby. Formal methods and critical systems in the real world. In *Formal Methods for Trustworthy Computer Systems (FM89)*, pages 121–125. Springer-Verlag Workshops in Computing, 1989. vi

[Rus92] John Rushby. Noninterference, transitivity and channel-control security policies. Technical Report CSL-92-02, SRI International, December 1992. 99, 103, 104

[RWW94] A. W. Roscoe, J. C. P. Woodcock, and L. Wulf. Noninterference through determinism. In *Proceedings of the Third European Symposium on Research in Computer Security (ESORICS '94)*, pages 33–53. Springer-Verlag, 1994. 93

[Rya91] P. Y. A. Ryan. A CSP formulation of non-interference and unwinding. *Cipher*, pages 19–30, Winter 1991. 93, 99, 100

[SA07] Jonathan S. Shapiro and Jonathan W. Adams. The Coyotos Microkernel Specification - Version 0.6+, September 2007. Available at: `http://www.coyotos.org/docs/ukernel/spec.html`. 15, 16, 147

[SDN+04] Jonathan Shapiro, Michael Scott Doerrie, Eric Northup, Swaroop Sridhar, and Mark Miller. Towards a verified, general-purpose operating system kernel. In *Proceedings of the 1st NICTA Workshop on Operating System Verification*, October 2004. 15

[Sea07] Mark Seaborn. Plash: tools for practical least privilege, 2007. Available at: `http://plash.beasts.org`. 15

[Sha99] Johnathan Strauss Shapiro. *EROS: A Capability System*. PhD thesis, University of Pennsylvania, 1999. 71

[SJV05] Fred Spiessens, Yves Jaradin, and Peter Van Roy. SCOLL and SCOLLAR: Safe collaboration based on partial trust. Research Report INFO-2005-12, Université catholique de Louvain, 2005. 2

[SLDW08] Jun Sun, Yang Liu, Jin Song Dong, and Hai H. Wang. Specifying and verifying event-based fairness enhanced systems. In *Formal Methods and Software Engineering, Proceedings of the 10th International Conference on Formal Engineering Methods (ICFEM '08)*, pages 5–24. Springer-Verlag, 2008. 5, 129, 131, 132, 134, 153, 157, 158, 160, 204

[SMS05] Marc Stiegler, Mark S. Miller, and Terry Stanley. 72 hours to DonutLab: A PlanetLab with no center. Technical Report HPL-2005-5, HP Laboratories, Polo Alto, 2005. 30

[Spi06] Fred Spiessens. Some examples of non-trivial causal influence (authority), 2006. Available at: `http://www.scoll.evoluware.eu/causalityinwrongdirection.pdf`. 177, 178

[Spi07] Alfred Spiessens. *Patterns of Safe Collaboration*. PhD thesis, Université catholique de Louvain, Louvain-la-Neuve, Belgium, February 2007. 2, 4, 5, 19, 51, 54, 55, 70, 71, 72, 79, 86, 88, 90, 91, 93, 125, 156, 197

[SQV06] Fred Spiessens, Luis Quesada, and Peter Van Roy. Confinement analysis with graph reachability constraints. In *Proceedings of the 1st Workshop on Constraints in Software Testing, Verification and Analysis*, pages 58–72, 2006. 197

[SRI09] D. Gift Samuel, Markus Roggenbach, and Yoshinao Isobe. The stable revivals model in CSP-Prover. In *Proceedings of the Eighth International Workshop on Automated Verification of Critical Systems (AVoCS '08)*, volume 250 of *Electronic Notes in Theoretical Computer Science*, pages 119–134. Elsevier Science Publishers B. V., 2009. 206

[SS75] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1208–1308, September 1975. 1

[SSF99] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: A fast capability system. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles (SOSP '99)*, pages 170–185, 1999. 15

[Sti04] Marc Stiegler. A PictureBook of Secure Cooperation, 2004. Presentation. Available at: `http://erights.org/talks/efun/SecurityPictureBook.pdf`. 40

[Sti06] Marc Stiegler. The E Language in a Walnut. Available at: `http://www.skyhunter.com/marcs/ewalnut.html`, 2006. 31, 90

[Sti07] Marc Stiegler. Emily: A high performance language for enabling secure cooperation. In *Proceedings of the Fifth International Conference on Creating, Connecting and Collaborating through Computing (C5 '07)*, pages 163–169, 2007. 15

[SV03] Peter Sewell and Jan Vitek. Secure composition of untrusted code: box π, wrappers, and causality types. *Journal of Computer Security*, 11(2):135–187, 2003. 198

[SV05] Fred Spiessens and Peter Van Roy. A practical formal model for safety analysis in capability-based systems. In *Revised Selected Papers of the 2005 International Symposium on Trustworthy Global Computing (TGC '05)*, volume 3705 of *Lecture Notes in Computer Science*, pages 248–278. Springer, 2005. 2, 51, 125

