# The Formal Development of Secure Systems

John Graham–Cumming
Lady Margaret Hall, Oxford

D.Phil. Thesis, Trinity Term 1992

# Abstract

The Formal Development of Secure Systems

John Graham–Cumming, Lady Margaret Hall
D.Phil. Thesis, Trinity Term 1992

In secure systems the availability, integrity and confidentiality of information are important concerns. Such systems are special because conventional formal techniques for specification and development prove inadequate when security is taken into account. It is possible for a conventionally specified secure system to exhibit security flaws, even after a formal development.

This thesis addresses that problem by propounding a theory (with applications) for the formal specification and development of secure systems.

Here a system (and its users) are CSP processes. The users are defined by specifying their interfaces to the system. Interactions at the interface are the only way in which a user is able to determine the state of the system. We say that information flows from one interface to another if changes at the first alter the interactions possible at the second. A security specification defines when (and how) one interface may affect another.

Most of our results concern the common security property called non–interference. It guarantees that no information flows from one user to another, i.e. that interactions at one interface do not affect interactions at another. We determine when one system can be replaced by another preserving its non–interference properties; we call that *secure replacement*. The properties of secure replacement are discussed and we investigate the relationship between maintaining functionality and maintaining security.

A collection of laws is presented which show how non–interfering systems can be constructed from operators of CSP. A case study demonstrates that our theory is strong enough to encompass the standard techniques for implementing secure systems (e.g. access control).

We show how to extend our work to encompass systems exhibiting timed behaviours. A suitable definition of timed non–interference is given and laws concerning it are proved.

A summary of related work is given. An appendix relates state–based refinement to secure replacement, and a further appendix shows how the laws of non–interference can be extended to more general security policies. A glossary of the terminology of secure systems forms the final appendix.

# Acknowledgements

The greatest thanks go to Jeff Sanders, my supervisor, who, after the experience of teaching me for three years as an undergraduate, agreed to supervise this work. Without his encouragement and guidance I would not have stayed the course of the last three years.

Angela Cowdery's friendship and support have been invaluable and I am deeply indebted to her.

Many thanks to Sue, Ian, Pauline Maclean, Pauline Shepheard and Tommy Wareing.

Thanks to Jeremy Jacob, Clare Martin and Steve Schneider for useful discussions concerning security and formal methods, and for proof–reading.

The OUP books *Hart's Rules for Compositors and Readers*, *The Oxford Dictionary for Writers and Editors* and *The Printing of Mathematics* proved invaluable in taming the LaTeX document preparation system; Jeff Sanders, and David Shirt of the Oxford Dictionary helped to improve my use of English.

Thanks are due to SERC for their financial support through SERC Quota Award 89314781.

*Dedicated to Gilbert Franklin*

# Contents


**1 Introduction** **1**
1.1 Secure Systems . . . . . . . . . . . . . . . . . . . . . . . . . 1
1.2 Structure of Thesis . . . . . . . . . . . . . . . . . . . . . . . 3

**2 Background** **5**
2.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
2.2 Relations and Orders . . . . . . . . . . . . . . . . . . . . . . 5
2.3 CSP . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
2.4 Refinement in CSP . . . . . . . . . . . . . . . . . . . . . . . 11

**3 Non–interference** **13**
3.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
3.2 Information Flow . . . . . . . . . . . . . . . . . . . . . . . . 13
3.2.1 Principle Security Properties . . . . . . . . . . . . . 14
3.2.2 Equivalent Traces . . . . . . . . . . . . . . . . . . . 15
3.3 Non–interference . . . . . . . . . . . . . . . . . . . . . . . . 21

**4 Development** **25**
4.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
4.2 Preserving Security by Refinement . . . . . . . . . . . . . . 25
4.2.1 Proving Security . . . . . . . . . . . . . . . . . . . . 26
4.2.2 Lack of Security . . . . . . . . . . . . . . . . . . . . 27
4.2.3 Secure Replacement . . . . . . . . . . . . . . . . . . 29
4.2.4 Refining Non–interference . . . . . . . . . . . . . . 33
4.3 Laws of CSP . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
4.3.1 Properties of $\approx_u^{\mathsf{S}}$ and $\equiv_u^A$ . . . . . . . . . . . . . . 34
4.3.2 Transaction Non–interference . . . . . . . . . . . . 37
4.3.3 Laws of Non–interference . . . . . . . . . . . . . . . 39

**5 Case Study** **51**
5.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
5.2 Specification . . . . . . . . . . . . . . . . . . . . . . . . . . . 52
5.3 Development . . . . . . . . . . . . . . . . . . . . . . . . . . . 55

5.4 occam Implementation . . . . . . 68
5.5 Possible Flaws . . . . . . 74
5.5.1 Errors of Translation . . . . . . 74
5.5.2 Security Breaches . . . . . . 75

**6 Timed Non–interference 79**
6.1 Introduction . . . . . . 79
6.2 Timed Security . . . . . . 79
6.2.1 Examples . . . . . . 82
6.2.2 Timed $v$–equivalence . . . . . . 84
6.2.3 Some Laws of Timed Non–interference . . . . . . 85

**7 Related Work 92**
7.1 Introduction . . . . . . 92
7.2 Access Control and Databases . . . . . . 92
7.3 Information Flow . . . . . . 94
7.3.1 Separability . . . . . . 94
7.3.2 Inference Functions . . . . . . 97
7.3.3 Deducibility . . . . . . 98
7.3.4 Other Information–Flow Approaches . . . . . . 99
7.4 Non–interference . . . . . . 100
7.4.1 Goguen and Meseguer's Non–interference . . . . . . 101
7.4.2 CSP Formulations of Non–interference . . . . . . 103
7.4.3 Restrictiveness and Correctability . . . . . . 106
7.4.4 Other Non–interference Results . . . . . . 108
7.5 Refinement of Security Policies . . . . . . 111

**8 Conclusion 114**
8.1 Review . . . . . . 114
8.2 Future Directions . . . . . . 115

**Bibliography 118**

**A Another Model of Computation 126**
A.1 Introduction . . . . . . 126
A.2 Abstract Data–types . . . . . . 127
A.2.1 The Garage Map . . . . . . 128
A.2.2 Refinement of an ADT . . . . . . 129
A.2.3 Secure Refinement . . . . . . 130
A.3 Example of a Secure Refinement . . . . . . 133

**B General Security Policies** **139**
B.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . 139
B.2 A Notation for Policy Specification . . . . . . . . . . . . 139
B.3 Laws of General Policies . . . . . . . . . . . . . . . . . . . . 142

**C Glossary** **147**

**Index of Symbols** **157**

# Chapter 1

# Introduction

## 1.1 Secure Systems

High–integrity computer systems are those that exhibit a great degree of reliability because they are used in critical situations. Examples of such systems are those having safety–critical properties (used, for example, in the actuation of the control surfaces of a passenger aircraft) and those with special constraints on the security of information stored in them.

This thesis considers secure systems. A secure system is one which is used to store information that is sensitive to a particular group. Examples are: hospital patients' record–keeping systems—where the confidentiality of the records is of great importance; banking systems—where the accuracy of financial transactions is paramount—and military systems—where not only must information be accurate and confidential, but the system must be robust so that commands and orders can be carried out efficiently (see [96] for further examples).

The systems we consider have multiple users. The security properties of the systems are defined in terms of the users. In turn, users are defined by their interfaces to the system. We say that *information flows* from one user to another if the interactions at the first user's interface are able to affect interactions at the other user's interface. That effect takes the form of altering the possible interactions, e.g. the possible commands (with their inputs) that may be performed, or the possible outputs that may be seen.

Each of the three properties mentioned in the previous paragraph (maintaining confidentiality, integrity and availability of the system) can be expressed as a property of the flow of information within a system. To maintain confidentiality information must be prevented from

flowing to unauthorised parties. To maintain integrity unauthorized information must not alter existing information. Availability can be seen as a flow of information by noting that whether or not a system is operating is a form of information—an observer's knowledge about the system is changed by knowing whether it is operating (see Section 3.2.1 for further details).

Frequently the security aspects of a computer system are realized through the use of *access control*. Mechanisms are introduced which control the access which users (called subjects) have to objects (files, devices, etc.). Access–control mechanisms provide a coarse method of controlling information flow, but they do not necessarily capture more subtle information flow. Denning has expressed it:

> Access controls regulate the accessing of objects, but not what subjects might do with the information contained in them. Many difficulties with information "leakage" arise not from defective access control, but from the lack of any policy about information flow. Flow controls are concerned with the right of **dissemination of information**, irrespective of what object holds the information; they specify valid channels along which information may flow. [22, page 265]

The problems mentioned by Denning occur because simply considering access control limits the designer of a secure system to a particular implementation technique and design. A more abstract approach is required to capture the flow of information. It is thus required to find an abstract characterization of security and suitable formal development methods so that secure systems that control dissemination of information can be constructed.

The problem of subtle forms of information flow (sometimes called flow through covert channels) is aggravated because of the problem of *trojan horses*. A trojan horse is a program (or process) that appears legitimate—and may perform a legitimate task—but which is able to use covert channels to communicate information in a manner not authorized by the system's security policy.

The presentation of suitable formal methods for the abstract specification and development of secure systems is the aim of this thesis.

Using such formal methods secure systems can be built and techniques, such as access control, can be incorporated in the implementation. The abstract specification and development will guarantee that the appropriate security property is present in the implementation; the developer is free to choose implementation designs as appropriate.

## 1.2 Structure of Thesis

Chapter 2 presents some of the theory necessary for the rest of the thesis. The theory of relations and orders is discussed briefly, as well as communicating processes (where the model of computation we use is described) and refinement.

Chapter 3 gives the definition of information flow in terms of the traces of a system, by showing when two traces are indistinguishable at the interface of a particular user. Non–interference is introduced: informally, a user is non–interfering with another if no information flows from the first user to the second. We show how interfaces are used to give the security properties of the system and use those interfaces to formally specify non–interference. With the formal specification of non–interference we can proceed to consider suitable formal development methods for non–interfering systems.

Chapter 4 describes techniques for refinement that preserve non–interference and presents a collection of laws for the algebraic development of non–interfering systems. Those laws can be used to decompose systems and prove that a systems exhibits non–interference. Together with the definition in Chapter 3 the laws form a development method for non–interfering systems.

Non–interference is used in Chapter 5 where a case study is undertaken to illustrate some of the techniques of Chapter 4 and to show how non–interference can be realized in a system using access control as the principle implementation technique. There, abstract formal methods capture the required security property (non–interference) and conventional access–control techniques are used in the implementation. Those techniques are only introduced, however, by appeal to the laws of Chapter 4.

Chapter 6 extends the definition of information flow and laws of non–interference to timed observations and examines some of the differences introduced by making timed observations of a system.

Chapter 7 gives a summary of related work in the areas of refinement and security. Conclusions are drawn in Chapter 8.

Functional refinement is a well established formal method. Appendix A relates the techniques of functional refinement in state–based systems to our work in security. We show how the definitions of non–interference and secure replacement are expressed in the notation of state–based specifications.

Appendix B shows how the work in Chapters 3 and 4 can be extended to properties more general than non–interference. A collection of laws for general security properties is presented.

Secure systems have developed their own, extensive jargon which is quite distinct from that familiar to those who construct systems without special reference to security. The final appendix provides a glossary of common terms.

An index of symbols is provided.

NB In places I have used the pronouns he and his and the objective him. That usage is not intended to be sexist, but is caused by a conscious decision not to use the clumsy s/he, him/her and his/hers; when referring to a computer system's users I do not distinguish between men, women and machines.

# Chapter 2

# Background

## 2.1 Introduction

This chapter introduces the mathematics required for the subsequent chapters and contains a summary of CSP (see [39]), the primary model of computation used in describing secure systems in this thesis.

The reader familiar with basic mathematical notation and CSP should proceed directly to Chapter 3. This chapter is intended as a reference for the basic notation used in the remainder of the thesis.

## 2.2 Relations and Orders

This section introduces the mathematical objects that form the basis of the subsequent work. We assume a basic knowledge of set theory, relations and functions and give some notation as well as defining equivalence relations and orders.

We use the following notation when describing relations:

**Definition 2.1** *If $R$ is a relation on $A \times B$ ($R : A \leftrightarrow B$) then the* domain *of $R$ is*

$$\mathbf{dom}\ R \triangleq \{a \mid a \in A \wedge \exists\, b \in B \bullet a\ R\ b\};$$

*the* range *of $R$ is*

$$\mathbf{ran}\ R \triangleq \{b \mid b \in B \wedge \exists\, a \in A \bullet a\ R\ b\}.$$

*If $(a, b) \in R$ then* $\mathbf{fst}\ (a, b) = a$ *and* $\mathbf{snd}\ (a, b) = b$.

*If $R : A \leftrightarrow A$ then the* restriction *of $R$ to $C \subseteq A$ is*

$$R \diamond C \triangleq R \cap C^2;$$

*the* anti–restriction *of $R$ to $C$ is*

$$R \mathbin{\Leftrightarrow} C \triangleq R \cap (A \setminus C)^2.$$

*The* identity *relation on $A$ is* $\mathbf{id}_A \triangleq \{(a, a) \mid a \in A\}$.

*The* inverse *of the relation $R : A \leftrightarrow B$ is* $R^{-1} \triangleq \{(b, a) \mid a \mathrel{R} b\}$.

*$F : A \to B$ denotes a function from $A$ to $B$.* △

**Definition 2.2** *$R : A \leftrightarrow A$ is* reflexive *iff $\mathbf{id}_A \subseteq R$; it is* symmetric *iff $R^{-1} = R$; and is* transitive *iff $(R \mathbin{;} R) \subseteq R$.* △

The empty set $\emptyset$ is both symmetric and transitive; it is not reflexive. A relation is anti–symmetric if whenever $a$ is related to $b$ and $b$ is related to $a$ then $a = b$.

**Definition 2.3** *If $R$ is a relation on $A$ and $R \cap R^{-1} = \mathbf{id}_A$ then $R$ is* anti–symmetric*.* △

An order (or ordering) is a special kind of relation. We define two kinds of ordering. When naming orderings we use symbols like $<$ instead of letters (such as $R$).

**Definition 2.4** *A* preorder *is a reflexive, transitive relation; a* partial order *is an anti–symmetric preorder. If $\leq$ is a partial order on the set $A$ we say that $A$ is* partially ordered *by $\leq$; similarly if $<$ is a preorder on $A$ then we say $A$ is* preordered *by $<$.* △

A partial order may have a top, an element greater than or equal to every element, and a bottom, an element less than or equal to every element.

**Definition 2.5** *If $\leq$ is a partial order on $A$ and there exists $\top_A \in A$ such that $\forall\, a \in A \bullet a \leq \top_A$ then $\top_A$ is a* top *element of $A$; similarly, if there exists $\bot_A \in A$ such that $\forall\, a \in A \bullet \bot_A \leq a$ then $\bot_A$ is a* bottom *element of $A$.* △

Given any two elements $a$ and $b$, of a partially ordered set $A$, an upper bound of $a$ and $b$ is an element $c \in A$ such that $a \leq c$ and $b \leq c$; the supremum of $a$ and $b$ is the least such upper bound. The infimum is the greatest lower bound of $a$ and $b$. Those definitions can be extended to sets of elements:

**Definition 2.6** *If $A$ is partially ordered by $\leq$ and $B \subseteq A$ then the* supremum *of $B$ is denoted $\bigsqcup B \in A$ and such that $\forall\, b \in B \bullet b \leq \bigsqcup B$ and $\forall\, b \in B \bullet b \leq a \Rightarrow \bigsqcup B \leq a$. The* infimum *$\bigsqcap B$ is similar.* $\triangle$

**Lemma 2.1** *If $A$ is partially ordered by $\leq$ and $B \subseteq A$ then there is at most one supremum and at most one infimum of $B$.*

**Proof** See [93, page 85]. $\square$

A partial order on $A$ is a lattice if $A$ is closed under the supremum and infimum of finite subsets of $A$. A lattice is complete if $A$ is closed under supermum and infimum of *any* subsets of $A$.

**Definition 2.7** *If $A$ is partially ordered by $\leq$ then $A$ is a* lattice *if, for all finite non–empty $B \subseteq A$, $\bigsqcup B$ and $\bigsqcap B$ exist in $A$. $A$ is a* complete lattice *if, for any $B \subseteq A$, $\bigsqcup B$ and $\bigsqcap B$ exist in $A$.* $\triangle$

Note that any complete lattice has a top and a bottom (see [93, page 89]).

A relation $R$ is an equivalence if: elements are equivalent to themselves; the order in which elements are tested for equivalence is irrelevant, and if $a$ is equivalent to $b$ and $b$ is equivalent to $c$ then $a$ is equivalent to $c$.

**Definition 2.8** *A relation $R$ is an* equivalence relation *iff it is reflexive, symmetric and transitive.* $\triangle$

An equivalence relation partitions its domain into sets called equivalence classes. Each of these equivalence classes consists of elements that are equivalent under the relation.

**Definition 2.9** *If $R$ is an equivalence relation on $A$ then the* equivalence class *of $a \in A$ is*

$$[a]^R \triangleq \{b \mid b \in A \wedge a\ R\ b\}. \qquad \triangle$$

When it is necessary to specify, or change, the value of a function at a point the operator $\oplus$ can be used. It alters the value of function at a single point leaving all other elements unchanged. (The notation $a \mapsto b$ means $(a, b)$ and is read *a maps to b*)

**Definition 2.10** *If* $F : A \rightarrow B$, $a \in A$ *and* $b \in B$ *then*

$$F \oplus \{a \mapsto b\} \triangleq \{(c, d) \mid d = (b \ \mathrm{C} \ c = a \ \mathrm{B} \ F(c))\}. \qquad \triangle$$

## 2.3 CSP

CSP is a notation with associated algebra and semantics for describing the behaviour of processes. A CSP process engages in events acting in synchrony with its environment. Events are atomic and occur in instantanteous co–operation with the environment.

The abstract syntax used in this thesis is given in Definition 2.11.

**Definition 2.11** *CSP has the abstract syntax:*

$$\begin{aligned} \mathsf{S} \ ::= \ & \mathtt{Stop}_A \mid \mathtt{Run}_A \mid \mathtt{Chaos}_A \mid e \rightarrow \mathsf{S} \mid (b : B \rightarrow \mathsf{S}_b) \mid \mathsf{S} \| \mathsf{T} \mid \\ & \mathsf{S} \sqcap \mathsf{T} \mid \mathsf{S} [\!] \mathsf{T} \mid \mathsf{S} \setminus C \mid \mathsf{S} \mathbin{//} \mathsf{T} \mid \mu \mathsf{X} \bullet \mathsf{F}(\mathsf{X}) \mid \mu \underline{\mathsf{X}} \bullet \underline{\mathsf{F}}(\underline{\mathsf{X}})_i. \end{aligned} \qquad \triangle$$

The process $\mathtt{Stop}_A$ does nothing; the process $\mathtt{Run}_A$ is willing to engage in any event from $A$ at any time; the process $\mathtt{Chaos}_A$ behaves chaotically—it may or may not engage in any event from $A$ at any time.

A prefixed process $(b : B \rightarrow \mathsf{S}_b)$ first engages in one of the events from $B$ and then behaves like $\mathsf{S}_b$ depending upon which element of $B$ was chosen.

The parallel combination of processes $\|$ forces the processes to agree on any common events; the internal choice operator $\sqcap$ specifies a non–deterministic choice beyond the control of the environment; external choice $[\!]$ allows the enivronment to choose.

The hiding operator $\setminus$ hides events taken from a set from the environment of the process on which is operates; the subordination operator $\mathsf{S} \mathbin{//} \mathsf{T}$ is defined as $\mathsf{S} \| \mathsf{T} \setminus \alpha \mathsf{T}$, i.e. $\mathsf{S}$ and $\mathsf{T}$ act in parallel with $\mathsf{T}$'s actions hidden from the environment.

Recursion $\mu \mathsf{X} \bullet \mathsf{F}(\mathsf{X})$ is defined in terms of the least fixed point of the function $\mathsf{F}$ in the models of CSP being used in this thesis; see [39, 78] for details.

Definition 2.11 has one non–standard syntactic element: mutual recursion. A mutually recursive process is defined in [39, page 38] as one of the collection of processes $\mathsf{X}_i = \mathsf{F}(\underline{\mathsf{X}})_i$ where $\underline{\mathsf{X}}$ is the vector of the processes $\mathsf{X}_i$ and $\underline{\mathsf{F}}$ a vector of functions on processes. We use the notation $\mu \underline{\mathsf{X}} \bullet \underline{\mathsf{F}}(\underline{\mathsf{X}})_i$ to refer to $\mathsf{X}_i$ when that recursion has a unique solution.

**Definition 2.12** *If* $\mathsf{X}_i = \mathsf{F}(\underline{\mathsf{X}})_i$ *and the* $\mathsf{F}(\underline{\mathsf{X}})_i$ *are guarded then we write* $\mu\,\underline{\mathsf{X}} \bullet \underline{\mathsf{F}}(\underline{\mathsf{X}})_i$ *for* $\mathsf{X}_i$*.* △

There are a number of semantic models of CSP to cover deterministic (see [38]), non–deterministic, deadlocking, diverging (see [13, 14, 15, 39, 82]—[78] contains a summary of those models), timed (see [20, 79, 80, 81, 89]) and probabilistic behaviour (see [90]).

The untimed models used in the majority of this thesis are the traces model and the failures–divergences model and we make use of the algebra of CSP (both those models and the algebraic laws are described in [39]).

### The Traces Model of CSP

Some notation (much from [39]) will be useful in reasoning about traces.

**Definition 2.13** *If $A$ is a set then $A^*$ is the set of finite sequences of elements of $A$.*

*The* catenation *of traces $s$ and $t$ is the trace formed by putting them together in that order; it is denoted $s^\frown t$.*

*Trace $s$ is* in *trace $t$ if there exists $s_0$ and $s_1$ such that*

$$t = s_0{}^\frown s^\frown s_1.$$

*A trace operator $f$ is* strict *if $f(\langle\rangle) = \langle\rangle$, and* distributive *if it distributes through $^\frown$, i.e. $f(s^\frown t) = f(s)^\frown f(t)$.*

Restriction $\restriction$ *is strict and distributive, its affect on the singleton trace is $\langle e\rangle \restriction A \triangleq (\langle e\rangle \ \mathrm{C}\ e \in A \ \mathrm{B}\ \langle\rangle)$. Note that $t \restriction \emptyset = \langle\rangle$ and $t \restriction A \restriction B = t \restriction (A \cap B)$.*

*The* length *of a trace is the number of events in the trace:*

$$\#\langle\rangle \triangleq 0\,; \quad \#\langle e\rangle \triangleq 1\,; \quad \#(s^\frown t) \triangleq \#s + \#t.$$

*The number of occurrences of the event $e$ in the trace $t$ is*

$$t \downarrow e \triangleq \#(t \restriction \{e\}).$$

*Four operators extract the first and last element of a trace, and all but the first and last elements of a trace (none are defined for the empty trace) :*

$$\begin{aligned} \textbf{head}\ (\langle e\rangle^\frown t) &\triangleq \textbf{last}\ (t^\frown\langle e\rangle) \triangleq e; \\ \textbf{tail}\ (\langle e\rangle^\frown t) &\triangleq \textbf{init}\ (t^\frown\langle e\rangle) \triangleq t. \end{aligned}$$

*The* set of events *in a trace* $t$ *is* **set** $t$*:*

$$\textbf{set}\ \langle\rangle \triangleq \emptyset\ ; \quad \textbf{set}\ \langle e\rangle \triangleq \{e\}\ ; \quad \textbf{set}\ (s ^\frown t) \triangleq (\textbf{set}\ s) \cup (\textbf{set}\ t). \qquad \triangle$$

The traces of a process $\mathsf{S}$ is a non–empty prefix–closed set of finite sequences of events. The events are drawn from the alphabet of the process—the set of events in which $\mathsf{S}$ may engage. We follow the use in [44] of Greek letters to denote the various semantic functions on a process.

**Definition 2.14** *If* $\mathsf{S}$ *is a process then it has alphabet* $\alpha\mathsf{S}$ *and traces* $\tau\mathsf{S} \subseteq \alpha\mathsf{S}^*$*.* $\tau\mathsf{S}$ *is prefix–closed and contains the empty trace* $\langle\rangle$*.* $\triangle$

In the traces model a process $\mathsf{S}$ is identified with the pair $(\alpha\mathsf{S}, \tau\mathsf{S})$; the definition of the alphabet and traces of processes composed using the operators in Definition 2.11 can be found in [39]. The traces model provides a model of deterministic behaviour and does not distinguish between internal and external non–determinism.

The events in which a process may initially engage are called its initials.

**Definition 2.15** *If* $\mathsf{S}$ *is a process then the* initials *of* $\mathsf{S}$ *are*

$$\iota\mathsf{S} \triangleq \{\textbf{head}\ t \mid t \in \tau\mathsf{S} \setminus \{\langle\rangle\}\}. \qquad \triangle$$

The traces of a process describe the sequences of events a process is prepared to engage in; they do not give information about non–determinism, deadlock or divergence. Non–determinism and deadlock are captured by defining the failures of a process.

### The Failures–Divergences Model of CSP

A failure is a pair consisting of a trace and a set of refusals. A refusal is a set of events from the alphabet of the process. After the process has engaged in the trace it may refuse to engage in any of the events contained in the refusals (the set of refusals is subset closed).

The following definition is a summary of the definitions in [39, page 130].

**Definition 2.16** *If* $\mathsf{S}$ *is a process then* $\phi\mathsf{S} : \alpha\mathsf{S}^* \leftrightarrow \mathbb{P}\, \alpha\mathsf{S}$ *is the set of* failures *of* $\mathsf{S}$*.* $\phi\mathsf{S}$ *satisfies:*

$$
\begin{aligned}
(\langle\rangle, \emptyset) &\in \phi\mathsf{S}; \\
(s^\frown t, X) \in \phi\mathsf{S} &\Rightarrow (s, \emptyset) \in \phi\mathsf{S}; \\
(t, X) \in \phi\mathsf{S} \wedge Y \subseteq X &\Rightarrow (t, Y) \in \phi\mathsf{S}; \\
(t, X) \in \phi\mathsf{S} \wedge e \in \alpha\mathsf{S} &\Rightarrow (t, X \cup \{e\}) \in \phi\mathsf{S} \vee (t^\frown\langle e\rangle, \emptyset) \in \phi\mathsf{S}. \quad \triangle
\end{aligned}
$$

Note that $\tau\mathsf{S} = \mathbf{dom}\ \phi\mathsf{S}$.

The semantic function for expressing divergence is $\delta\mathsf{S}$. A divergence of a process is a trace after which the process behaves chaotically, see [39, page 130].

**Definition 2.17** *If* $\mathsf{S}$ *is a process then the set of* divergences *is* $\delta\mathsf{S} \subseteq \tau\mathsf{S}$. *Any extension of a divergence is a divergence and after a divergence any event may be refused:*

$$
\begin{aligned}
s \in \delta\mathsf{S} \wedge t \in \alpha\mathsf{S}^* &\Rightarrow (s^\frown t) \in \delta\mathsf{S}; \\
s \in \delta\mathsf{S} \wedge X \subseteq \alpha\mathsf{S} &\Rightarrow (s, X) \in \phi\mathsf{S}.
\end{aligned}
\quad \triangle
$$

In the failures–divergences model the process $\mathsf{S}$ is identified with the triple $(\alpha\mathsf{S}, \phi\mathsf{S}, \delta\mathsf{S})$; definitions of those triples for each of the CSP operators can be found in [39, pages 131–132].

In this thesis we use the algebra of CSP, as much as possible, and avoid a particular semantic model. Where necessary we use the failures–divergences model.

## 2.4 Refinement in CSP

One process $\mathsf{T}$ refines another $\mathsf{S}$ if the observed behaviour of $\mathsf{T}$ is a possible behaviour of $\mathsf{S}$. Then $\mathsf{T}$ can replace $\mathsf{S}$ without exhibiting any unexpected behaviour. The observations may take the form of traces or failures–divergences.

In the traces model process $\mathsf{T}$ refines $\mathsf{S}$ when the traces of $\mathsf{T}$ are a subset of the traces of $\mathsf{S}$, i.e. any trace of $\mathsf{T}$ is a trace of $\mathsf{S}$: the behaviour of $\mathsf{T}$ is expected of $\mathsf{S}$.

**Definition 2.18** *If* $\mathsf{S}$ *and* $\mathsf{T}$ *are processes then* $\mathsf{T}$ refines $\mathsf{S}$, *written* $\mathsf{S} \sqsubseteq \mathsf{T}$, *iff*

$$(\alpha\mathsf{S} = \alpha\mathsf{T}) \wedge (\tau\mathsf{T} \subseteq \tau\mathsf{S}). \quad \triangle$$

In traces $\sqsubseteq$ forms a complete lattice with top $\mathtt{Run}_A$ and bottom $\mathtt{Stop}_A$ and $(A, \tau\mathsf{S} \cap \tau\mathsf{T})$ is the supremum of $(A, \tau\mathsf{S})$ and $(A, \tau\mathsf{T})$.

In the failures–divergences model $\mathsf{T}$ refines $\mathsf{S}$ if $\mathsf{T}$ is no more non–deterministic than $\mathsf{S}$ and diverges no more than $\mathsf{S}$.

**Definition 2.19** *If* $\mathsf{S}$ *and* $\mathsf{T}$ *are processes then* $\mathsf{T}$ refines $\mathsf{S}$*, written* $\mathsf{S} \sqsubseteq \mathsf{T}$*, iff*

$$(\alpha\mathsf{S} = \alpha\mathsf{T}) \wedge (\phi\mathsf{T} \subseteq \phi\mathsf{S}) \wedge (\delta\mathsf{T} \subseteq \delta\mathsf{S}). \qquad \triangle$$

In failures–divergences the order $\sqsubseteq$ on processes forms a complete lattice with every deterministic process as a top and $\mathtt{Chaos}_A$ as the bottom element. $(A, \phi\mathsf{S} \cap \phi\mathsf{T}, \delta\mathsf{S} \cap \delta\mathsf{T})$ is the supremum of $(A, \phi\mathsf{S}, \delta\mathsf{S})$ and $(A, \phi\mathsf{T}, \delta\mathsf{T})$.

# Chapter 3

# Non–interference

## 3.1 Introduction

In this chapter we introduce the definitions of information flow and non–interference. Before that we examine some of the factors that motivate the definition by considering three of the standard 'security flaws' of computer systems. By examining these flaws and considering their positive counterparts (the security policies that prevent them) as properties about information flow we are then able to suggest a way of capturing security.

We define security by constructing the states of a trace–based system observed from the viewpoint of individual users. This definition of local state depends solely on the observed interactions of the users at defined interfaces. We define states as classes of traces and this leads us to a definition of non–interference as a relation on traces. We then show how to cast these definitions in the process algebra of CSP. This extension is useful as we are freed from using semantic definitions and are able to work algebraically.

We examine a few examples to illustrate non–interference and highlight some of its properties.

## 3.2 Information Flow

Conventional specification techniques revolve around the specification of the functional requirements of a system; those requirements are given in terms of the safety and liveness properties of such a system.

We specify security separately from safety and liveness for two reasons.

The Bell & LaPadula specification of the security of MULTICS,

see [10], was shown in [66, 67] to be insecure because an operation to downgrade all objects to unclassified was permitted by their model (this problem was caused in part by not considering security at a high enough level of abstraction and the problems caused by the Bell & LaPadula model motivated much work in information flow, see for example comments in [44, page 95]).

Also, it is necessary to consider security separately when attempting a refinement of a system (it has been shown in [32, 47] and by others that functional refinement does not preserve security properties. We shall return to this problem in Chapter 4). In order to perform security–preserving refinements the standard techniques of refinement must be augmented to preserve the policies specified. Such refinement techniques are discussed later.

Our work is based on two assumptions: that users gain information through a clearly specified interface (interfaces are defined below); and that users do not necessarily know the construction (e.g. they do not need know the entire code and the state) of the system they are using.

### 3.2.1 Principle Security Properties

In 1981 Carl Landwehr made a survey of the various formal techniques available for the construction of secure computer systems (see [59]). In doing so he identified three areas of particular concern when considering a secure computer system. The three possible 'security threats' which should be considered when examining a computer system, Landwehr suggests, are: unauthorized disclosure of information (Jacob, see [50], and others call the policy that prevents this the *confidentiality* property); unauthorized modification of information (called in [11, 49] the *integrity* property) and the unauthorized withholding of information (termed the *availability* property in [42] and also commonly called denial of service). Both confidentiality and integrity are properties which exist in secure computing and in the paper–based military world which many systems emulate (see [59]). Denial of service is one of a number of problems (such as covert communication and viruses) which only occur in computer systems (see comments in [59, page 5, 6]). These new problems force the examination of security at a higher level of abstraction than relating computer systems to their paper analogues (see [44]).

**confidentiality** Suppose a file–store contains a file $f$, created and owned by user *high*, which may not be examined by user *low*. Then there is an unauthorized disclosure of the information in $f$ to *low* (*low* can read $f$ say), if information has flowed from *high*

to *low*. Changes in $f$ can be observed by *low* when he reads the file.

**integrity** Consider again the file $f$. If *low* can write to $f$ then *high* will be able to observe the change and information will have flowed from *low* to *high*.

**availability** Suppose that *low* is able to prevent *high* from accessing file $f$ in some way; then *low*'s commands are able to affect *high*'s information (since ability to access information is a form of information) and there is a flow from *low* to *high*.

Many authors, for example [22, 25, 28, 46, 84], have suggested that security can be viewed by considering the flow of information within the system, as we have seen. We can view Landwehr's three properties as properties of information flow.

Exactly what is meant by 'information flow' is not clear and authors disagree about its definition. Our idea of information flow comes from a quotation (an informal definition) taken from [2]:

> Information is transmitted over a channel when variety is conveyed from the source to the destination.

We interpret this information–theory concept by considering the system as the channel in use and the source and destination as users. Non–interference from source to destination will mean that no variety in the source is conveyed to the destination.

Thus we specify security in terms of information flow from user to user; to do so we need to identify the users and their interfaces to the system. We define non–interference by describing when a user is unable to distinguish different states of the system, and thus when variety in the system cannot be observed by a user. To do so we need first to define our notions of users, states and indistinguishability, which we do in the next section.

### 3.2.2 Equivalent Traces

We model a system as a CSP process. A user of a system has an interface consisting of events in which he can engage. These events form the only way that a user can deduce information about, or provide information to, the system in use.

**Definition 3.1** *A user $v$ of a system* $\mathsf{S}$ *is defined by his* interface*, a set of events, $v \subseteq \alpha\mathsf{S}$.* $\triangle$

The traces $\tau\mathsf{S}$, along with the set of events, $\alpha\mathsf{S}$, can be used to construct the set of states. The states are equivalence classes of traces which appear indistinguishable to any interaction across $\alpha\mathsf{S}$. When two traces are indistinguishable we consider the system to be in the 'same' state, as in both cases there is no way to differentiate the states. (That idea is illustrated in Example 3.1.) A trace is indistinguishable from another if any sequence of events that can occur after one trace can also occur after the other and vice versa. This view of state is totally external. We are only considering the interactions at an interface as the way of determining state. We do so as we are later to define security by the relationship between interfaces.

We define a relation on the traces from which the states are constructed. The relation holds for traces $s$ and $t$ when after $s$ or $t$ the system is indistinguishable at the interface $\alpha\mathsf{S}$. Thus equivalence classes under this relation form the states we require.

In the traces model of CSP the definition is given in the following way.

**Definition 3.2** *In system* $\mathsf{S}$*, with $s$ and $t$ in $\alpha\mathsf{S}^*$, $s$ and $t$ are* equivalent *(written $s \approx^{\mathsf{S}} t$) iff* $\forall r \in \alpha\mathsf{S}^* \bullet (s^\frown r \in \tau\mathsf{S} \Leftrightarrow t^\frown r \in \tau\mathsf{S})$. $\triangle$

This definition is given in the traces semantics of CSP, we recast the definition in the algebra of CSP.

**Definition 3.3** *In system* $\mathsf{S}$*, with $s$ and $t$ in $\alpha\mathsf{S}^*$, $s$ and $t$ are* equivalent *(written $s \approx^{\mathsf{S}} t$) iff* $\mathsf{S}/s = \mathsf{S}/t$. $\triangle$

The use of process algebra enables us to give definitions and proofs which are not expressed in a particular semantics. Note that Definition 3.3 is stronger than Definition 3.2 as the equality given in Definition 3.3 implies the expression of $s \approx^{\mathsf{S}} t$ in Definition 3.2.

The relation $s \approx^{\mathsf{S}} t$ is an equivalence relation and the equivalence classes form *global states* for the system $\mathsf{S}$; a class containing trace $t$ will be written $[t]^{\mathsf{S}}$.

**Lemma 3.1** $\approx^{\mathsf{S}}$ *is an equivalence relation.*

**Proof** Obvious from the equality $\mathsf{S}/s = \mathsf{S}/t$ in Definition 3.3. $\square$

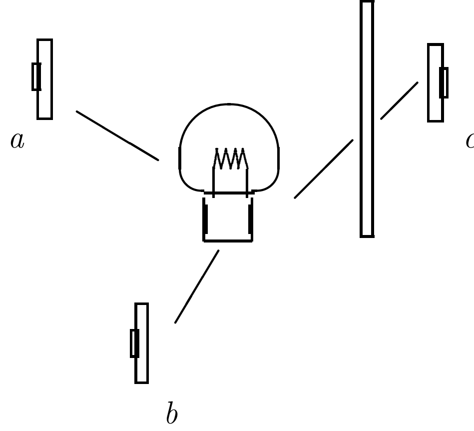


Figure 1: Process $\mathsf{L}$.

To illustrate global states consider the following example of a CSP process.

**Example 3.1** The process $\mathsf{L}$ models a lamp with three switches. At each activation of a switch the lamp reverses its state, changing from on to off or off to on. The three switches are defined by the interfaces $\{a.0, a.1\}$, $\{b.0, b.1\}$ and $\{ce\}$. At the first two of these interfaces the state (whether on, represented by 1, or off, represented by 0) of the lamp is visible (it is part of the event), whereas the third interface does not have that state information. This example will be used to illustrate a number of our definitions and to aid its understanding Figure 1 is provided. In that figure $a$ and $b$ are the switches to the left and $c$ is hidden behind a screen on the right.