[SW00] Jonathan S. Shapiro and Sam Weber. Verifying the EROS confinement mechanism. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy (SP '00)*, page 166. IEEE Computer Society, 2000. 51

[TMHK95] E. Dean Tribble, Mark S. Miller, Norm Hardy, and David Krieger. Joule: Distributed application foundations. Technical Report ADd03.4P, Agorics Inc., Los Altos, December 1995. Available at: `http://www.erights.org/history/joule/index.html`. 39

[Tri06] Dean Tribble. EQ not required by object-cap model, December 2006. E-mail communication to the `cap-talk` mailing list, available at: `http://www.eros-os.org/pipermail/cap-talk/2006-December/006246.html`. 70

[vdM08] Ron van der Meyden. What, indeed, is intransitive noninterference? (extended abstract). In *Proceedings of the 12th European Symposium on Research in Computer Security (ESORICS '07)*, volume 4734 of *Lecture Notes in Computer Science*, 2008. 103, 104

[vdM09] Ron van der Meyden. Architectural refinement and notions of intransitive noninterference. In *Proceedings of the First International Symposium on Engineering Secure Software and Systems (ESSoS 2009)*, volume 5429 of *Lecture Notes in Computer Science*, pages 60–74. Springer, 2009. 89, 125

[vGV03] Rob van Glabbeek and Frits Vaandrager. Bundle event structures and CCSP. In *Proceedings of the 14th International Conference on Concurrency Theory (CONCUR '03)*, pages 57–71. Springer-Verlag, 2003. 199

[VH04] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, 2004. 40

[VVK05] Hagen Völzer, Daniele Varacca, and Ekkart Kindler. Defining fairness. In *Proceedings of the 16th International Conference on Concurrency Theory (CONCUR '05)*, volume 3653 of *Lecture Notes in Computer Science*, pages 458–472. Springer, 2005. 130, 138, 181

[VW86] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First IEEE Symposium on Logic in Computer Science (LICS '86)*, pages 322–331, 1986. 204

[Wag08a] David Wagner. An attack on a mint, March 2008. E-mail communication to the `e-lang` mailing list, available at: `http://www.eros-os.org/pipermail/e-lang/2008-March/012516.html`. 38

[Wag08b] David Wagner. A broken brand?, March 2008. E-mail communication to the `e-lang` mailing list, available at: `http://www.eros-os.org/pipermail/e-lang/2008-March/012508.html`. 38, 49

[Wat09] Robert Watson. Capsicum - an incremental approach to capability systems, 2009. Available at: `http://www.cl.cam.ac.uk/research/security/capsicum/`. 15

[WBDF97] Dan S. Wallach, Dirk Balfanz, Drew Dean, and Edward W. Felten. Extensible security architectures for Java. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 116–128. ACM, 1997. 173

[Win89] Glynn Winskel. An introduction to event structures. In *REX School of Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, pages 364–397. Springer-Verlag, 1989. 199

[Wol86] Pierre Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages (POPL '86)*, pages 184–193, New York, NY, USA, 1986. ACM. 26

[WVS83] Pierre Wolper, Moshe Y. Vardi, and A. Prasad Sistla. Reasoning about infinite computation paths. In *Proceedings of the 24th Annual Symposium on Foundations of Computer Science (SFCS '83)*, pages 185–194. IEEE Computer Society, 1983. 204

[Yee99] Ka-Ping Yee. A stab at a sealer in E, 1999. E-mail communication to the `e-lang` mailing list, available at: `http://www.eros-os.org/pipermail/e-lang/1999-November/002983.html`. 40, 146

A Subsidiary Results

Lemma A.0.1. For any sets F, C and D, for all $F' \subseteq F$, $C' \subseteq C$ and $D' \subseteq D$,

$$\mathit{Untrusted}_{OS}(F, C, D) \sqsubseteq_{FD} \mathit{Untrusted}_{OS}(F', C', D'),$$

and

$$\mathit{Untrusted}_{lang}(F, C, D) \sqsubseteq_{FD} \mathit{Untrusted}_{lang}(F', C', D),$$

and

$$\mathit{UntrustedActive}_{lang}(F, C, D) \sqsubseteq_{FD} \mathit{UntrustedActive}_{lang}(F', C', D).$$

Proof. By inspection. □

The following lemmas state that, as an untrusted object interacts with its environment, its range of possible behaviours can only increase over time. As such, we say that its behaviour increases *monotonically* with its interactions with other objects in any system.