The process $\mathsf{L}$ is defined

$$\begin{aligned}\mathsf{L} &\triangleq \mathsf{L}_0\\ \mathsf{L}_i &\triangleq (a.i \rightarrow \mathsf{L}_{\neg i} \mid b.i \rightarrow \mathsf{L}_{\neg i} \mid ce \rightarrow \mathsf{L}_{\neg i}),\end{aligned}$$

where the subscript $i$ denotes the state of the lamp ($\neg 0 = 1$ and $\neg 1 = 0$). The traces of this process are described as follows:

$$\begin{aligned}&\langle\rangle \in \tau\mathsf{L}\\ &t \in \tau\mathsf{L} \Rightarrow t^\frown\langle ce\rangle \in \tau\mathsf{L}\\ &t \in \tau\mathsf{L} \wedge \exists\, n \in \mathbb{N} \bullet \#t = 2n \Rightarrow t^\frown\langle a.0\rangle \in \tau\mathsf{L} \wedge t^\frown\langle b.0\rangle \in \tau\mathsf{L}\\ &t \in \tau\mathsf{L} \wedge \exists\, n \in \mathbb{N} \bullet \#t = 2n+1 \Rightarrow t^\frown\langle a.1\rangle \in \tau\mathsf{L} \wedge t^\frown\langle b.1\rangle \in \tau\mathsf{L}\end{aligned}$$

and $\mathsf{L}$ has two global states (sets of traces) representing the case when the lamp is on

$$[\langle ce\rangle]^{\mathsf{L}} = \{t \in \tau\mathsf{L} \mid \exists\, n \in \mathbb{N} \bullet \#t = 2n+1\},$$

and when the lamp is off

$$[\langle\rangle]^{\mathsf{L}} = \{t \in \tau\mathsf{L} \mid \exists\, n \in \mathbb{N} \bullet \#t = 2n\}.$$

Algebraically the states are the processes $\mathsf{L}/\langle ce\rangle$ and $\mathsf{L}$. $\triangle$

Similarly, we can define the states that a particular user perceives. These states are equivalence classes of traces or processes. The definition is similar to Definition 3.3, but instead of examining all possible sequences of events we only consider those events that can be engaged in by the user in question; i.e. those events which form that user's interface.

**Definition 3.4** *If in system* $\mathsf{S}$ *with user* $v$ *and* $s$ *and* $t$ *in* $\alpha\mathsf{S}^*$

$$\forall\, r \in v^* \bullet (s^\frown r \in \tau\mathsf{S} \Leftrightarrow t^\frown r \in \tau\mathsf{S})$$

*then* $s$ *and* $t$ *are* $v$–equivalent*, written* $s \approx_v^{\mathsf{S}} t$*.* $\triangle$

Again algebra will provide a semantics–free definition. To give that definition we define a special process for a user $v$ which describes the user engaging in any events from $v$ while the other users are prevented from interacting at all.

**Definition 3.5** *If user* $v$ *is a subset of alphabet* $A$ *then* $\overline{v}_A \triangleq \mathtt{Stop}_{A\setminus v}$ *is the process which only allows* $v$*'s events to occur.* $\triangle$

$\mathtt{Stop}$ is used here instead of hiding $\setminus$ to abstract from the other users' events because hiding would continue to allow other users to engage in events and change the state that is under examination. Using $\mathtt{Stop}$ allows a user to determine the current state at his interface, by preventing other users from changing the state.

We define the equivalence relation $\equiv_v^A$, which holds when two processes are equal when put in an environment only allowing $v$ to interact with a system with alphabet $A$.

**Definition 3.6** *If systems* $\mathsf{S}$ *and* $\mathsf{T}$*, with* $\alpha\mathsf{S} = \alpha\mathsf{T} = A$*, have user* $v$*, then* $\mathsf{S} \equiv_v^A \mathsf{T}$ *iff*

$$\mathsf{S} \| \overline{v}_A = \mathsf{T} \| \overline{v}_A.$$ $\triangle$

The definition of $\equiv_v^A$ is used to give the algebraic form of $\approx_v^{\mathsf{S}}$.

**Definition 3.7** *If in system* $\mathsf{S}$ $(\alpha\mathsf{S} = A)$ *with user* $v$ *and* $s$ *and* $t$ *in* $\alpha\mathsf{S}^*$

$$\mathsf{S}/s \equiv_v^A \mathsf{S}/t$$

*then $s$ and $t$ are $v$–*equivalent*, again written $s \approx_v^{\mathsf{S}} t$.* △

As with $\approx^{\mathsf{S}}$ we find that the algebraic definition is stronger than the traces definition.

The relation $\approx_v^{\mathsf{S}}$ is also an equivalence relation and the equivalence classes it induces can be considered as the *local states* perceived by a particular user. We write $[t]_v^{\mathsf{S}}$ for the $v$–equivalence class (state) containing trace $t$ from system $\mathsf{S}$.

These classes are fundamental to our definition of security as we are considering security by examining flow of information through a system. A user can detect an information flow only by examining his interface, or, equivalently, the local state he perceives. Changes in the local state may reflect communication (see Example 3.2 below).

**Lemma 3.2** *If system $\mathsf{S}$ with alphabet $A$ has user $v$ then $\approx_v^{\mathsf{S}}$ and $\equiv_v^A$ are equivalence relations.*

**Proof** Obvious from the equalities in Definitions 3.6 and 3.7. □

Returning to the process $\mathsf{L}$ we examine the states apparent to two of the users.

**Example 3.2** We define $a$ by $a \triangleq \{a.0, a.1\}$ and $c$ by $c \triangleq \{ce\}$. There are two states from $a$'s point of view

$$
\begin{aligned}
[\langle ce \rangle]_a^{\mathsf{L}} &= \{t \in \tau \mathsf{L} \mid \exists\, n \in \mathrm{N} \bullet \#t = 2n+1\} \\
[\langle \rangle]_a^{\mathsf{L}} &= \{t \in \tau \mathsf{L} \mid \exists\, n \in \mathrm{N} \bullet \#t = 2n\},
\end{aligned}
$$

exactly the global states that we found in Example 3.1. From $c$'s viewpoint, however, there is only one state, the set of traces $[\langle \rangle]_c^{\mathsf{L}} = \tau \mathsf{L}$ since $c$ cannot view the lamp and so cannot tell whether it is on or off.

Note that after the trace $\langle b.0 \rangle$ the lamp is on and $a$ can see this by engaging in the event $a.1$. But if $b$ had not flipped the switch $a$ would only be able to engage in $a.0$ and thus $b$ can communicate with $a$ by way of the lamp. △

From the last two examples we also see that $\approx^{\mathsf{L}} \subseteq \approx_c^{\mathsf{L}}$; a similar inclusion is true for any system and any user. The following lemma shows that removing events from the interface of a user does not enable that user to distinguish more traces and that $\approx^{\mathsf{S}}$ is at least as fine as any $\approx_v^{\mathsf{S}}$.

**Lemma 3.3** *If system $\mathsf{S}$ has users $u$ and $v$ and alphabet $A$, then*

$u \subseteq v \Rightarrow \equiv_u^A \supseteq \equiv_v^A$, $u \subseteq v \Rightarrow \approx_u^{\mathsf{S}} \supseteq \approx_v^{\mathsf{S}}$, *and* $\approx^{\mathsf{S}} \subseteq \approx_u^{\mathsf{S}}$.

**Proof** If systems $\mathsf{P}$ and $\mathsf{Q}$ have $\alpha\mathsf{P} = \alpha\mathsf{Q} = A$ and $u \subseteq v$ then

$$
\begin{array}{lll}
 & \mathsf{P} \equiv_v^A \mathsf{Q} & \\
\Leftrightarrow & \mathsf{P} \| \overline{v}_A = \mathsf{Q} \| \overline{v}_A & \text{[Definition 3.6]} \\
\Leftrightarrow & \mathsf{P} \| \mathrm{Stop}_{A \backslash v} = \mathsf{Q} \| \mathrm{Stop}_{A \backslash v} & \text{[Definition 3.5]} \\
\Rightarrow & \mathsf{P} \| \mathrm{Stop}_{A \backslash u} = \mathsf{Q} \| \mathrm{Stop}_{A \backslash u} & [\mathrm{Stop}_X \| \mathrm{Stop}_Y = \mathrm{Stop}_{X \cup Y}] \\
\Leftrightarrow & \mathsf{P} \| \overline{u}_A = \mathsf{Q} \| \overline{u}_A & \text{[Definition 3.5]} \\
\Leftrightarrow & \mathsf{P} \equiv_u^A \mathsf{Q}, & \text{[Definition 3.6]}
\end{array}
$$

and hence $\equiv_u^A \supseteq \equiv_v^A$.

For $s$ and $t$ in $\alpha\mathsf{S}^*$,

$$
\begin{array}{lll}
 & s \approx_v^{\mathsf{S}} t & \\
\Leftrightarrow & \mathsf{S}/s \equiv_v^A \mathsf{S}/t & \text{[Definition 3.7]} \\
\Rightarrow & \mathsf{S}/s \equiv_u^A \mathsf{S}/t & \\
\Leftrightarrow & s \approx_u^{\mathsf{S}} t. & \text{[Definition 3.7]}
\end{array}
$$

Hence $\approx_u^{\mathsf{S}} \supseteq \approx_v^{\mathsf{S}}$.

If user $\epsilon = \alpha\mathsf{S}$, then $\approx^{\mathsf{S}} = \approx_\epsilon^{\mathsf{S}}$ and $u \subseteq \alpha\mathsf{S} = \epsilon$, and so $\approx^{\mathsf{S}} \subseteq \approx_u^{\mathsf{S}}$. □

An additional property of $\approx_v^{\mathsf{S}}$ is that if traces $s$ and $t$ are indistinguishable to $v$ ($s \approx_v^{\mathsf{S}} t$) then so are $s \frown r_0$ and $t \frown r_0$ (where $r_0$ consists of events in which $v$ engage). That is, if two traces are indistinguishable to $v$ then no events in which $v$ can engage will later enable him to distinguish them.

This fact is important because it highlights a significant property of our definition of equivalence which is not present in many other formalisms used to analyse secure systems; namely, two traces are $v$–equivalent if they cannot be distinguished by $v$ at any time in the future. Many formalisms only consider the 'next output' when defining equivalence (see, for example, [28, 64, 84]).

**Lemma 3.4** *In a system* $\mathsf{S}$ *with user* $v$ *with traces* $s$ *and* $t$*,*

$$s \approx_v^{\mathsf{S}} t \Rightarrow s \frown r_0 \approx_v^{\mathsf{S}} t \frown r_0$$

*for any sequence of* $v$*–events,* $r_0 \in v^*$*.*

**Proof** Consider $r_0 \in v^*$ and suppose that $\alpha\mathsf{S} = A$.

$$
\begin{array}{lll}
 & s \approx_v^{\mathsf{S}} t & \\
\Leftrightarrow & (\mathsf{S}/s) \| \overline{v}_A = (\mathsf{S}/t) \| \overline{v}_A & \text{[Definitions 3.6 and 3.7]} \\
\Rightarrow & (\mathsf{S}/s) \| \overline{v}_A / r_0 = (\mathsf{S}/t) \| \overline{v}_A / r_0 &
\end{array}
$$

$$
\begin{array}{lll}
\Rightarrow & ((\mathsf{S}/s)/r_0)\|\overline{v}_A = ((\mathsf{S}/t)/r_0)\|\overline{v}_A & \text{[L2:72; L1:53]} \\
\Rightarrow & (\mathsf{S}/s^\frown r_0)\|\overline{v}_A = (\mathsf{S}/t^\frown r_0)\|\overline{v}_A & \text{[L2:53]} \\
\Leftrightarrow & s^\frown r_0 \approx_v^{\mathsf{S}} t^\frown r_0. & \text{[Definitions 3.6 and 3.7]} \quad \square
\end{array}
$$

We can now proceed to define non–interference.

## 3.3 Non–interference

The papers [28] and [84] defined the information flow property non–interference (see Section 7.4.1). This property was designed to capture the idea that information cannot flow from one user to another if the events in which a user engages do not affect the states another sees. Rushby writes:

> We say that there is no flow of information from one user to another, or that the first user is *noninterfering* with the second, if the results seen by the second are completely unaffected by the presence or absence of events issued by the first. [84, page 3]

Millen showed, using the techniques of information theory, that asserting that one user does not interfere with another guarantees zero–bits per second flow from the first user to the second (see [72]).

The definitions given in the papers cited were both in terms of state machines and outputs generated by these machines, but we can easily translate the definitions into our notation. We borrow the symbol $\not\rightsquigarrow$ from [84] and write $u \not\rightsquigarrow v$ when we mean user $u$ does not interfere with user $v$.

**Definition 3.8** *For a user $u$ the function $|_u$ removes events which belong to $u$ from a trace; it is defined, for the singleton trace by*

$$\langle e\rangle|_u \;\triangleq\; \langle\rangle \;\mathrm{C}\; e \in u \;\mathrm{B}\; \langle e\rangle,$$

*and is strict and distributive.* $\triangle$

Then a user $u$ is non–interfering with another user $v$ if traces $t$ and $t|_u$ ($t$ with events from $u$'s interface removed) are indistinguishable from $v$'s viewpoint. That is, $v$ is unable to detect the presence or absence of $u$'s events; $v$'s only way to detect those events is through the equivalence $\approx_v^{\mathsf{S}}$, and the definition of non–interference prevents $v$ from observing any difference between traces which do and do not contain any events from $u$'s interface.

**Definition 3.9** *In a system* $\mathsf{S}$ *with users* $u$ *and* $v$*,* $u$ *is* non–interfering *with* $v$ ($u \not\rightsquigarrow v : \mathsf{S}$) *iff* $\forall\, t \in \tau\mathsf{S} \bullet t \approx^{\mathsf{S}}_{v} t|_{u}$*, with the assumption that the users are disjoint:* $u \cap v = \emptyset$*.* △

The assumption that the users are disjoint is necessary because if there were an event common to both users our use of a synchronous model of computation would require that both users co–operated when the event was performed and then they would have communicated by synchronization.

To illustrate that definition we return to process $\mathsf{L}$, the lamp with three switches. As before $a$ (and $b$) have switches that flip the state of the bulb and have the bulb in their view. User $c$ can change the state of the bulb with his switch but cannot see it. We would expect that $a$ is unable to communicate with $c$ and examine $a \not\rightsquigarrow c$ to see if this is true.

**Example 3.3** Recall (from Example 3.2) that $a = \{a.0, a.1\}$ and $c = \{ce\}$. We find that $a$ is non–interfering with $c$. Although $a$ can change the lamp's state $c$ cannot view the bulb and $c$ never knows (i.e. can never deduce from his interface) what events $a$ (or $b$) has engaged in.

To see this consider any trace $t$ of $\mathsf{L}$. From $c$'s viewpoint there is only one equivalence class of traces, $\tau\mathsf{L}$ and so both $t$ and $t|_{a}$ are in that class. Thus $t \approx^{\mathsf{L}}_{c} t|_{a}$ and so $a \not\rightsquigarrow c : \mathsf{L}$. △

An important property of non–interference is (see [28]) that it is intransitive and we have given its definition in terms of the transitive relation $\approx^{\mathsf{S}}_{v}$. It is, however, true that our definition of non–interference is intransitive, as illustrated by the following example.

**Example 3.4** Consider the system $\mathsf{M}$, which is similar to $\mathsf{L}$ (Example 3.1) except that we have restricted $c$ further by preventing his switch from changing the state of the bulb (see Figure 2).

$$\begin{aligned} \mathsf{M} &\triangleq \mathsf{M}_0 \\ \mathsf{M}_i &\triangleq (a.i \rightarrow \mathsf{M}_{\neg i} \mid b.i \rightarrow \mathsf{M}_{\neg i} \mid ce \rightarrow \mathsf{M}_i) \end{aligned}$$

It has traces $\tau\mathsf{M} = \{t \mid t \in \alpha\mathsf{M}^* \wedge t|_{c} \in \tau\mathsf{L}\}$.

In this case $a \not\rightsquigarrow c : \mathsf{M}$ (we established that fact in Example 3.3 and, although we have modified $\mathsf{L}$, $c$ still has only one local state, $\tau\mathsf{M}$) and $c \not\rightsquigarrow b : \mathsf{M}$ (since $c$'s events do not change the lamp's state), but $a$ interferes with $b$. This interference occurs because $a$ can change the state of the lamp and $b$ can view that state, as Example 3.2 illustrates.

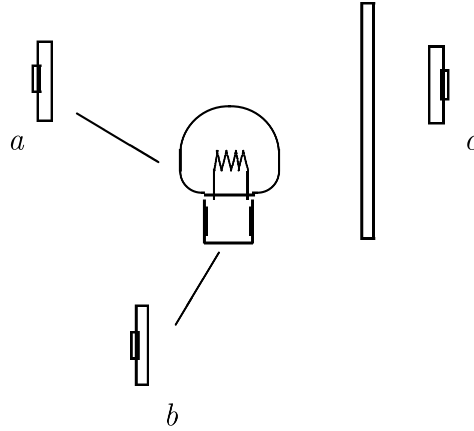


Figure 2: Process M.

Thus non–interference is intransitive. △

Frequently secure computer systems are considered to have two users, named *high* and *low*, and the simple security policy: no information may flow from *high* to *low*. That is captured by writing $high \not\rightsquigarrow low$. In the examples we have considered so far the users' events have partitioned the set of events in which those processes could engage. It is important to consider an example where that is not the case and we do so here.

**Example 3.5** Consider the following system, S with three users

$$\mathsf{S} \triangleq (ae \to ce \to \mathsf{S} \mid be \to \mathsf{S}),$$

and compare it with the similar system R,

$$\mathsf{R} \triangleq (ae \to \mathsf{R} \mid be \to \mathsf{R}).$$

Define the users $a$, $b$ and $c$ by $\{ae\}$, $\{be\}$ and $\{ce\}$, respectively.

Firstly $a \not\rightsquigarrow b : \mathsf{R}$ as is easily seen from the traces of R as $\tau\mathsf{R} = \{ae, be\}^*$. But the same is not true of S; in S the third user $c$ can prevent $b$ from performing the event $be$, but only after $a$ has performed an $ae$. Thus $a$ interferes with $b$ by way of $c$.

Although there is no problem with S not exhibiting non–interference (it is correct: $a$ can communicate with $b$) there may be cases where this effect is undesirable. Suppose the event $c$ were an internal event of the system (an auditing event, for example) then $c$ would normally be hidden from view and should not affect the operation of the machine from $b$ and $a$'s points of view, or the analysis of security, as we have defined security between visible interfaces.

The solution is to hide such internal events when examining security (in our definition) and the CSP operator $\backslash$ does just that. We obtain $\mathsf{S} \setminus c = \mathsf{R}$. $\triangle$

One final point is that some 'obviously' non–interfering systems are not as expected. This results from the distinction between $\mathsf{P}/t$ when $t$ is a trace of $\mathsf{P}$ and $\mathsf{P}/t$ when $t$ is not a trace. In the former case the process is well–defined, but in the later the process is undefined. Such a $t$ not in the traces of $\mathsf{P}$ may occur when the events belonging to a user are purged.

Nevertheless it is important to point out a simple example of a system that is not non–interfering (although it appears to be) as such small examples may accidentally be used as building blocks when the laws we shall present are used in the development of systems.

**Example 3.6** Consider the two systems $\mathsf{R}$ and $\mathsf{S}$,

$$\mathsf{R} \triangleq ae \to ce \to \mathtt{Stop}_{\{ae,be,ce\}},$$

$$\mathsf{S} \triangleq ae \to \mathtt{Stop}_{\{ae,be\}}.$$

$a \triangleq \{ae\}$ and $b \triangleq \{be\}$ and we expect that both $\mathsf{R}$ and $\mathsf{S}$ exhibit the security property $a \not\rightsquigarrow b$, as neither mentions the event $be$.

Considering $\mathsf{S}$, which has traces $\langle\rangle$ and $\langle ae\rangle$, we see that the definition of non–interference gives $a \not\rightsquigarrow b : \mathsf{S}$, since $\langle ae\rangle|_a = \langle\rangle$.

But, considering $\mathsf{R}$ we find that the non–interference property does not hold because the trace $\langle ce\rangle = \langle ae, ce\rangle|_a$ is not in $\tau\mathsf{R}$.

The difference between the purged and unpurged traces, from $b$'s viewpoint, is made clearer by considering the processes resulting from performing the two traces: $\mathsf{R}/\langle ae, ce\rangle$ and $\mathsf{R}/\langle ce\rangle$. The former exists and is $\mathtt{Stop}_{\{ae,be,ce\}}$, but the latter is not a process as $\langle ce\rangle$ is not a trace of $\mathsf{R}$. From $b$'s viewpoint the difference is that in the former case the system is operating as expected, whereas in the latter the system is no longer operating; it is not defined. $\triangle$

Note that were we to *define* non–interference so that system $\mathsf{R}$ (in Example 3.6) is non–interfering, the parallel combination of $\mathsf{R}$ with another system $\mathsf{P}$ would no longer need to be non–interfering. It is, therefore, important to avoid such examples, and we do so by ensuring the definition of non–interference does not encompass them.

# Chapter 4

# Development

## 4.1 Introduction

This chapter splits into two parts: the first describes techniques for refinement which preserve non–interference; the second gives a collection of laws for use in developing non–interfering systems.

In Section 4.2.2 the techniques of refinement of functional properties are shown not to preserve security specifications. We do that by example, and then define *secure replacement*: preservation of the equivalence $\approx_v^{\mathsf{S}}$.

Non–interference is considered and specific conditions for that property to be preserved by refinement are given in Section 4.2.4 below.

In Section 4.3 we give laws (using CSP) for the construction of secure systems.

Non–interference concerns the effect of individual events from some user's interface on another user; sometimes, however, entire transactions (or sequences of events) may need to be considered as non–interfering with another user. That concept is defined, related to non–interference (as defined in Section 3.3) and a collection of laws concerning it are given in Section 4.3.

## 4.2 Preserving Security by Refinement

The purpose of this section is to show how refinement can be used in the development of secure systems. We define a relation between systems which expresses the relationship 'has the security properties of.'

In Section 4.2.2 the need for such refinement relations is shown by an example which illustrates why refinement is insufficient when considering security. Section 4.2.3 gives the basic definition of when

one system can be replaced by another preserving $\approx_v^{\mathsf{S}}$.

Section 4.2.1 describes a technique called 'unwinding,' (see [29]) which can be used to prove that a system is non–interfering.

### 4.2.1 Proving Security

The definition of non–interference given in Chapter 3 is over all sequences of events of a system, and so proving security requires reasoning over all the traces. It would be better to have a proof technique that incorporates induction over traces, removing the need to perform an induction each time a proof of security is required; such a technique called 'unwinding' is given in [29]. The use, in the proof of the unwinding theorem, of induction over possible sequences of events for a system simplifies the proof of non–interference, in a particular example, to checking proof rules that concern individual events.

Theorem 4.1 is our unwinding theorem for non–interference.

**Theorem 4.1** *If system* $\mathsf{S}$ *has disjoint users* $u$ *and* $v$ *and*

$$\forall\, t \in \tau\mathsf{S}\, \forall\, ue \in u \bullet (t ^\frown \langle ue \rangle) \approx_v^{\mathsf{S}} t \qquad (1)$$

$$\forall\, s, t \in \tau\mathsf{S}\, \forall\, e \in (\alpha\mathsf{S} \setminus v) \bullet s \approx_v^{\mathsf{S}} t \Rightarrow (s ^\frown \langle e \rangle) \approx_v^{\mathsf{S}} (t ^\frown \langle e \rangle) \qquad (2)$$

*then* $u \not\rightsquigarrow v : \mathsf{S}$.

**Proof** We prove $t \approx_v^{\mathsf{S}} t|_u$ by induction on the length of a trace $t \in \tau\mathsf{S}$.

**(base case)** $\langle\rangle|_u = \langle\rangle$, hence this case follows as $\approx_v^{\mathsf{S}}$ is reflexive (see Lemma 3.2).

**(inductive step)** Suppose that for some $n$

$$\forall\, t \in \tau\mathsf{S} \bullet \#t \leq n \Rightarrow t \approx_v^{\mathsf{S}} t|_u$$

and consider $e \in \alpha\mathsf{S}$ such that $t ^\frown \langle e \rangle \in \tau\mathsf{S}$.

**(case** $e \in u$**)** Condition (1) and the hypothesis give:

$$\begin{array}{lll} & ((t ^\frown \langle e \rangle) \approx_v^{\mathsf{S}} t) \wedge (t \approx_v^{\mathsf{S}} t|_u) & \\ \Rightarrow & (t ^\frown \langle e \rangle) \approx_v^{\mathsf{S}} t|_u & \text{[Lemma 3.2]} \\ \Rightarrow & (t ^\frown \langle e \rangle) \approx_v^{\mathsf{S}} (t ^\frown \langle e \rangle)|_u. & \text{[this case]} \end{array}$$

**(case** $e \notin u$**)** The hypothesis gives:

$$\begin{array}{lll} & (t \approx_v^{\mathsf{S}} t|_u) & \\ \Rightarrow & ((t ^\frown \langle e \rangle) \approx_v^{\mathsf{S}} t|_u{}^\frown \langle e \rangle) & \text{[condition (2)]} \\ \Rightarrow & (t ^\frown \langle e \rangle) \approx_v^{\mathsf{S}} (t ^\frown \langle e \rangle)|_u. & \text{[this case]} \end{array}$$

□

Condition (2) is not a necessary condition for non–interference. Example 4.1 illustrates that fact with an example of a non–interfering system in which condition (2) does not hold.

**Example 4.1** Consider the system

$$\mathsf{S} \triangleq b \to b \to ve \to \mathrm{Stop}_{\{b,ue,ve\}}$$

where $\alpha\mathsf{S} = \{b, ue, ve\}$ and user $u = \{ue\}$ and user $v = \{ve\}$. Clearly $u \not\rightsquigarrow v : \mathsf{S}$ (as there are no events from $u$ in $\mathsf{S}$—see Lemma 4.22), but condition (2) does not hold because $\langle\rangle \approx_v^{\mathsf{S}} \langle b\rangle$ but $\langle b\rangle \not\approx_v^{\mathsf{S}} \langle b, b\rangle$. △

Condition (1) is necessary; Lemma 4.1 shows why.

**Lemma 4.1** *If system* $\mathsf{S}$ *has users* $u$ *and* $v$ *such that* $u \not\rightsquigarrow v : \mathsf{S}$ *then condition* (1) *holds.*

**Proof** Suppose $t^\frown\langle ue\rangle$ is a trace of $\mathsf{S}$ such that $ue \in u$; then

$$\begin{array}{lll} & (t^\frown\langle ue\rangle \approx_v^{\mathsf{S}} (t^\frown\langle ue\rangle)|_u) \wedge (t \approx_v^{\mathsf{S}} t|_u) & [\text{since } u \not\rightsquigarrow v : \mathsf{S}] \\ \Rightarrow & (t^\frown\langle ue\rangle \approx_v^{\mathsf{S}} t|_u) \wedge (t \approx_v^{\mathsf{S}} t|_u) & [\text{since } (t^\frown\langle ue\rangle)|_u = t|_u] \\ \Rightarrow & (t^\frown\langle ue\rangle \approx_v^{\mathsf{S}} t). & [\text{transitivity—Lemma 3.2}] \end{array}$$

Hence condition (1) holds. □

## 4.2.2 Lack of Security

It has been noted by a number of authors (see, for example, [31, 32] and [47]) that the definitions of refinement in Section 2.4 do not preserve security properties. To illustrate that fact consider the following example.

**Example 4.2** Consider the system $\mathsf{A}$ which represents a pair of binary–valued variables. The system has two users $u$ and $v$ who each have exclusive access to their own variable (the state of the variables is given by the super- and subscripts on $\mathsf{A}$) through a single event that both updates the value in the variable and outputs the previous value.

Here users $u$ and $v$ are sharing access to a system which stores private information for them; the security requirement is that the users cannot communicate with one another.

$$
\begin{array}{lcl}
\mathsf{A} & \triangleq & \mathsf{A}_0^0 \\
\mathsf{A}_{vSt}^{uSt} & \triangleq & \displaystyle\mathop{[\!]}_{uSt' \in B} \; \mathop{\sqcap}_{b \in B} \; (uUpdate.(uSt', uSt, b) \rightarrow \mathsf{A}_{vSt}^{uSt'}) \\
& & [\!] \\
& & \displaystyle\mathop{[\!]}_{vSt' \in B} \; \mathop{\sqcap}_{b \in B} \; (vUpdate.(vSt', vSt, b) \rightarrow \mathsf{A}_{vSt'}^{uSt}),
\end{array}
$$

where $B \triangleq \{0, 1\}$.

The two users are $u \triangleq \{uUpdate.b \mid b \in B^3\}$ and $v \triangleq \{vUpdate.b \mid b \in B^3\}$. $u$'s state is the superscript (a binary value) and $v$'s the subscript. Initially both parts of the state are 0.

The users each have a single event; consider $u$, for example. $u$'s event is $uUpdate.(b_0, b_1, b_2)$ with three input/output parts: $b_0$ is the new value of $u$'s state; $b_1$ represents the previous value; and $b_2$ is unspecified and may take any value. Each instance of $u$'s event, effectively, updates the value of the state to $b_0$, 'outputting' the current value of the state.

Both $u \not\rightsquigarrow v : \mathsf{A}$ and $v \not\rightsquigarrow u : \mathsf{A}$ since neither event affects the values present in, or the possibility of the occurrence of, the other event. That fact is easily verified using the laws of Section 4.3:

**Lemma 4.2** $u \not\rightsquigarrow v : \mathsf{A}$ *and* $v \not\rightsquigarrow u : \mathsf{A}$.

**Proof** Both follow from Laws 4.1, 4.10, 4.11 and 4.14. □

Consider the system $\mathsf{C}$.

$$
\begin{array}{lcl}
\mathsf{C} & \triangleq & \mathsf{C}_0^0 \\
\mathsf{C}_{vSt}^{uSt} & \triangleq & \displaystyle\mathop{[\!]}_{uSt' \in B} \; (uUpdate.(uSt', uSt, vSt) \rightarrow \mathsf{C}_{vSt}^{uSt'}) \\
& & [\!] \\
& & \displaystyle\mathop{[\!]}_{vSt' \in B} \; (vUpdate.(vSt', vSt, uSt) \rightarrow \mathsf{C}_{vSt'}^{uSt})
\end{array}
$$

There the third component of the input/output in each event has been specified to tell each user the value of the other user's state. Then on the occurrence of each event a user is told the value of his state and the other user's state; updating the state is done as before. Evidently $\mathsf{C}$ no longer has the non–interference properties present in $\mathsf{A}$.

$\mathsf{C}$ is a refinement of $\mathsf{A}$ with respect to the ordering $\sqsubseteq$.

**Lemma 4.3** $\mathsf{A} \sqsubseteq \mathsf{C}$. □

Hence the refinement $\mathsf{A} \sqsubseteq \mathsf{C}$ does not preserve non–interference. △

Refinement of systems which have non–deterministic components is a common concern amongst the computer–security community, see, for example, comments in [57]. Non–determinism does not pose a problem in this thesis; indeed it is *useful* to be able to give non–deterministic specifications. We ensure that correct refinement takes place by the definition of the ordering on systems given in Section 4.2.3.

### 4.2.3 Secure Replacement

Section 4.2.1 shows how to prove that a system meets a security specification and Section 4.2.2 shows that refinement can introduce security breaches. We define what it means for one system to replace another preserving the security properties of the first system. A similar approach is taken by Jacob in [44] (and in [50]); a comparison of our approach with his is given in Section 7.5.

The condition for replacement of one process by another is given in terms of the equivalence relations $\approx_v^{\mathsf{S}}$ and $\approx_v^{\mathsf{T}}$ for a user $v$. $\mathsf{S}$ can be replaced by $\mathsf{T}$ if $v$ cannot distinguish traces of $\mathsf{T}$ that were indistinguishable as traces of $\mathsf{S}$. We insist that the systems $\mathsf{S}$ and $\mathsf{T}$ have the same alphabet so that they are comparable by the equivalence relations $\approx_v^{\mathsf{S}}$ and $\approx_v^{\mathsf{T}}$, as those relations are defined on all sequences of events from some alphabet.

**Definition 4.1** *System* $\mathsf{T}$ *is a* secure replacement *for system* $\mathsf{S}$ *for user* $v$ *when* $\alpha\mathsf{S} = \alpha\mathsf{T}$ *and*

$$\approx_v^{\mathsf{S}} \subseteq \approx_v^{\mathsf{T}} .$$

*That is written* $\mathsf{S} \preceq_v \mathsf{T}$. △

We have not yet made clear the relationship between secure replacement and refinement. Section 4.2.2 showed that refinement does not imply secure replacement; equally a secure replacement does not imply refinement. Refinement and secure replacement are opposing concerns (consider the fact that in Example 4.4 $\mathtt{Stop}$ is a secure replacement of the live process $ue \rightarrow \mathtt{Stop}$ ).

We give a weaker version of secure replacement, *semi–secure replacement*, which Lemma 4.5 shows to be equivalent to secure replacement when system $\mathsf{T}$ is a refinement of $\mathsf{S}$; the relationship between refinement and semi–secure replacement is demonstrated by Lemma 4.4.

**Definition 4.2** *System* $\mathsf{T}$ *is a* semi–secure replacement *for system* $\mathsf{S}$ *and user* $v$ *(written* $\mathsf{S} \; \underline{\subset}_v \; \mathsf{T}$*) when* $\alpha\mathsf{S} = \alpha\mathsf{T}$ *and*

$$(\approx_v^{\mathsf{S}} \diamond \tau\mathsf{S}) \subseteq \approx_v^{\mathsf{T}} .$$ △

$\preceq_v$ is stronger than $\underline{\subset}_v$.

**Lemma 4.4** *If* $\mathsf{S} \preceq_v \mathsf{T}$ *then* $\mathsf{S}\ \underline{\subset}_v\ \mathsf{T}$.

**Proof**
$$\begin{array}{llll} \mathsf{S} \preceq_v \mathsf{T} & \Leftrightarrow & \approx_v^{\mathsf{S}} \subseteq \approx_v^{\mathsf{T}} & \text{[Definition 4.1]} \\ & \Rightarrow & (\approx_v^{\mathsf{S}} \diamond \tau\mathsf{S}) \subseteq \approx_v^{\mathsf{T}} & \text{[Definition 2.1]} \\ & \Leftrightarrow & \mathsf{S}\ \underline{\subset}_v\ \mathsf{T}. & \text{[Definition 4.2]} \end{array}$$
□

Lemma 4.5 shows that refinement and semi–secure replacement are enough to ensure secure replacement. That is, if T is a refinement *and* a semi–secure replacement of S then Lemma 4.5 holds and hence semi–secure replacement may be used instead of secure replacement.

**Lemma 4.5** *If* $\mathsf{S} \sqsubseteq \mathsf{T}$ *then* $\mathsf{S}\ \underline{\subset}_v\ \mathsf{T}$ *and* $\mathsf{S} \preceq_v \mathsf{T}$ *are equivalent.*

**Proof** Lemma 4.4 shows that $\mathsf{S} \preceq_v \mathsf{T}$ implies $\mathsf{S}\ \underline{\subset}_v\ \mathsf{T}$; it remains to prove the converse.

Suppose $s \approx_v^{\mathsf{S}} t$ for some $s$ and $t$ in $\alpha\mathsf{S}^*$. If $s$ and $t$ are in $\tau\mathsf{S}$ then $s \approx_v^{\mathsf{T}} t$ follows from the fact that $\mathsf{S}\ \underline{\subset}_v\ \mathsf{T}$. If neither $s$ nor $t$ are in $\tau\mathsf{S}$ then neither of them are in $\tau\mathsf{T}$ (since $\mathsf{S} \sqsubseteq \mathsf{T}$) and so $s \approx_v^{\mathsf{T}} t$.

Consider the case when $s \in \tau\mathsf{S}$ and $t \notin \tau\mathsf{S}$ (and hence $t \notin \tau\mathsf{T}$). If $s \notin \tau\mathsf{T}$ then $s \approx_v^{\mathsf{T}} t$, so consider the case when $s \in \tau\mathsf{T}$ and suppose that $s \not\approx_v^{\mathsf{T}} t$. Thus there must exist $r \in v^*$ such that $(s^\frown r) \in \tau\mathsf{T}$ and $(t^\frown r) \notin \tau\mathsf{T}$. Hence $(s^\frown r) \in \tau\mathsf{S}$ (since $\mathsf{S} \sqsubseteq \mathsf{T}$) and as $s \approx_v^{\mathsf{S}} t$ we must have $(t^\frown r) \in \tau\mathsf{S}$, but $t$ is not a trace of S. Therefore, by contradiction, $s \approx_v^{\mathsf{T}} t$. □

To verify that T is a secure replacement for S for all its users requires checking that $\mathsf{S} \preceq_v \mathsf{T}$ for all $v$.

**Definition 4.3** *If system* S *has a set of users* $U$ *and if* $\alpha\mathsf{S} = \alpha\mathsf{T}$ *then* T *is a* secure replacement *for* S *iff*

$$\forall\, v \in U \bullet \mathsf{S} \preceq_v \mathsf{T}.$$

*We write* $\mathsf{S} \preceq^U \mathsf{T}$ *or* $\mathsf{S} \preceq \mathsf{T}$ *when the definition of* $U$ *is understood from the context.* △

A similar definition is made in the case of $\underline{\subset}_v$.

**Definition 4.4** *If system* S *has users* $u \in U$, *for some set* $U$, *and* $\alpha\mathsf{S} = \alpha\mathsf{T}$ *then* T *is a* semi–secure replacement *for* S *iff*

$$\forall\, v \in U \bullet \mathsf{S}\ \underline{\subset}_v\ \mathsf{T}.$$

*We write* $\mathsf{S} \ \underline{\subset}^U \ \mathsf{T}$ *or* $\mathsf{S} \ \underline{\subset} \ \mathsf{T}$ *when the definition of* $U$ *is understood from the context.* △

Definition 4.2 can be simplified when $\mathsf{T}$ is a refinement of $\mathsf{S}$: the inclusion on the equivalence relations $\approx_v^{\mathsf{S}}$ and $\approx_v^{\mathsf{T}}$ need only be checked for elements of $\tau\mathsf{S}$ (see Lemma 4.5); Lemma 4.6 expresses $\underline{\subset}_u$ in terms of equivalence classes of traces.

**Lemma 4.6** $\mathsf{S} \ \underline{\subset}_v \ \mathsf{T}$ *iff* $\forall\, t \in \tau\mathsf{S} \bullet [t]_v^{\mathsf{S}} \subseteq [t]_v^{\mathsf{T}}$.

**Proof**

$$
\begin{array}{lll}
 & (\mathsf{S} \ \underline{\subset}_v \ \mathsf{T}) & \\
\Leftrightarrow & (\approx_v^{\mathsf{S}} \diamond \tau\mathsf{S}) \subseteq \approx_v^{\mathsf{T}} & \text{[Definition 4.2]} \\
\Leftrightarrow & \forall\, s, t \in \tau\mathsf{S} \bullet (s \approx_v^{\mathsf{S}} t) \Rightarrow (s \approx_v^{\mathsf{T}} t) & \text{[Definition 2.1]} \\
\Leftrightarrow & \forall\, s, t \in \tau\mathsf{S} \bullet (s \in [t]_v^{\mathsf{S}}) \Rightarrow (s \in [t]_v^{\mathsf{T}}) & \\
\Leftrightarrow & \forall\, t \in \tau\mathsf{S} \bullet [t]_v^{\mathsf{S}} \subseteq [t]_v^{\mathsf{T}}. &
\end{array}
$$

□

We define $\mathcal{C}_v^{\mathsf{S}}$, the set of equivalence classes of traces under $\approx_v^{\mathsf{S}}$.