Lemma A.0.2. Let A_u denote the set of events that appear in a sequence u. Then for any sets F, C and D and traces s and t from $\mathit{traces}(\mathit{Untrusted}_{OS}(F, C, D))$, if $A_t \subseteq A_s$ then

$$\mathit{Untrusted}_{OS}(F, C, D) \,/\, s \sqsubseteq_{FD} \mathit{Untrusted}_{OS}(F, C, D) \,/\, t.$$

Proof. Straightforward. □

Lemma A.0.3. Let A_u denote the set of events that appear in a sequence u and fix U to mean either $\mathit{UntrustedActive}_{lang}$ or $\mathit{Untrusted}_{lang}$. Then for any sets F, C and D, and traces s and t from $\mathit{traces}(U(F, C, D))$, if $A_t \subseteq A_s$ and $U(F, C, D)$ is active after performing s if and only if it is active after performing t, then

$$U(F, C, D) \,/\, s \sqsubseteq_{FD} U(F, C, D) \,/\, t.$$

Proof. Straightforward.

□

Lemma A.0.4 (Trace-equivalent deterministic refinements)**.** Every process P with finite alphabet 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, $T_Q = \{s \mid (s, X) \in F_Q\}$.

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

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 $s \frown \langle e \rangle \in T_Q$. 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 A.0.5. 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 2.1) 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 $v \frown \langle a \rangle \notin T_Q$ 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 $s \frown \langle e \rangle \in \mathit{traces}(P) \wedge e \in X \cup \{a\}$. Since $(v, X) \in F_Q$ by assumption, we have that $\forall\, e' \in X \bullet v \frown \langle e' \rangle \notin \mathit{traces}(P)$ or else this failure would have been removed. Hence, it must be the case that $e = a$, *i.e.* that $v \frown \langle a \rangle \in \mathit{traces}(P)$, equivalently $v \frown \langle a \rangle \in T_Q$, which clearly contradicts our assumptions. Hence this axiom must be satisfied. □

Lemma A.0.6. Given a divergence- and ✓-free process P with finite alphabet, and some failure $(s, Y) \in \mathit{failures}(P)$, there exists a divergence-free refinement, Q, of P that does not have the traces $\{s \frown \langle e \rangle \mid e \in Y\}$ and whose failures are defined as follows:

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

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 $traces(P)$ 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 $v \frown \langle a \rangle \notin T_Q$. Then $(v, X) \in failures(P)$. We perform a case analysis.

- Case $v \frown \langle a \rangle \neq s \frown \langle e \rangle$ for any $e \in Y$. Then, $v \frown \langle a \rangle$ is not a trace that has been removed from P. So, because $v \frown \langle a \rangle \notin T_Q$, $v \frown \langle a \rangle \notin traces(P)$ either. Then, because P satisfies **F3**, $(v, X \cup \{a\}) \in failures(P)$. And hence $(v, X \cup \{a\}) \in F_Q$, by construction.

- Case $v = s \wedge a = e$ for some $e \in Y$. Then $(s, X) \in F_Q$ and $s \frown \langle e \rangle \notin T_Q$. Recall that $(s, Y) \in failures(P)$. We must have, then, that $(s, X \cup Y) \in failures(P)$ since if it were not, (s, X) would have been removed and hence not be present in F_Q. Hence, because P satisfies Axiom **F2** and $e \in Y$, $(s, X \cup \{e\}) \in failures(P)$. Then $(s, X \cup \{e\}) \in F_Q$ by construction, since only those failures (s, Z) for which $(s, Z \cup Y) \notin failures(P)$ are removed. Because $v = s \wedge a = e$, then $(v, X \cup \{a\}) \in F_Q$ as required.

□

Theorem A.0.7. For any effect E, there exists a safety effect E_S and a liveness effect E_L such that

$$E = E_S \cap E_L$$

Proof. This proof is a direct adaptation of that for Theorem 5 from [CS10, Appendix D]. Given an effect E, our strategy is to construct a safety effect E_S that contains E as a subset. We also construct a liveness effect E_L that contains E and then show that their intersection is E.

To construct E_S, we define the safety effect $Safe(E)$ for which the bad thing that it asserts cannot happen is a finite set of executions that cannot be extended so as to satisfy E. So $Safe(E)$ contains systems all of whose observations (*i.e.* finite sets of finite refusal-traces) can be extended to satisfy E.

$$Safe(E) = \left\{ \mathcal{C}[\![P]\!] \;\middle|\; P \in \mathbf{CSP} \wedge \begin{array}{l} \forall M \bullet |M| \in \mathbb{N} \wedge M \subseteq \mathcal{R}[\![Sys]\!] \Rightarrow \\ (\exists P' \in \mathbf{CSP} \bullet M \subseteq \mathcal{R}[\![P']\!] \wedge \mathcal{C}[\![P']\!] \in E) \end{array} \right\}$$

We show that $Safe(E)$ is a safety effect. Consider any system $Sys \in \mathbf{CSP}$ for which $\mathcal{C}[\![Sys]\!] \notin Safe(E)$. Then there exists some finite set M where