**Definition 4.5** *If system* $\mathsf{S}$ *has user* $v$ *then the set of equivalence classes of traces under* $\approx_v^{\mathsf{S}}$ *is* $\mathcal{C}_v^{\mathsf{S}} \triangleq \{[t]_v^{\mathsf{S}} \mid t \in \alpha\mathsf{S}^*\}$. △

Using Lemma 4.6 we now prove that if there is a function which maps the classes of $\mathsf{S}$ to the classes of $\mathsf{T}$ then $\mathsf{S}$ is a semi–secure replacement of $\mathsf{T}$. We instantiate the function $\theta_v$ in Section A.2.3 to demonstrate how secure replacement is related to downwards simulation.

**Theorem 4.2** *If* $\theta_v : \mathcal{C}_v^{\mathsf{S}} \to \mathcal{C}_v^{\mathsf{T}}$ *such that* $(t \in \tau\mathsf{S})\ \theta_v([t]_v^{\mathsf{S}}) = [t]_v^{\mathsf{T}}$ *then* $\mathsf{S} \ \underline{\subset}_v \ \mathsf{T}$.

**Proof** Consider traces $s$ and $t$ of $\mathsf{S}$.

$$
\begin{array}{llll}
s \in [t]_v^{\mathsf{S}} & \Leftrightarrow & [s]_v^{\mathsf{S}} = [t]_v^{\mathsf{S}} & \\
 & \Rightarrow & \theta_v([s]_v^{\mathsf{S}}) = \theta_v([t]_v^{\mathsf{S}}) & [\theta_v \text{ a function}] \\
 & \Rightarrow & [s]_v^{\mathsf{T}} = [t]_v^{\mathsf{T}} & [\text{definition of } \theta_v] \\
 & \Leftrightarrow & s \in [t]_v^{\mathsf{T}}. &
\end{array}
$$

Hence $[t]_v^{\mathsf{S}} \subseteq [t]_v^{\mathsf{T}}$ and so $\mathsf{S} \ \underline{\subset}_v \ \mathsf{T}$ by Lemma 4.6. □

It is trivial to show that $\preceq_v$ and $\preceq$ are preorders and that $\underline{\subset}_v$ and $\underline{\subset}$ are reflexive.

**Lemma 4.7** $\preceq_v$ *and* $\preceq$ *are preorders;* $\underline{\subset}_v$ *and* $\underline{\subset}$ *are reflexive.* □

$\preceq_v$ (and hence $\preceq$), however, is not a partial order; the following example illustrates that fact, by showing that the order is not anti–symmetric. Similarly for $\underline{\subset}_v$ and $\underline{\subset}$.

**Example 4.3** Consider the systems

$$\mathsf{S} \triangleq (ue \rightarrow \mathtt{Stop}_{\{ue,ve\}}) \Box (ve \rightarrow \mathtt{Stop}_{\{ue,ve\}})$$

and

$$\mathsf{T} \triangleq (ue \rightarrow \mathtt{Stop}_{\{ue,ve\}}) \sqcap (ve \rightarrow \mathtt{Stop}_{\{ue,ve\}})$$

with user $u \triangleq \{ue\}$. Both $\mathsf{S} \preceq_u \mathsf{T}$ and $\mathsf{T} \preceq_u \mathsf{S}$ but $\mathsf{S} \neq \mathsf{T}$. Hence $\preceq_u$ is not anti–symmetric. Similarly for $\underline{\subset}_v$ and $\underline{\subset}$. △

$\underline{\subset}_v$ (and hence $\underline{\subset}$) is intransitive; consider the following example.

**Example 4.4** Consider the systems $\mathsf{R}$, $\mathsf{S}$ and $\mathsf{T}$.

$$\begin{array}{lcl} \mathsf{R} & \triangleq & ue \rightarrow \mathtt{Stop}_{\{ue,ve\}} \\ \mathsf{S} & \triangleq & \mathtt{Stop}_{\{ue,ve\}} \\ \mathsf{T} & \triangleq & ue \rightarrow ve \rightarrow \mathtt{Stop}_{\{ue,ve\}} \end{array}$$

Each have users $u \triangleq \{ue\}$ and $v \triangleq \{ve\}$. $\mathsf{R} \ \underline{\subset}_v \ \mathsf{S}$ and $\mathsf{S} \ \underline{\subset}_v \ \mathsf{T}$, but $\mathsf{R}$ is not a semi–secure replacement of $\mathsf{T}$ since $\langle\rangle \approx_v^{\mathsf{R}} \langle ue\rangle$ whereas $\langle\rangle \not\approx_v^{\mathsf{T}} \langle ue\rangle$.

Hence $\underline{\subset}_v$ (and therefore $\underline{\subset}$) is not transitive. △

Although $\underline{\subset}_v$ is intransitive (and therefore not useful when performing a chain of refinements) Lemmas 4.5 and 4.7 show that $\underline{\subset}_v$ and $\sqsubseteq$ together do form a preorder and hence can be used as a refinement relation.

**Lemma 4.8** *$\mathtt{Run}_A$ and $\mathtt{Chaos}_A$ are tops of $\preceq_v$.*

**Proof** $\approx_v^{\mathrm{Run}_A} = \approx_v^{\mathrm{Chaos}_A} = A^* \times A^*$ and as for any process $\mathsf{S}$ we have $\approx_v^{\mathsf{S}} \subseteq A^* \times A^*$ both $\mathtt{Run}_A$ and $\mathtt{Chaos}_A$ are tops of $\preceq_v$. □

A direct consequence of Lemma 4.4 and Lemma 4.8 is that $\mathtt{Run}_A$ and $\mathtt{Chaos}_A$ are also tops of $\underline{\subset}_v$.

**Lemma 4.9** *$\mathtt{Run}_A$ and $\mathtt{Chaos}_A$ are tops of $\underline{\subset}_v$.* □

Neither order has a bottom element; there is no process $\mathsf{S}$ for which $\approx_v^{\mathsf{S}} = \emptyset$. That is because $\approx_v^{\mathsf{S}}$ is reflexive (see Lemma 3.2) and so (at least) $\langle\rangle \approx_v^{\mathsf{S}} \langle\rangle$ and so $\approx_v^{\mathsf{S}} \neq \emptyset$.

### 4.2.4 Refining Non–interference

Lemma 4.10 shows that semi–secure replacement and refinement guarantee preservation of non–interference.

**Lemma 4.10** *If* $u \not\rightsquigarrow v : \mathsf{S}$, $\mathsf{S} \underline{\subset}_v \mathsf{T}$ *and* $\mathsf{S} \sqsubseteq \mathsf{T}$ *then* $u \not\rightsquigarrow v : \mathsf{T}$.

**Proof**
$$
\begin{array}{ll}
 & (u \not\rightsquigarrow v : \mathsf{S}) \wedge (\mathsf{S} \underline{\subset}_v \mathsf{T}) \\
\Leftrightarrow & (\forall\, t \in \tau\mathsf{S} \bullet t \approx_v^{\mathsf{S}} t|_u) \wedge (\approx_v^{\mathsf{S}} \diamond \tau\mathsf{S} \subseteq \approx_v^{\mathsf{T}}) \\
 & \qquad \text{[Definitions 3.9 and 4.2]} \\
\Rightarrow & \forall\, t \in \tau\mathsf{S} \bullet t \approx_v^{\mathsf{T}} t|_u \\
\Rightarrow & \forall\, t \in \tau\mathsf{T} \bullet t \approx_v^{\mathsf{T}} t|_u \qquad [\mathsf{S} \sqsubseteq \mathsf{T}] \\
\Leftrightarrow & u \not\rightsquigarrow v : \mathsf{T}. \qquad \text{[Definition 3.9]} \quad \square
\end{array}
$$

Although Definition 4.2 (secure replacement) is sufficient to show that if $u \not\rightsquigarrow v : \mathsf{S}$ and $\mathsf{T}$ is a secure replacement of $\mathsf{S}$ then $\mathsf{T}$ is non–interfering, semi–secure replacement alone is not enough to guarantee $u \not\rightsquigarrow v : \mathsf{T}$.

Consider the following example.

**Example 4.5** Consider the systems $\mathsf{S}$ and $\mathsf{T}$ of Example 4.4 with users $u \triangleq \{ue\}$ and $v \triangleq \{ve\}$. Both the systems have the alphabet $\{ue, ve\}$.

$$
\begin{array}{lcl}
\mathsf{S} & \triangleq & \mathrm{Stop}_{\{ue,ve\}} \\
\mathsf{T} & \triangleq & ue \rightarrow ve \rightarrow \mathrm{Stop}_{\{ue,ve\}}
\end{array}
$$

Clearly $u \not\rightsquigarrow v : \mathsf{S}$ (see Lemma 4.22, part 3) and $u$ does interfere with $v$ in $\mathsf{T}$. $\mathsf{S} \underline{\subset}_v \mathsf{T}$ is seen by considering the following table of equivalence classes.

| $v$–classes in $\mathsf{S}$ | $v$–classes in $\mathsf{T}$ |
|---|---|
| $\langle\rangle$ | $\langle\rangle$ $\langle ue, ve\rangle$ |
| | $\langle ue\rangle$ |
| Rest of $\alpha\mathsf{S}^*$ | Rest of $\alpha\mathsf{T}^*$ |

Thus semi–secure replacement is not sufficient to establish preservation of non–interference. △

In Appendix A semi–secure replacement and refinement are considered together giving examples using a state–based refinement method. That

appendix shows that the results of this section are not restricted to refinement in CSP.

## 4.3 Laws of CSP

Section 3.3 showed how to specify non–interference. This section gives laws of CSP operators which may be used to develop systems satisfying non–interference.

Section 4.3.3 gives laws for a generalized version of non–interference (transaction non–interference) which is defined in Section 4.3.2. In Section 4.3.2 we show that non–interference is a special case of that definition.

We outline some useful properties of the equivalence relations $\approx_u^{\mathsf{S}}$ and $\equiv_u^A$ in Section 4.3.1.

Throughout the sections we refer to laws from [39] using the notation [Lx:pp] to mean law x on page pp; semantic definitions are referred to using the notation [Dx:pp] (definition x on page pp).

### 4.3.1 Properties of $\approx_u^{\mathsf{S}}$ and $\equiv_u^A$

This section gives some of the properties of the equivalence relations $\approx_u^{\mathsf{S}}$ and $\equiv_u^A$ which are useful for proving the laws given in Section 4.3.3.

If systems $\mathsf{S}$ and $\mathsf{T}$ appear identical to the user whose interface is the entire alphabet then the systems are identical.

**Lemma 4.11** *If systems* $\mathsf{S}$ *and* $\mathsf{T}$ *have alphabets* $\alpha\mathsf{S} = \alpha\mathsf{T} = A$ *then* $\mathsf{S} = \mathsf{T}$ *iff* $\mathsf{S} \equiv_A^A \mathsf{T}$.

**Proof**
$$\begin{array}{rlll} \mathsf{S} \equiv_A^A \mathsf{T} & \Leftrightarrow & \mathsf{S} \| \overline{A}_A = \mathsf{T} \| \overline{A}_A & \text{[Definition 3.6]} \\ & \Leftrightarrow & \mathsf{S} \| \mathrm{Stop}_{\emptyset} = \mathsf{T} \| \mathrm{Stop}_{\emptyset} & [\overline{A}_A = \mathrm{Stop}_{\emptyset}] \\ & \Leftrightarrow & \mathsf{S} = \mathsf{T}. & [\mathrm{Stop}_{\emptyset} \text{ is a unit of } \|] \end{array}$$
□

Equivalence from the view of a user $v$ is preserved by parallel composition.

**Lemma 4.12** *If systems* $\mathsf{R}$, $\mathsf{S}$ *and* $\mathsf{T}$ *have alphabet* $A$ *and user* $u$, *and* $\mathsf{S} \equiv_v^A \mathsf{T}$ *then* $(\mathsf{S} \| \mathsf{R}) \equiv_v^A (\mathsf{T} \| \mathsf{R})$.

**Proof**
$$\begin{array}{rlll} \mathsf{S} \equiv_v^A \mathsf{T} & \Leftrightarrow & \mathsf{S} \| \overline{v}_A = \mathsf{T} \| \overline{v}_A & \text{[Definition 3.6]} \\ & \Rightarrow & (\mathsf{S} \| \overline{v}_A) \| \mathsf{R} = (\mathsf{T} \| \overline{v}_A) \| \mathsf{R} & \\ & \Rightarrow & (\mathsf{S} \| \mathsf{R}) \| \overline{v}_A = (\mathsf{T} \| \mathsf{R}) \| \overline{v}_A & \text{[L2:70]} \\ & \Leftrightarrow & (\mathsf{S} \| \mathsf{R}) \equiv_v^A (\mathsf{T} \| \mathsf{R}). & \text{[Definition 3.6]} \end{array}$$
□

Lemma 4.13 shows that $v$–equivalence is monotonic with respect to non–deterministic choice.

**Lemma 4.13** *If systems* $\mathsf{P}$, $\mathsf{Q}$, $\mathsf{S}$ *and* $\mathsf{T}$ *have alphabet* $A$ *and user* $u$, *and* $\mathsf{P} \equiv_v^A \mathsf{Q}$ *and* $\mathsf{S} \equiv_v^A \mathsf{T}$ *then* $(\mathsf{P} \sqcap \mathsf{S}) \equiv_v^A (\mathsf{Q} \sqcap \mathsf{T})$.

**Proof**
$$
\begin{aligned}
(\mathsf{P} \sqcap \mathsf{S}) \| \overline{v}_A &= (\mathsf{P} \| \overline{v}_A) \sqcap (\mathsf{S} \| \overline{v}_A) && \text{[L7:103]} \\
&= (\mathsf{Q} \| \overline{v}_A) \sqcap (\mathsf{T} \| \overline{v}_A) && [\mathsf{P} \equiv_v^A \mathsf{Q} \text{ and } \mathsf{S} \equiv_v^A \mathsf{T}] \\
&= (\mathsf{Q} \sqcap \mathsf{T}) \| \overline{v}_A. && \text{[L7:103]}
\end{aligned}
$$

Hence $(\mathsf{P} \sqcap \mathsf{S}) \equiv_v^A (\mathsf{Q} \sqcap \mathsf{T})$, by Definition 3.6. □

Changing the alphabet of a system (see [39, section 2.6]) changes the interfaces of its users and alters the equivalence relation $\equiv_u^A$ in the manner described by Lemma 4.14.

**Lemma 4.14** *If* $\mathsf{S}$ *and* $\mathsf{T}$ *have user* $u$ *and alphabets* $A$ *and* $B$ *respectively,* $f : A \rightarrow B$ *is such that* $f(A) = B$ *then* $\mathsf{S} \equiv_u^A \mathsf{T} \Rightarrow f(\mathsf{S}) \equiv_v^B f(\mathsf{T})$ *where* $f(u) = v$.

**Proof**
$$
\begin{aligned}
\mathsf{S} \equiv_u^A \mathsf{T} &\Leftrightarrow \mathsf{S} \| \overline{u}_A = \mathsf{T} \| \overline{u}_A && \text{[Definition 3.6]} \\
&\Rightarrow f(\mathsf{S} \| \overline{u}_A) = f(\mathsf{T} \| \overline{u}_A) && [f \text{ a function}] \\
&\Leftrightarrow f(\mathsf{S}) \| f(\overline{u}_A) = f(\mathsf{T}) \| f(\overline{u}_A) && \text{[L3:85]} \\
&\Leftrightarrow f(\mathsf{S}) \| \overline{v}_B = f(\mathsf{T}) \| \overline{v}_B && \text{[L1:84]} \\
&\Leftrightarrow f(\mathsf{S}) \equiv_v^B f(\mathsf{T}). && \text{[Definition 3.6]}
\end{aligned}
$$
□

Lemma 4.15 demonstrates the relationship between $\equiv_v^A$ and hiding ($\setminus$); it shows the effect of hiding events from $v$ on $\equiv_v^A$.

**Lemma 4.15** *If systems* $\mathsf{S}$ *and* $\mathsf{T}$ *have alphabet* $A$, *user* $v$, $C \subseteq u$ *and* $\mathsf{S} \equiv_v^A \mathsf{T}$ *then* $(\mathsf{S} \setminus C) \equiv_{v \setminus C}^{A \setminus C} (\mathsf{T} \setminus C)$.

**Proof**
$$
\begin{aligned}
& \mathsf{S} \equiv_v^A \mathsf{T} \\
\Leftrightarrow\ & (\mathsf{S} \| \overline{v}_A) = (\mathsf{T} \| \overline{v}_A) && \text{[Definition 3.6]} \\
\Rightarrow\ & (\mathsf{S} \| \overline{v}_A) \setminus C = (\mathsf{T} \| \overline{v}_A) \setminus C \\
\Leftrightarrow\ & (\mathsf{S} \setminus C) \| (\overline{v}_A \setminus C) = (\mathsf{T} \setminus C) \| (\overline{v}_A \setminus C) \\
& && [\text{L6:112 and } A \cap (A \setminus v) \cap C = \emptyset] \\
\Leftrightarrow\ & (\mathsf{S} \setminus C) \| \overline{v \setminus C}_{A \setminus C} = (\mathsf{T} \setminus C) \| \overline{v \setminus C}_{A \setminus C} && \text{[L4:112]} \\
\Leftrightarrow\ & (\mathsf{S} \setminus C) \equiv_{v \setminus C}^{A \setminus C} (\mathsf{T} \setminus C). && \text{[Definition 3.6]}
\end{aligned}
$$
□

Note the following properties of $\overline{v}_A$ used in the proofs of laws of parallel composition given later:

**Lemma 4.16** *If user $v$ is a subset of $A$ then $\overline{v}_A \| \overline{v}_A = \overline{v}_A$.*

**Proof** $\overline{v}_A \| \overline{v}_A$

$$
\begin{aligned}
&= \mathrm{Stop}_{A \setminus v} \| \mathrm{Stop}_{A \setminus v} && [\text{Definition 3.5}] \\
&= \mathrm{Stop}_{A \setminus v} && [\mathrm{Stop}_X \| \mathrm{Stop}_Y = \mathrm{Stop}_{X \cup Y}] \\
&= \overline{v}_A. && [\text{Definition 3.5}] \quad \square
\end{aligned}
$$

**Lemma 4.17** *If user $v$ is a subset of both $A$ and $B$ then $\overline{v}_{A \cup B} = \overline{v}_{(A \cup B)} \| \overline{v}_A = \overline{v}_{(A \cup B)} \| \overline{v}_B$.*

**Proof** $\overline{v}_{(A \cup B)} \| \overline{v}_A$

$$
\begin{aligned}
&= \mathrm{Stop}_{(A \cup B) \setminus v} \| \mathrm{Stop}_{A \setminus v} && [\text{Definition 3.5}] \\
&= \mathrm{Stop}_{(A \setminus v) \cup (B \setminus v)} \| \mathrm{Stop}_{A \setminus v} && [\text{De Morgan}] \\
&= \mathrm{Stop}_{(A \setminus u) \cup (B \setminus u)} && [\mathrm{Stop}_X \| \mathrm{Stop}_Y = \mathrm{Stop}_{X \cup Y}] \\
&= \mathrm{Stop}_{(A \cup B) \setminus v} && [\text{De Morgan}] \\
&= \overline{v}_{(A \cup B)}. && [\text{Definition 3.5}]
\end{aligned}
$$

Hence $\overline{v}_{A \cup B} = \overline{v}_{(A \cup B)} \| \overline{v}_A$ and $\overline{v}_{A \cup B} = \overline{v}_{(A \cup B)} \| \overline{v}_B$ follows similarly. $\square$

**Lemma 4.18** *If user $u$ is a subset of $A$, user $v$ is a subset of $B$ and $A \cap B = \emptyset$ then $\overline{u \cup v}_{(A \cup B)} = \overline{u \cup v}_{(A \cup B)} \| \overline{u}_A = \overline{u \cup v}_{(A \cup B)} \| \overline{v}_B$.*

**Proof** $\overline{u \cup v}_{(A \cup B)} \| \overline{u}_A$

$$
\begin{aligned}
&= \mathrm{Stop}_{(A \cup B) \setminus (u \cup v)} \| \mathrm{Stop}_{A \setminus u} && [\text{Definition 3.5}] \\
&= \mathrm{Stop}_{(A \setminus u) \cup (B \setminus v)} \| \mathrm{Stop}_{A \setminus u} && [A \cap B = \emptyset] \\
&= \mathrm{Stop}_{(A \setminus u) \cup (B \setminus v)} && [\mathrm{Stop}_X \| \mathrm{Stop}_Y = \mathrm{Stop}_{X \cup Y}] \\
&= \mathrm{Stop}_{(A \cup B) \setminus (u \cup v)} && [A \cap B = \emptyset] \\
&= \overline{u \cup v}_{(A \cup B)}. && [\text{Definition 3.5}]
\end{aligned}
$$

Similarly $\overline{u \cup v}_{(A \cup B)} = \overline{u \cup v}_{(A \cup B)} \| \overline{v}_B$. $\square$

Lemma 4.19 is useful when considering recursively defined systems. It shows how $v$–equivalence for each $\mathsf{F}^n(\mathrm{Chaos}_A)$ is related to the fixed–point of the $\mathsf{F}^n(\mathrm{Chaos}_A)$.

**Lemma 4.19** *If $\mathsf{F}$ is guarded and defined on alphabet $A$ with user $v$ then, putting $\mathsf{Z} = \bigsqcup_{n \geq 0} \mathsf{F}^n(\mathrm{Chaos}_A)$,*

$$\bigcap_{n \geq 0} \approx_v^{\mathsf{F}^n(\mathrm{Chaos}_A)} \subseteq \approx_v^{\mathsf{Z}} .$$

**Proof** The proof is given in the semantics of CSP using the failures–divergences model. Note that in the failures–divergences model $s \approx_v^{\mathsf{S}} t$ for system $\mathsf{S}$ with user $v$ and traces $s$ and $t$ iff

$$\forall r \in v^* \, \exists X, Y \bullet (s ^\frown r, X) \in \phi\mathsf{S} \Leftrightarrow (t ^\frown r, Y) \in \phi\mathsf{S} \wedge X \upharpoonright v = Y \upharpoonright v$$

and $\forall\, r \in v^* \bullet s^\frown r \in \delta\mathsf{S} \equiv t^\frown r \in \delta\mathsf{S}$.

Put $Y_n = \approx_v^{\mathsf{F}^n(\mathrm{Chaos}_A)}$ and $F_n = \phi\mathsf{F}^n(\mathrm{Chaos}\ _A)$.

$$
\begin{array}{ll}
& s\ (\bigcap_{n\geq 0} Y_n)\ t \\
\Leftrightarrow & \forall\, n \in \mathbb{N} \bullet s\ \ Y_n\ \ t \\
\Leftrightarrow & \forall\, n \in \mathbb{N}\, \forall\, r \in v^*\, \exists\, X, Y \bullet (s^\frown r, X) \in F_n \equiv (t^\frown r, Y) \in F_n \\
& \qquad \wedge \\
& \qquad (X \upharpoonright u = Y \upharpoonright u) \\
\Rightarrow & \forall\, r \in v^*\, \exists\, X, Y \bullet\ (s^\frown r, X) \in \bigcap_{n\geq 0} F_n \equiv (t^\frown r, Y) \in \bigcap_{n\geq 0} F_n \\
& \qquad \wedge \\
& \qquad (X \upharpoonright v = Y \upharpoonright v).
\end{array}
$$

Similarly for the divergences and so $s\ (\approx_u^{\mathsf{Z}})\ t$. □

## 4.3.2 Transaction Non–interference

The definition of non–interference (Definition 3.9) can be extended as follows. Suppose that rather than wishing that each individual event (belonging to a user $u$) does not interfere with another user (call that user $v$), we wish the system (for which we wish to prove non–interference) to allow certain sequences of events (which we call transactions) for user $u$. We define transaction non–interference by saying that once a transaction has completed—we define completion below—the system must appear in an identical state to user $v$ as when none of the events from the transaction have occurred.

That is, we say that complete transactions do not interfere with a user; transactions that have not completed may interfere (by synchronization, for example). Consider Example 4.6.

**Example 4.6** System $\mathsf{R}$ is defined:

$$\mathsf{R} \ \triangleq\ (u.start \to u.stop \to \mathsf{R} \mid ve \to \mathsf{R}).$$

If $u \triangleq \{u.start, u.stop\}$ and $v \triangleq \{ve\}$ then $u$ interferes with $v$ in $\mathsf{R}$ since the traces $\langle u.start \rangle$ and $\langle\rangle (= \langle u.start \rangle|_u)$ are not $v$–equivalent.

But suppose that we wished to assert that $u \not\rightsquigarrow v : \mathsf{R}$ by considering the 'transaction' $\langle u.start, u.stop \rangle$ as atomic (as a single event: $u.transaction$) then $\mathsf{R}$ would be like:

$$\mathsf{R}' \ \triangleq\ (u.transaction \to \mathsf{R}' \mid ve \to \mathsf{R}'),$$

and $\{u.transaction\} \not\rightsquigarrow v : \mathsf{R}'$ (by Laws 4.1 and 4.14).

We can achieve that by noting that $\langle u.start, u.stop\rangle$ is $v$–equivalent to $\langle u.start, u.stop\rangle|_u = \langle\rangle$, and by saying that $u$ is transaction non–interfering with $v$ when

$$\forall\, t \in \tau\mathsf{R}\, \exists\, r \bullet t^\frown r \approx_v^{\mathsf{R}} t|_u.$$

$r$ is the necessary extension to a trace to 'complete' a transaction—in this case the extension of $\langle u.start\rangle$ would be $\langle u.stop\rangle$. △

So we define transaction non–interference by defining the users of a system and the set of completions for transactions.

The set of completions $T$ is suffix–closed: a set $S$ of traces is *suffix–closed* if $\forall\, s^\frown t \in S \bullet t \in S$. That is because we require that at any stage a transaction can be completed to the point where it has not caused a change in the view the other user has of the system.

**Definition 4.6** *If system* $\mathsf{S}$ *has disjoint users* $u$ *and* $v$ *and a suffix–closed set of completions* $T \subseteq u^*$ *then* $u$ *is* transaction non–interfering *with* $v$ *iff*

$$\forall\, t \in \tau\mathsf{S}\, \exists\, r \in T \bullet (t^\frown r) \approx_v^{\mathsf{S}} t|_u.$$

*That is written* $T, u \not\rightsquigarrow v : \mathsf{S}$. △

When a trace $r$ from $T$ is a suitable completion for trace $t$ we say that $r$ *completes* $t$ and write $\mathbf{com}_{\mathsf{S}}(t, r)$. Definition 4.7 gives that formally.

**Definition 4.7** *If* $T, u \not\rightsquigarrow v : \mathsf{S}$ *then* $r \in T$ completes $t \in \tau\mathsf{S}$ *iff*

$$(t^\frown r) \approx_v^{\mathsf{S}} t|_u.$$

*That is written* $\mathbf{com}_{\mathsf{S}}(t, r, u, v)$ *or just* $\mathbf{com}_{\mathsf{S}}(t, r)$. △

**Example 4.7** Consider system $\mathsf{R}$ of Example 4.6. The transaction non–interference assertion required is $\{\langle\rangle, \langle u.stop\rangle\}, u \not\rightsquigarrow v : \mathsf{R}$. △

Transaction non–interference is closely related to non–interference and Lemma 4.20 shows that if the set of completions is just the empty trace then transaction non–interference and non–interference are equivalent.

**Lemma 4.20** *If system* $\mathsf{S}$ *has users* $u$ *and* $v$ *then*

$$\{\langle\rangle\}, u \not\rightsquigarrow v : \mathsf{S} \Leftrightarrow u \not\rightsquigarrow v : \mathsf{S}.$$

**Proof** $\{\langle\rangle\}, u \not\rightsquigarrow v : \mathsf{S}$

$$\begin{array}{lll}
\Leftrightarrow & \forall\, t \in \tau\mathsf{S}\, \exists\, r \in \{\langle\rangle\} \bullet (t^\frown r) \approx_v^{\mathsf{S}} t|_u & \text{[Definition 4.6]} \\
\Leftrightarrow & \forall\, t \in \tau\mathsf{S} \bullet (t^\frown\langle\rangle) \approx_v^{\mathsf{S}} t|_u & [r = \langle\rangle] \\
\Leftrightarrow & \forall\, t \in \tau\mathsf{S} \bullet t \approx_v^{\mathsf{S}} t|_u & [t^\frown\langle\rangle = t] \\
\Leftrightarrow & u \not\rightsquigarrow v : \mathsf{S}. & \text{[Definition 3.9]} \quad \square
\end{array}$$

Non–interference is stronger than transaction non–interference.

**Lemma 4.21** *If system* $\mathsf{S}$ *has users* $u$ *and* $v$ *and* $T \subseteq u^*$ *then*

$$u \not\rightsquigarrow v : \mathsf{S} \Rightarrow T, u \not\rightsquigarrow v : \mathsf{S}.$$

**Proof** Note that because $T \subseteq u^*$ and $u \not\rightsquigarrow v : \mathsf{S}$ Lemma 4.1 shows that $\forall\, t \in \tau\mathsf{S}\, \forall\, r \in T \bullet (t^\frown r) \approx_v^{\mathsf{S}} t$.

$$\begin{array}{lll}
 & u \not\rightsquigarrow v : \mathsf{S} & \\
\Leftrightarrow & \forall\, t \in \tau\mathsf{S} \bullet t \approx_v^{\mathsf{S}} t|_u & \text{[Definition 3.9]} \\
\Rightarrow & \forall\, t \in \tau\mathsf{S}\, \forall\, r \in T \bullet (t^\frown r) \approx_v^{\mathsf{S}} t|_u & \text{[Lemma 4.1 and transitivity]} \\
\Rightarrow & \forall\, t \in \tau\mathsf{S}\, \exists\, r \in T \bullet (t^\frown r) \approx_v^{\mathsf{S}} t|_u & [\forall \Rightarrow \exists] \\
\Leftrightarrow & T, u \not\rightsquigarrow v : \mathsf{S}. & \text{[Definition 4.6]} \quad \square
\end{array}$$

## 4.3.3 Laws of Non–interference

This section deals with laws for transaction non–interference, of which non–interference is a special case (see Lemmas 4.20 and 4.21).

Laws are written in the form

$$\begin{array}{l}
(4.0.1)\ \mathit{antecedent}_1 \\
(4.0.2)\ \mathit{antecedent}_2 \\
\hline
\mathit{consequent}
\end{array}$$

which means that the conjunction of the antecedents implies the consequent; the antecedents are numbered for ease of reference.

Note that Lemma 4.20 shows when non–interference and transaction non–interference are equivalent. The antecedents of many of the laws given below are simplified greatly when non–interference is considered; the empty trace always completes any trace (as Lemma 4.20 shows).

Lemma 4.22 highlights some of the properties of transaction non–interference: parts 1 and 2 show that if either user's interface is empty then non–interference is trivial (note in part 2 the requirement that the purged traces exist: cf. Example 3.6); part 3 shows that the systems $\mathtt{Stop}_A$, $\mathtt{Run}_A$ and $\mathtt{Chaos}_A$ are non–interfering; parts 4 and 5 show that contracting the interface of a user does not introduce interference

and neither does expanding the completion set; parts 6 and 7 show that if the system is never willing to engage in events from $u$ or $v$ then the system is non–interfering.

**Lemma 4.22** *For system* $\mathsf{S}$ *with alphabet* $A$ *and users* $u, v, v_0 \subseteq A$ *and* $T, T_0 \subseteq u^*$*:*

1. $\{\langle\rangle\}, \emptyset \not\rightsquigarrow v : \mathsf{S}$;
2. $\forall\, t \in \tau\mathsf{S} \bullet t|_u \in \tau\mathsf{S} \Rightarrow T, u \not\rightsquigarrow \emptyset : \mathsf{S}$;
3. $T, u \not\rightsquigarrow v : \mathtt{Stop}_A; T, u \not\rightsquigarrow v : \mathtt{Run}_A; T, u \not\rightsquigarrow v : \mathtt{Chaos}_A$;
4. $(T, u \not\rightsquigarrow v : \mathsf{S}) \wedge (v_0 \subseteq v) \Rightarrow T, u \not\rightsquigarrow v_0 : \mathsf{S}$;
5. $(T, u \not\rightsquigarrow v : \mathsf{S}) \wedge (T \subseteq T_0) \Rightarrow T_0, u \not\rightsquigarrow v : \mathsf{S}$;
6. $\forall\, t \in \tau\mathsf{S} \bullet t|_u = t \Rightarrow T, u \not\rightsquigarrow v : \mathsf{S}$;
7. $\forall\, t \in \tau\mathsf{S} \bullet (\mathsf{S}/t \equiv_v^A \mathtt{Stop}_A) \wedge (t|_u \in \tau\mathsf{S}) \Rightarrow T, u \not\rightsquigarrow v : \mathsf{S}$.

**Proof**

1. Clearly $t|_\emptyset = t$ and so Definition 4.6 gives $\{\langle\rangle\}, \emptyset \not\rightsquigarrow v : \mathsf{S}$ iff $\forall\, t \in \tau\mathsf{S} \bullet t \approx_v^{\mathsf{S}} t$. That equivalence holds since $\approx_v^{\mathsf{S}}$ is reflexive (see Lemma 3.2).
2. This part follows from the fact that $\forall\, t \in \tau\mathsf{S} \bullet t|_u \in \tau\mathsf{S}$ and that $\tau\mathsf{S}^2 \subseteq \approx_\emptyset^{\mathsf{S}}$.
3. $T, u \not\rightsquigarrow v : \mathtt{Run}_A$ and $T, u \not\rightsquigarrow v : \mathtt{Chaos}_A$ follow from the fact that $\forall\, t \in A^*$: $\mathtt{Run}_A/t = \mathtt{Run}_A$ and $\mathtt{Chaos}_A/t = \mathtt{Chaos}_A$. Hence, all traces are $v$–equivalent.

   Since the only trace of $\mathtt{Stop}_A$ is $\langle\rangle$ and $\langle\rangle \in T$, Definition 4.6 gives $\langle\rangle \approx_v^{\mathrm{Stop}_A} \langle\rangle|_u$ as the condition that $T, u \not\rightsquigarrow v : \mathtt{Stop}_A$. That follows from the fact that $\approx_v^{\mathrm{Stop}_A}$ is reflexive (Lemma 3.2).
4. Follows from Lemma 3.3 and Definition 4.6.
5. Adding extra completions does not change the fact that at least one completion exists for each trace:

$$\begin{array}{lll} & (T, u \not\rightsquigarrow v : \mathsf{S}) & \\ \Leftrightarrow & \forall\, t \in \tau\mathsf{S}\, \exists\, r \in T \bullet (t ^\frown r) \approx_v^{\mathsf{S}} t|_u & [\text{Definition 4.6}] \\ \Rightarrow & \forall\, t \in \tau\mathsf{S}\, \exists\, r \in T_0 \bullet (t ^\frown r) \approx_v^{\mathsf{S}} t|_u & [T \subseteq T_0] \\ \Leftrightarrow & T_0, u \not\rightsquigarrow v : \mathsf{S}. & [\text{Definition 4.6}] \end{array}$$

6. For any $t \in \tau\mathsf{S}$ $t|_u = t$ and as $\approx_v^{\mathsf{S}}$ is reflexive (Lemma 3.2) we have $t \approx_v^{\mathsf{S}} t|_u$. So by Definition 3.9 $u \not\rightsquigarrow v : \mathsf{S}$ and by Lemma 4.21 $T, u \not\rightsquigarrow v : \mathsf{S}$.
7. Consider a trace $t$ of $\mathsf{S}$: we show that $t \approx_v^{\mathsf{S}} t|_u$. $t|_u \in \tau\mathsf{S}$ and $\mathsf{S}/t \equiv_v^A \mathtt{Stop}_A \equiv_v^A \mathsf{S}/t|_u$. Hence, by transitivity (Lemma 3.2) and Definition 3.7, $t \approx_v^{\mathsf{S}} t|_u$. So $u \not\rightsquigarrow v : \mathsf{S}$ (Definition 3.9) and by Lemma 4.21 $T, u \not\rightsquigarrow v : \mathsf{S}$. □

**Laws concerning Prefixing**

If a process exhibits non–interference from $u$ to $v$ then prefixing that process by events *not* in $u$ does not introduce interference.

**Law 4.1** (*choice–1*)

$$\frac{\begin{array}{ll}(4.1.1) & \forall\, b \in B \bullet T, u \not\rightsquigarrow v : \mathsf{S}_b \\ (4.1.2) & B \cap u = \emptyset\end{array}}{T, u \not\rightsquigarrow v : (b : B \rightarrow \mathsf{S}_b)}$$

**Proof** If $t$ is a non–empty trace of $b : B \rightarrow \mathsf{S}_b$ (the case $t = \langle\rangle$ is trivial) then $t = \langle b \rangle ^\frown s$ for some $b \in B$ and $s \in \tau\mathsf{S}_b$.

$$\begin{array}{lll} & \exists\, r \in T \bullet (b : B \rightarrow \mathsf{S}_b)/(t ^\frown r) & \\ = & \mathsf{S}_b/(s ^\frown r) & [\text{L2:53; L3:53}] \\ \equiv_v^A & (\mathsf{S}_b/s|_u) & [T, u \not\rightsquigarrow v : \mathsf{S}_b] \\ = & (b : B \rightarrow \mathsf{S}_b)/(\langle b \rangle ^\frown s|_u) & [\text{L2:53; L3:53}] \\ = & (b : B \rightarrow \mathsf{S}_b)/t|_u. & [B \cap u = \emptyset] \end{array}$$

Hence $T, u \not\rightsquigarrow v : (b : B \rightarrow \mathsf{S}_b)$ by Definitions 3.7 and 4.6. □

The previous rule excluded any elements of $u$ from the set $B$: Law 4.2 shows that when $B$ is a subset of $u$ non–interference holds when the system is given the choice of performing elements from $B$, or not.

Condition (4.2.4) states that if the $\mathsf{S}_b$ have traces in common then they must not be distinguishable by $v$.

**Law 4.2** (*choice–2*)

$$\frac{\begin{array}{ll}(4.2.1) & \forall\, b \in B \bullet T, u \not\rightsquigarrow v : \mathsf{S}_b \\ (4.2.2) & \mathsf{R} = (\mathop{\|}_{b:B} \mathsf{S}_b) \\ (4.2.3) & B \cap \iota\mathsf{R} = u \cap \iota\mathsf{R} = \emptyset \\ (4.2.4) & \forall\, b, c \in B\, \forall\, t \in \tau\mathsf{S}_b \cap \tau\mathsf{S}_c \bullet \mathsf{S}_b/t \equiv_v^A \mathsf{S}_c/t\end{array}}{T, u \not\rightsquigarrow v : (b : B \rightarrow \mathsf{S}_b) \| \mathsf{R}}$$

**Proof** $B \cap \iota\mathsf{R} = \emptyset$ guarantees that $(b : B \rightarrow \mathsf{S}_b)$ and $\mathsf{R}$ have no non–empty traces in common.