$M \subseteq \mathcal{R}[\![Sys]\!]$ and

$$\forall\, Sys' \in \mathbf{CSP} \bullet M \subseteq \mathcal{R}[\![Sys']\!] \Rightarrow \mathcal{C}[\![Sys']\!] \notin E. \tag{A.1}$$

Because $M \subseteq \mathcal{R}[\![Sys]\!]$, we have that $\mathcal{C}[\![Sys]\!] \notin E$. Hence, $\mathcal{C}[\![Sys]\!] \notin \mathit{Safe}(E) \Rightarrow \mathcal{C}[\![Sys]\!] \notin E$ and so $(\exists\, Sys' \in \mathbf{CSP} \bullet M \subseteq \mathcal{R}[\![Sys']\!] \wedge \mathcal{C}[\![Sys']\!] \notin \mathit{Safe}(E)) \Rightarrow (\exists\, Sys' \in \mathbf{CSP} \bullet M \subseteq \mathcal{R}[\![Sys']\!] \wedge \mathcal{C}[\![Sys']\!] \notin E)$ since in both cases we can set $Sys' = Sys$. It follows that

$$\begin{array}{l}(\forall\, Sys' \in \mathbf{CSP} \bullet M \subseteq \mathcal{R}[\![Sys']\!] \Rightarrow \mathcal{C}[\![Sys']\!] \notin E) \Rightarrow \\ \quad (\forall\, Sys' \in \mathbf{CSP} \bullet M \subseteq \mathcal{R}[\![Sys']\!] \Rightarrow \mathcal{C}[\![Sys']\!] \notin \mathit{Safe}(E)).\end{array} \tag{A.2}$$

So, combining Equations A.1 and A.2,

$$\forall\, Sys' \in \mathbf{CSP} \bullet M \subseteq \mathcal{R}[\![Sys']\!] \Rightarrow \mathcal{C}[\![Sys']\!] \notin \mathit{Safe}(E).$$

Hence $\mathit{Safe}(E)$ satisfies Equation 7.12 and so is a safety effect.

To construct E_L we define the liveness effect $\mathit{Live}(E)$ that asserts that it's always possible either to satisfy E or for satisfying E to become impossible due to $\mathit{Safe}(E)$ having been violated. Formally

$$\mathit{Live}(E) = E \cup \overline{\mathit{Safe}(E)}.$$

We show that $\mathit{Live}(E)$ is a liveness effect. Consider any partial observation M where $|M| \in \mathbb{N} \wedge M \subseteq \mathcal{R}[\![P]\!] \cap \mathit{PRT}$ for some process $P \in \mathbf{CSP}$. Suppose there exists some process $P' \in \mathbf{CSP}$ for which $M \subseteq \mathcal{R}[\![P']\!]$ and $\mathcal{C}[\![P']\!] \in E$. Then $\mathcal{C}[\![P']\!] \in \mathit{Live}(E)$ as required. Otherwise, we must have that for all processes $P' \in \mathbf{CSP}$, if $M \subseteq \mathcal{R}[\![P']\!]$ then $\mathcal{C}[\![P']\!] \notin E$. Let P' be an arbitrary process such that $M \subseteq \mathcal{R}[\![P']\!]$. Then following the same reasoning that led to Equation A.1, $C[P'] \notin \mathit{Safe}(E)$ so $\mathcal{C}[\![P']\!] \in \overline{\mathit{Safe}(E)}$. So $\mathcal{C}[\![P']\!] \in \mathit{Live}(E)$ again. Hence $\mathit{Live}(E)$ satisfies Equation 7.13 and so is a liveness effect.

We now show that $E \subseteq \mathit{Safe}(E)$. Consider any process $P \in \mathbf{CSP}$ for which $\mathcal{C}[\![P]\!] \in E$. Then for any finite $M \subseteq \mathcal{R}[\![P]\!]$ there exists a process P' such that $M \subseteq \mathcal{R}[\![P']\!]$ and $\mathcal{C}[\![P']\!] \in E$, namely P itself. So $\mathcal{C}[\![P]\!] \in \mathit{Safe}(E)$. Hence, $\mathit{Safe}(E) = E \cup \mathit{Safe}(E)$.

Let $E_S = \mathit{Safe}(E)$ and $E_L = \mathit{Live}(P)$. Then

$$\begin{aligned} E_S \cap E_L &= \mathit{Safe}(E) \cap \mathit{Live}(E) \\ &= (E \cup \mathit{Safe}(E)) \cap (E \cup \overline{\mathit{Safe}(E)}) \\ &= E \cap (\mathit{Safe}(E) \cup \overline{\mathit{Safe}(E)}) \\ &= E \cap \{\mathcal{C}[\![P]\!] \mid P \in \mathbf{CSP}\} \\ &= E. \end{aligned}$$

□