$u \cap \iota\mathsf{R} = \emptyset$ guarantees that if $t \in \tau\mathsf{S}_b$ and $t$ is non–empty then $t|_u$ is non–empty.

Put $\mathsf{T} = (b : B \rightarrow \mathsf{S}_b) \| \mathsf{R}$ and take $t \in \tau\mathsf{T}$. The case $t = \langle\rangle$ is trivial so assume that $t \neq \langle\rangle$.

(case $t \in \tau(b : B \rightarrow \mathsf{S}_b)$, $b \in B$ and $t = \langle b\rangle^\frown t'$)

$$
\begin{array}{lcll}
 & & \exists\, r \in T \bullet \mathsf{T}/(t^\frown r) & \\
 & = & (b : B \rightarrow \mathsf{S}_b)/(t^\frown r) & \text{[L2:108]} \\
 & = & \mathsf{S}_b/(t'^\frown r) & \text{[L2:53; L3:53]} \\
 & \equiv^A_v & \mathsf{S}_b/t'|_u : & [T, u \not\rightsquigarrow v : \mathsf{S}_b] \\
\text{(case } b \notin u) & = & (b : B \rightarrow \mathsf{S}_b)/(\langle b\rangle^\frown t')|_u & \text{[L2:53; L3:53]} \\
 & = & \mathsf{T}/t|_u ; & \text{[L2:108]} \\
\text{(case } b \in u) & \equiv^A_v & \mathsf{R}/t'|_u & \text{[(4.2.4)]} \\
 & = & \mathsf{T}/t|_u . & \text{[L2:108]}
\end{array}
$$

(case $t \in \tau\mathsf{R}$)

$$
\begin{array}{cll}
 & \exists\, r \in T \bullet \mathsf{T}/(t^\frown r) & \\
= & \mathsf{R}/(t^\frown r) & \text{[L2:108]} \\
\equiv^A_v & \mathsf{S}_b/(t^\frown r) & \text{[(4.2.4); some } b] \\
\equiv^A_v & \mathsf{S}_b/t|_u & [T, u \not\rightsquigarrow v : \mathsf{S}_b] \\
\equiv^A_v & \mathsf{R}/t|_u & \text{[(4.2.4)]} \\
= & \mathsf{T}/t|_u . & \text{[L2:108]} \quad \square
\end{array}
$$

After a trace, which does not contain events from $u$'s interface, a non–interfering system is still non–interfering.

**Law 4.3** (*after*)

$$
\begin{array}{l}
(4.3.1)\ T, u \not\rightsquigarrow v : \mathsf{S} \\
(4.3.2)\ t \in \tau\mathsf{S} \\
(4.3.3)\ t|_u = t \\
\hline
T, u \not\rightsquigarrow v : (\mathsf{S}/t)
\end{array}
$$

**Proof** Consider a trace $s$ of $\mathsf{S}/t$.

$$
\begin{array}{cll}
 & \exists\, r \in T \bullet \mathsf{S}/(t^\frown s^\frown r) & \\
\equiv^A_v & \mathsf{S}/(t^\frown s)|_u & [T, u \not\rightsquigarrow v : \mathsf{S}] \\
= & \mathsf{S}/(t|_u{}^\frown s|_u) & [\ |_u \text{ distributive}] \\
= & (\mathsf{S}/t|_u)/s|_u & \text{[L2:53]} \\
= & (\mathsf{S}/t)/s|_u . & [t|_u = t]
\end{array}
$$

Hence $\exists\, r \in T \bullet (\mathsf{S}/t)/(s^\frown r) \equiv_v^A (\mathsf{S}/t)/s|_u$ and so by Definition 4.6 $T, u \not\rightsquigarrow v : (\mathsf{S}/t)$. □

**Laws concerning Change of Alphabet**

Changing the alphabet of a non–interfering process does not introduce interference; changing the alphabet just changes the interfaces and set of completions.

**Law 4.4** (*change alphabet*)

$$\frac{\begin{array}{ll}(4.4.1) & T, u \not\rightsquigarrow v : \mathsf{S} \\ (4.4.2) & f \textit{ is injective}\end{array}}{f(T), f(u) \not\rightsquigarrow f(v) : f(\mathsf{S})}$$

**Proof** Put $B = \alpha f(\mathsf{S})$, $u' = f(u)$ and $v' = f(v)$ (note that $u'$ and $v'$ are disjoint since $f$ is injective).

If $s \in \tau f(\mathsf{S})$ then there exists $t \in \tau\mathsf{S}$ such that $f^*(t) = s$, $f^*(t|_u) = s|_{u'}$, and

$$\begin{array}{lll} & \exists\, r \in T \bullet \mathsf{S}/(t^\frown r) \equiv_v^A \mathsf{S}/t|_u & [T, u \not\rightsquigarrow v : \mathsf{S} \text{ and Definition 4.6}] \\ \Rightarrow & \exists\, r \in T \bullet f(\mathsf{S}/(t^\frown r)) \equiv_{v'}^B f(\mathsf{S}/t|_u) & [\text{Lemma 4.14}] \\ \Rightarrow & \exists\, r \in T \bullet f(\mathsf{S})/f^*(t^\frown r) \equiv_{v'}^B f(\mathsf{S})/f^*(t|_u) & [\text{L7:85}] \\ \Rightarrow & \exists\, r' \in f(T) \bullet f(\mathsf{S})/(s^\frown r') \equiv_{v'}^B f(\mathsf{S})/(s|_{u'}). & \end{array}$$

Hence $f(T), f(u) \not\rightsquigarrow f(v) : f(\mathsf{S})$ by Definition 4.6. □

Hiding events from the alphabet of a system does not cause interference. The set $H(t)$ is the set of traces of $\mathsf{S}$ that are equal to $t$ when events from the set $C_v$ are hidden: $H(t) \triangleq \{s \mid s \in \tau\mathsf{S} \wedge s|_{C_v} = t\}$.

Condition (4.5.2) means that for all traces $s$, that when $C_v$ is hidden become $t$, there is a single completion $r$ which is completes them all. Thus hiding $C_v$ will not enable $v$ to distinguish traces.

**Law 4.5** (*hiding*)

$$\frac{\begin{array}{ll}(4.5.1) & T, u \not\rightsquigarrow v : \mathsf{S} \\ (4.5.2) & \forall\, t \in \tau\mathsf{S}\, \exists\, r \in T\, \forall\, s \in H(t) \bullet \mathbf{com}_{\mathsf{S}}(s, r) \\ (4.5.3) & C \subseteq \alpha\mathsf{S}\end{array}}{(T \setminus C), (u \setminus C) \not\rightsquigarrow (v \setminus C) : (\mathsf{S} \setminus C)}$$

**Proof** Suppose $C = C' \cup C_v$ where $C' \cap v = \emptyset$ and $C_v \subseteq v$.

$$T, u \not\rightsquigarrow (v \setminus C_v) : (\mathsf{S} \setminus C_v)$$

follows by considering any $t \in \tau(\mathsf{S} \setminus C_v)$.

$$\begin{array}{ll}
 & \forall\, s \in H(t)\, \exists\, r \in T \bullet \mathsf{S}/(s^\frown r) \equiv_v^A \mathsf{S}/s|_u \\
 & \hfill [T, u \not\rightsquigarrow v : \mathsf{S}] \\
\Rightarrow & \forall\, s \in H(t)\, \exists\, r \in T \bullet (\mathsf{S}/(s^\frown r)) \setminus C_v \equiv_{v \setminus C_v}^{A \setminus C_v} (\mathsf{S}/s|_u) \setminus C_v \\
 & \hfill [\text{Lemma 4.15}] \\
\Rightarrow & \exists\, r \in T\, \forall\, s \in H(t) \bullet (\mathsf{S}/(s^\frown r)) \setminus C_v \equiv_{v \setminus C_v}^{A \setminus C_v} (\mathsf{S}/s|_u) \setminus C_v \\
 & \hfill [(4.5.2)] \\
\Rightarrow & \exists\, r \in T \bullet \sqcap_{s \in H(t)} (\mathsf{S}/(s^\frown r)) \setminus C_v \equiv_{v \setminus C_v}^{A \setminus C_v} \sqcap_{s' \in H(t|_u)} (\mathsf{S}/s') \setminus C_v \\
 & \hfill [\text{Lemma 4.13}] \\
\Rightarrow & \exists\, r \in (T \setminus C_v) \bullet (\mathsf{S} \setminus C_v)/(t^\frown r) \equiv_{v \setminus C_v}^{A \setminus C_v} (\mathsf{S} \setminus C_v)/t|_u . \\
 & \hfill [\text{L2:116}]
\end{array}$$

Then hide $C'$ to obtain the result; we verify

$$(T \setminus C'), (u \setminus C') \not\rightsquigarrow v : (\mathsf{S} \setminus C')$$

by arguing in the failures–divergences model (cf. proof of Lemma 4.19). Take $t \in \tau(\mathsf{S} \setminus C')$ and $w \in v^*$ then there is $r \in T \setminus C'$ such that

$$\begin{array}{lll}
 & (t^\frown r^\frown w, X) \in \phi(\mathsf{S} \setminus C') & \\
\Leftrightarrow & \exists\, s' \in \tau\mathsf{S}\, \exists\, r' \in T \bullet\, (s'^\frown r'^\frown w, X \cup C') \in \phi\mathsf{S} & [\text{D14:131}] \\
 & \qquad \wedge & \\
 & \qquad (s'|_{C'} = t) & [w \in v^*] \\
 & \qquad \wedge & \\
 & \qquad (r'|_{C'} = r) & [r \in (T \setminus C')] \\
\Leftrightarrow & (s'|_u{}^\frown w, Y \cup C') \in \phi\mathsf{S} & [T, u \not\rightsquigarrow v : \mathsf{S}] \\
\Leftrightarrow & ((s'|_u{}^\frown w)|_{C'}, Y) \in \phi(\mathsf{S} \setminus C') & [\text{D14:131}] \\
\Leftrightarrow & ((s'|_{C'}|_u{}^\frown w), Y) \in \phi(\mathsf{S} \setminus C') & [C' \cap v = \emptyset] \\
\Leftrightarrow & (t|_u{}^\frown w, Y) \in \phi(\mathsf{S} \setminus C'). & [w \in v^*\,;\, s'|_{C'} = t]
\end{array}$$

Similarly for the divergences. □

Note that (4.5.2) is not required if $C \cap v = \emptyset$ and is trivial when non–interference is considered instead of transaction non–interference (just put $r = \langle\rangle$).

### Laws concerning Composition of Systems

The simplest law of parallel composition is that if two systems are non–interfering then their concurrent combination is also non–interfering.

Condition (4.6.3) means that for any trace $t$ of $(\mathsf{S} \| \mathsf{T})$ there is an $r$ that completes both $t \upharpoonright \alpha\mathsf{S}$ and $t \upharpoonright \alpha\mathsf{T}$. That condition means systems $\mathsf{S}$ and $\mathsf{T}$ must agree upon the valid completions of any trace of $(\mathsf{S} \| \mathsf{T})$.

**Law 4.6** (*parallel*)

$$\begin{array}{ll}
(4.6.1) & T, u \not\rightsquigarrow v : \mathsf{S} \\
(4.6.2) & T, u \not\rightsquigarrow v : \mathsf{T} \\
(4.6.3) & \forall\, t \in \tau(\mathsf{S} \| \mathsf{T})\, \exists\, r \in T \bullet \mathbf{com}_{\mathsf{S}}(t \upharpoonright \alpha\mathsf{S}, r) \\
& \qquad\qquad\qquad\qquad\quad \wedge \\
& \qquad\qquad\qquad\qquad\quad \mathbf{com}_{\mathsf{T}}(t \upharpoonright \alpha\mathsf{T}, r) \\
\hline
& T, u \not\rightsquigarrow v : (\mathsf{S} \| \mathsf{T})
\end{array}$$

**Proof** If $t \in \tau(\mathsf{S}\|\mathsf{T})$ then by (4.6.3), $\alpha\mathsf{S} = A$ and $\alpha\mathsf{T} = B$ then there exists $r \in T$ such that $\mathbf{com}_{\mathsf{S}}(t \upharpoonright A, r)$ and $\mathbf{com}_{\mathsf{T}}(t \upharpoonright B, r)$ and

$$\begin{array}{cll}
& ((\mathsf{S}\|\mathsf{T})/(t^\frown r))\|\overline{v}_{A\cup B} & \\
= & (\mathsf{S}/(t^\frown r)\upharpoonright A\|\mathsf{T}/(t^\frown r)\upharpoonright B)\|\overline{v}_{A\cup B} & [\text{L2:72}] \\
= & (\mathsf{S}/(t^\frown r)\upharpoonright A\|\overline{v}_{A\cup B})\|(\mathsf{T}/(t^\frown r)\upharpoonright B\|\overline{v}_{A\cup B}) & [\text{L2:70}] \\
= & (\mathsf{S}/(t^\frown r)\upharpoonright A\|\overline{v}_{A}\|\overline{v}_{A\cup B})\|(\mathsf{T}/(t^\frown r)\upharpoonright B\|\overline{v}_{B}\|\overline{v}_{A\cup B}) & \\
& & [\text{Lemma 4.17}] \\
= & (\mathsf{S}/(t\upharpoonright A^\frown r)\|\overline{v}_{A}\|\overline{v}_{A\cup B})\|(\mathsf{T}/(t\upharpoonright B^\frown r)\|\overline{v}_{B}\|\overline{v}_{A\cup B}) & \\
& & [u \subseteq A \cap B] \\
= & (\mathsf{S}/(t\upharpoonright A)|_u\|\overline{v}_{A}\|\overline{v}_{A\cup B})\|(\mathsf{T}/(t\upharpoonright B)|_u\|\overline{v}_{B}\|\overline{v}_{A\cup B}) & \\
& & [(4.6.1);\ (4.6.2);\ (4.6.3)] \\
= & (\mathsf{S}/(t\upharpoonright A)|_u\|\overline{v}_{A\cup B})\|(\mathsf{T}/(t\upharpoonright B)|_u\|\overline{v}_{A\cup B}) & \\
& & [\text{Lemma 4.17}] \\
= & (\mathsf{S}/(t\upharpoonright A)|_u\|\mathsf{T}/(t\upharpoonright B)|_u)\|\overline{v}_{A\cup B} & [\text{L2:70}] \\
= & ((\mathsf{S}\|\mathsf{T})/t|_u)\|\overline{v}_{A\cup B}. & [\text{L2:72}] \quad \square
\end{array}$$

Law 4.7 enables two non–interfering processes to be interleaved: their users and completion sets are also interleaved.

The interleaving of sets of traces is given by Definition 4.8.

**Definition 4.8** *If $T_0$ and $T_1$ are sets of traces drawn from the sets $w$ and $y$ then the* interleaving *of their traces is*

$$T_0 \mid T_1 \triangleq \{t \mid t\upharpoonright w \in T_0 \wedge t\upharpoonright y \in T_1\}. \qquad \triangle$$

**Law 4.7** (*amalgamate users*)

$$\begin{array}{ll}
(4.7.1) & T_0, w \not\rightsquigarrow x : \mathsf{S} \\
(4.7.2) & T_1, y \not\rightsquigarrow z : \mathsf{T} \\
(4.7.3) & \alpha\mathsf{S} \cap \alpha\mathsf{T} = \emptyset \\
\hline
& (T_0 \mid T_1), (w \cup y) \not\rightsquigarrow (x \cup z) : (\mathsf{S}\|\mathsf{T})
\end{array}$$

**Proof** Put $u = w \cup y$, $v = x \cup z$, $\alpha\mathsf{S} = A$ and $\alpha\mathsf{T} = B$. If $t \in \tau(\mathsf{S}\|\mathsf{T})$ then there exists $r \in (T_0 \mid T_1)$ such that

$$
\begin{array}{lll}
& ((\mathsf{S}\|\mathsf{T})/(t^\frown r))\|\overline{v}_{A\cup B} & \\
= & (\mathsf{S}/(t^\frown r)\upharpoonright A\|\mathsf{T}/(t^\frown r)\upharpoonright B)\|\overline{v}_{A\cup B} & \text{[L2:72]} \\
= & (\mathsf{S}/(t^\frown r)\upharpoonright A\|\overline{v}_{A\cup B})\|(\mathsf{T}/(t^\frown r)\upharpoonright B\|\overline{v}_{A\cup B}) & \text{[L2:70]} \\
= & (\mathsf{S}/(t^\frown r)\upharpoonright A\|\overline{x}_A\|\overline{v}_{A\cup B})\|(\mathsf{T}/(t^\frown r)\upharpoonright B\|\overline{z}_B\|\overline{v}_{A\cup B}) & \\
& & \text{[Lemma 4.18]} \\
= & (\mathsf{S}/(t\upharpoonright A^\frown r\upharpoonright A)\|\overline{x}_A\|\overline{v}_{A\cup B})\|(\mathsf{T}/(t\upharpoonright B^\frown r\upharpoonright B)\|\overline{z}_B\|\overline{v}_{A\cup B}) & \\
= & (\mathsf{S}/(t\upharpoonright A)|_w\|\overline{x}_A\|\overline{v}_{A\cup B})\|(\mathsf{T}/(t\upharpoonright B)|_y\|\overline{z}_B\|\overline{v}_{A\cup B}) & \\
& & \text{[(4.7.1); (4.7.2)]} \\
= & (\mathsf{S}/(t\upharpoonright A)|_w\|\overline{v}_{A\cup B})\|(\mathsf{T}/(t\upharpoonright B)|_y\|\overline{v}_{A\cup B}) & \text{[Lemma 4.18]} \\
= & (\mathsf{S}/(t\upharpoonright A)|_w\|\mathsf{T}/(t\upharpoonright B)|_y)\|\overline{v}_{A\cup B} & \text{[L2:70]} \\
= & ((\mathsf{S}\|\mathsf{T})/t|_u)\|\overline{v}_{A\cup B}. & \text{[L2:72]} \quad \square
\end{array}
$$

Given a non–interfering system $\mathsf{S}$ ($T, u \not\rightsquigarrow v : \mathsf{S}$) it is possible to constrain the activity of users $u$ and $v$ without introducing interference. Law 4.8 shows that not only may users be constrained, but their interfaces may be expanded. System $\mathsf{U}$ synchronizes with $\mathsf{S}$ on a subset of $u$, and system $\mathsf{V}$ synchronizes with $\mathsf{S}$ on any subset of $\alpha\mathsf{S}$ not containing events from $u$.

**Law 4.8** (*constrain users*)

$$
\begin{array}{ll}
(4.8.1) & T, u \not\rightsquigarrow v : \mathsf{S} \\
(4.8.2) & u' = \alpha\mathsf{U} \\
(4.8.3) & (\alpha\mathsf{S} \cap u') \subseteq u \\
(4.8.4) & \forall\, t \in \tau\mathsf{S}\, \exists\, r \in T \bullet \mathbf{com}_{\mathsf{S}}(t, r) \Rightarrow (t^\frown r)\upharpoonright u' \in \tau\mathsf{U} \\
(4.8.5) & \forall\, t \in \tau\mathsf{U} \bullet t|_u \in \tau\mathsf{U} \\
(4.8.6) & v' = \alpha\mathsf{V} \\
(4.8.7) & (\alpha\mathsf{S} \cap v') \subseteq \alpha\mathsf{S} \setminus u \\
\hline
& T, u' \not\rightsquigarrow v' : (\mathsf{S}\|\mathsf{U}\|\mathsf{V})
\end{array}
$$

**Proof** Take $t \in \tau(\mathsf{S}\|\mathsf{U}\|\mathsf{V})$. Then $t\upharpoonright\alpha\mathsf{S} \in \tau\mathsf{S}$ and there exists $r \in T$ such that $\mathbf{com}_{\mathsf{S}}(t\upharpoonright\alpha\mathsf{S}, r)$, since $T, u \not\rightsquigarrow v : \mathsf{S}$.

$$
\begin{array}{lll}
& (\mathsf{S}\|\mathsf{U}\|\mathsf{V})/(t^\frown r) & \\
= & (\mathsf{S}/(t^\frown r)\upharpoonright\alpha\mathsf{S})\|(\mathsf{U}/(t^\frown r)\upharpoonright u')\|(\mathsf{V}/(t^\frown r)\upharpoonright v') & \\
& & \text{[L2:72 and (4.8.4)]} \\
= & (\mathsf{S}/(t\upharpoonright\alpha\mathsf{S}^\frown r))\|(\mathsf{U}/(t^\frown r)\upharpoonright u')\|(\mathsf{V}/t\upharpoonright v') & [r \in u^*] \\
\equiv_v^A & (\mathsf{S}/t|_u\upharpoonright\alpha\mathsf{S})\|(\mathsf{U}/t|_u\upharpoonright u')\|(\mathsf{V}/t|_u\upharpoonright v') & \\
& & [\mathbf{com}_{\mathsf{S}}(t\upharpoonright\alpha\mathsf{S}, r), \text{(4.8.1) and (4.8.5)}] \\
= & (\mathsf{S}\|\mathsf{U}\|\mathsf{V})/t|_u. & \text{[L2:72]} \quad \square
\end{array}
$$

Either of the processes $\mathsf{U}$ or $\mathsf{V}$ may be dropped from Law 4.8. If $\mathsf{U}$ is dropped then conditions (4.8.3) to (4.8.5) are no longer required. If $\mathsf{V}$ is dropped then condition (4.8.7) need not be checked. In that way a process to constrain the activity of one of user $u$ and user $v$ can be constructed. Law 4.8 simply expressed both possibilities in a single law.

Law 4.8 showed how to compose a non–interfering system with one that synchronizes with one of the users. Law 4.9 shows how to compose a non–interfering system with one that may synchronize with both the users.

**Law 4.9** (*constrain system*)

(4.9.1) $T, u \not\rightsquigarrow v : \mathsf{S}$

(4.9.2) $T', (u \cap \alpha\mathsf{T}) \not\rightsquigarrow (\alpha\mathsf{T} \setminus u) : \mathsf{T}$
$\vee$
$T', (\alpha\mathsf{T} \setminus v) \not\rightsquigarrow (v \cap \alpha\mathsf{T}) : \mathsf{T}$

(4.9.3) $\forall\, t \in \tau(\mathsf{S} \| \mathsf{T})\, \exists\, r \in T \bullet \mathbf{com}_{\mathsf{S}}(t, r) \wedge \mathbf{com}_{\mathsf{T}}(t \restriction \alpha\mathsf{T}, r \restriction \alpha\mathsf{T})$

---

$T, u \not\rightsquigarrow v : (\mathsf{S} \| \mathsf{T})$

**Proof** The important step in this proof is the replacement of $\overline{v}_A$ with $\overline{v}_A \| \overline{v}_{\alpha\mathsf{T}}$ or $\overline{v}_A \| \overline{v}_u$ using Lemma 4.17, since $A \cup \alpha\mathsf{T} = A$ and $A \cup u = A$. The replacement is used on the fourth line of the following deduction to replace $(\mathsf{T}/(t^\frown r) \restriction \alpha\mathsf{T}) \| \overline{v}_A$ with $(\mathsf{T}/t|_u) \| \overline{v}_A$. That substitution is justified by conditions (4.9.2) and (4.9.3).

Take any $t \in \tau(\mathsf{S} \| \mathsf{T})$ then there exists $r \in T$ such that

$$
\begin{array}{lll}
 & (\mathsf{S} \| \mathsf{T})/(t^\frown r) \| \overline{v}_A & \\
= & (\mathsf{S}/(t^\frown r)) \| (\mathsf{T}/(t^\frown r) \restriction \alpha\mathsf{T}) \| \overline{v}_A & \text{[L2:72]} \\
= & (\mathsf{S}/(t^\frown r)) \| \overline{v}_A \| (\mathsf{T}/(t^\frown r) \restriction \alpha\mathsf{T}) \| \overline{v}_A & \text{[L2:70]} \\
= & (\mathsf{S}/t|_u) \| \overline{v}_A \| (\mathsf{T}/t|_u) \| \overline{v}_A & \text{[see above]} \\
= & (\mathsf{S}/t|_u) \| (\mathsf{T}/t|_u) \| \overline{v}_A & \text{[L2:70]} \\
= & (\mathsf{S} \| \mathsf{T})/t|_u \| \overline{v}_A & \text{[L2:72]}
\end{array}
$$

Hence $T, u \not\rightsquigarrow v : (\mathsf{S} \| \mathsf{T})$. □

The law for non–deterministic choice is simple: if both components are non–interfering then the composition is non–interfering if the common traces are indistinguishable. That means that user $v$ will be unable to gain information from knowing in which direction the non–determinism was resolved.

Conditions (4.10.4) and (4.10.5) guarantee that $\mathsf{S}$ and $\mathsf{T}$ agree on $v$'s view of common traces and that there is a trace $r$ that completes any common trace in both $\mathsf{S}$ and $\mathsf{T}$.

**Law 4.10** (*internal choice*)

$$
\begin{array}{ll}
(4.10.1) & T, u \not\rightsquigarrow v : \mathsf{S} \\
(4.10.2) & T, u \not\rightsquigarrow v : \mathsf{T} \\
(4.10.3) & \alpha\mathsf{S} = \alpha\mathsf{T} = A \\
(4.10.4) & \forall\, t \in \tau\mathsf{S} \cap \tau\mathsf{T} \bullet \mathsf{S}/t \equiv_v^A \mathsf{T}/t \\
(4.10.5) & \forall\, t \in \tau\mathsf{S} \cap \tau\mathsf{T}\, \exists\, r \in T \bullet \mathbf{com}_{\mathsf{S}}(t, r) \wedge \mathbf{com}_{\mathsf{T}}(t, r) \\
\hline
 & T, u \not\rightsquigarrow v : (\mathsf{S} \sqcap \mathsf{T})
\end{array}
$$

**Proof** If $t \in \tau\mathsf{S} \setminus \tau\mathsf{T}$ then:

$$
\begin{array}{lll}
 & (\mathsf{S} \sqcap \mathsf{T})/(t ^\frown r) & \\
\equiv_v^A & \mathsf{S}/(t ^\frown r) & [\text{L2:106}] \\
\equiv_v^A & \mathsf{S}/t|_u & [T, u \not\rightsquigarrow v : \mathsf{S}] \\
\equiv_v^A & (\mathsf{S} \sqcap \mathsf{T})/t|_u. & [(4.10.4)]
\end{array}
$$

Similarly for $t \in \tau\mathsf{T} \setminus \tau\mathsf{S}$. If $t \in \tau\mathsf{S} \cap \tau\mathsf{T}$ then:

$$
\begin{array}{lll}
 & (\mathsf{S} \sqcap \mathsf{T})/(t ^\frown r) & \\
\equiv_v^A & (\mathsf{S}/(t ^\frown r)) \sqcap (\mathsf{T}/(t ^\frown r)) & [\text{L2:106}] \\
\equiv_v^A & (\mathsf{S}/t|_u) \sqcap (\mathsf{T}/t|_u) & [(4.10.1),\ (4.10.2) \text{ and } (4.10.5)] \\
= & (\mathsf{S} \sqcap \mathsf{T})/t|_u. & [\text{L2:106}]
\end{array}
$$

When $t \notin \tau\mathsf{S} \cup \tau\mathsf{T}$ $(\mathsf{S} \sqcap \mathsf{T})/(t ^\frown r)$ and $(\mathsf{S} \sqcap \mathsf{T})/t|_u$ are undefined. □

Note that if the systems $\mathsf{S}$ and $\mathsf{T}$ have disjoint initials then they have no non–empty traces in common and conditions (4.10.4) and (4.10.5) are easily verified.

The law for external choice is similar, except that the empty trace must not be equivalent to any non–empty traces in the choice composition. That restriction is formalized by saying that the initials of $\mathsf{S}$ and $\mathsf{T}$ must not contain elements from $u$.

**Law 4.11** (*external choice*)

$$
\begin{array}{ll}
(4.11.1) & T, u \not\rightsquigarrow v : \mathsf{S} \\
(4.11.2) & T, u \not\rightsquigarrow v : \mathsf{T} \\
(4.11.3) & \alpha\mathsf{S} = \alpha\mathsf{T} = A \\
(4.11.4) & \iota\mathsf{S} \cap u = \iota\mathsf{T} \cap u = \emptyset \\
(4.11.5) & \forall\, t \in \tau\mathsf{S} \cap \tau\mathsf{T} \bullet \mathsf{S}/t \equiv_v^A \mathsf{T}/t \\
(4.11.6) & \forall\, t \in \tau\mathsf{S} \cap \tau\mathsf{T}\, \exists\, r \in T \bullet \mathbf{com}_{\mathsf{S}}(t, r) \wedge \mathbf{com}_{\mathsf{T}}(t, ) \\
\hline
 & T, u \not\rightsquigarrow v : (\mathsf{S} [\!] \mathsf{T})
\end{array}
$$

**Proof** As for Law 4.10 with [L2:106] replaced by [L2:108] and the case $t \in \tau\mathsf{S} \cap \tau\mathsf{T}$ changed to $(t \in \tau\mathsf{S} \cap \tau\mathsf{T}) \wedge (t \neq \langle\rangle)$. □

**Laws concerning Recursive Systems**

Recursion is handled by Law 4.12. If $\mathsf{X}$ non–interfering implies $\mathsf{F}(\mathsf{X})$ non–interfering then $\mu\,\mathsf{X} \bullet \mathsf{F}(\mathsf{X})$ is non–interfering.

**Law 4.12** (*recursion*)

$$\frac{(4.12.1)\ T, u \not\rightsquigarrow v : \mathsf{X} \Rightarrow T, u \not\rightsquigarrow v : \mathsf{F}(\mathsf{X})}{T, u \not\rightsquigarrow v : \mu\,\mathsf{X} \bullet \mathsf{F}(\mathsf{X})}$$

**Proof** Assume the alphabet of $\mu\,\mathsf{X} \bullet \mathsf{F}(\mathsf{X})$ is $A$. The antecedent and Lemma 4.22 part 3 guarantee that $T, u \not\rightsquigarrow v : \mathsf{F}^n(\mathrm{Chaos}_A)$ for all $n \geq 0$. If $\mathsf{Z} = \bigsqcup_{n \geq 0} \mathsf{F}^n(\mathrm{Chaos}_A)$ then

$$\begin{array}{lll}
 & \forall\, n \in \mathrm{N} \bullet T, u \not\rightsquigarrow v : \mathsf{F}^n(\mathrm{Chaos}_A) & \\
\Rightarrow & \forall\, n \in \mathrm{N}\, \forall\, t \in \tau\mathsf{F}^n(\mathrm{Chaos}_A)\, \exists\, r \in T \bullet (t^\frown r) \approx_v^{\mathsf{F}^n(\mathrm{Chaos}_A)} t|_u & \\
 & & [\text{Definition 4.6}] \\
\Rightarrow & \forall\, t \in \bigcap_{n \geq 0} \tau\mathsf{F}^n(\mathrm{Chaos}_A)\, \exists\, r \in T \bullet (t^\frown r) \bigcap_{n \geq 0} \approx_v^{\mathsf{F}^n(\mathrm{Chaos}_A)} t|_u & \\
\Rightarrow & \forall\, t \in \tau\mathsf{Z}\, \exists\, r \in T \bullet (t^\frown r) \approx_v^{\mathsf{Z}} t|_u & [\text{Lemma 4.19}] \\
\Rightarrow & \forall\, t \in \tau\mu\,\mathsf{X} \bullet \mathsf{F}(\mathsf{X})\, \exists\, r \in T \bullet (t^\frown r) \approx_v^{\mu\,\mathsf{X} \bullet \mathsf{F}(\mathsf{X})} t|_u. & [\text{D17:132}]
\end{array}$$

□

Law 4.12 can be extended to a mutually recursive process. If the vector $\underline{\mathsf{X}}$ has length 1 this law reduces to simple recursion ($\mu\,\mathsf{X} \bullet \mathsf{F}(\mathsf{X})$). (See Definition 2.12.)

**Law 4.13** (*mutual recursion–1*)

$$\frac{(4.13.1)\ \forall\, q \bullet (\forall\, p \bullet T, u \not\rightsquigarrow v : \mathsf{X}_p) \Rightarrow T, u \not\rightsquigarrow v : \mathsf{F}(\underline{\mathsf{X}})_q}{\forall\, q \bullet T, u \not\rightsquigarrow v : \mu\,\underline{\mathsf{X}} \bullet \underline{\mathsf{F}}(\underline{\mathsf{X}})_q}$$

**Proof** Simple extension of the result of Law 4.12. □

A special case of mutual recursion is given by Law 4.14. The definition of $\underline{\mathsf{G}}$ in Law 4.14 means that the recursion can (on each loop) either act as the non–interfering process $\underline{\mathsf{F}}$ or perform a complete transaction from $T$.

The function **do** is used to perform the transaction and is given by Definition 4.9.

**Definition 4.9** $\mathbf{do}\ \langle\rangle\ \mathsf{S} \triangleq \mathsf{S}$;
$\mathbf{do}\ (\langle e\rangle ^\frown t)\ \mathsf{S} \triangleq e \rightarrow (\mathbf{do}\ t\ \mathsf{S})$. △

**Law 4.14** (*mutual recursion–2*)

$$\frac{\begin{array}{ll}(4.14.1) & \forall q \bullet (\forall p \bullet T, u \not\rightsquigarrow v : \mathsf{X}_p) \Rightarrow T, u \not\rightsquigarrow v : \mathsf{F}(\underline{\mathsf{X}})_q \\ (4.14.2) & \forall q\, \exists e \in u\, \exists r \in T \bullet \mathsf{G}(\underline{\mathsf{X}})_q = \mathsf{F}(\underline{\mathsf{X}})_q \| \mathbf{do}\ (\langle e\rangle ^\frown r)\ \mathsf{X}_q\end{array}}{\forall q \bullet T, u \not\rightsquigarrow v : \mu\, \underline{\mathsf{X}} \bullet \underline{\mathsf{G}}(\underline{\mathsf{X}})_q}$$

**Proof** We prove that $T, u \not\rightsquigarrow v : \mathsf{G}(\underline{\mathsf{X}})_p$, for each $p$, and use Law 4.13 to derive $T, u \not\rightsquigarrow v : \mu\, \underline{\mathsf{X}} \bullet \underline{\mathsf{G}}(\underline{\mathsf{X}})_q$. Definition 4.6 shows that we must prove

$$\forall t \in \tau \mathsf{G}(\underline{\mathsf{X}})_p\, \exists r \in T \bullet (t ^\frown r) \approx_v^{\mathsf{G}(\underline{\mathsf{X}})_p} t|_u$$

to prove $T, u \not\rightsquigarrow v : \mathsf{G}(\underline{\mathsf{X}})_p$. Hence consider $t \in \tau \mathsf{G}(\underline{\mathsf{X}})_p$:

(case $t \in \tau \mathsf{F}(\underline{\mathsf{X}})_p$)

$$\begin{array}{rcll} \exists r \in T \bullet \mathsf{G}(\underline{\mathsf{X}})_p/(t ^\frown r) & = & \mathsf{F}(\underline{\mathsf{X}})_p/(t ^\frown r) & [\text{this case}] \\ & \equiv_v^A & \mathsf{F}(\underline{\mathsf{X}})_p/t|_u & [T, u \not\rightsquigarrow v : \mathsf{F}(\underline{\mathsf{X}})_p] \\ & \equiv_v^A & \mathsf{G}(\underline{\mathsf{X}})_p/t|_u; & [\mathbf{head}\ t|_u \notin u] \end{array}$$

(case $t \in \tau \mathbf{do}\ (\langle e\rangle ^\frown r')\ \mathsf{X}_p)$)

$$\begin{array}{rcll} \exists r \in T \bullet \mathsf{G}(\underline{\mathsf{X}})_p/(t ^\frown r) & = & (\mathbf{do}\ (\langle e\rangle ^\frown r')\ \mathsf{X}_p)/(t ^\frown r) & [\text{this case}] \\ & \equiv_v^A & \mathsf{X}_p/t|_u & [T, u \not\rightsquigarrow v : \mathsf{X}_p,\ r \in T] \\ & = & \mathsf{G}(\underline{\mathsf{X}})_p/t|_u. & [\mathsf{X}_p = \mathsf{G}(\underline{\mathsf{X}})_p] \end{array}$$

□

# Chapter 5

# Case Study

## 5.1 Introduction

This case study illustrates the laws for secure–system development presented in Section 4.3. The study is based on a multi–user system satisfying the MLS security policy. Users are assigned one of a number of levels and may only communicate with users at or below that level. The main design decision is to distribute the system between the users, thus avoiding a kernelized implementation. The reason for avoiding a security kernel is that our approach asserts that the total system must be considered when defining its security properties and putting all assurance in one part of a system does not easily fit with that approach. The file–store and security functions are distributed and implemented locally.

The development starts with the abstract functional specification and security policy. The functional properties are expressed in the traces model of CSP and the security property used is non–interference (see Section 3.3). We find that, as [28, 29] intended, the MLS property is easily expressed in terms of non–interference.

We proceed by giving the functional and security development side–by–side and work to a CSP process which uses the access restrictions 'no read–up' and 'no write–down' to help to achieve the MLS property desired. In doing so we discover a novel way of preventing communication by denial of service, through the design of the interface.

Finally the system is translated to occam (see [41]) to produce a working prototype. That implementation is examined to discover what further security problems exist and solutions to them are discussed in Section 5.5.

## 5.2 Specification

The system consists of a file–store whose files each belong to a named user and contain a natural number. The use of files containing a single number abstracts from sequential and random access files simplifying the exposition. That abstraction does not, however, detract from the specification or assurance of security. The security policy is defined by reference to the users. The system accepts commands (engages in events for reading and writing files) from the users and issues replies (other events which return data or suitable messages).

The system is intended to be a distributed database of named files belonging to named users. The values returned by read events will reflect values previously written by write events. The order in which the files are updated (to reflect write events) is not specified and will become evident in the development (see Section 5.3).

The system has $n_u$ (finite) users and the set of users is $USR$. Note that Definition 5.4 introduces the structure of the $u_i$.

**Definition 5.1** $USR \triangleq \{u_i \mid 0 \leq i < n_u\}$. △

Each user is identified with a set of events. In order to list those events we define four further sets:

- ◇ file names (there are $n_f$ such file names);
- ◇ messages (which are described below);
- ◇ data items (the values that may be stored in a file are natural numbers between 0 and some fixed value $n_d$—$n_d$ is chosen to be finite so that the file–store may be implemented) and
- ◇ commands (described below).

We do not identify a user with a particular behaviour; the user is allowed to attempt to engage in any of his events at any time. User $u_i$'s behaviour is (at worst) `Run` on $u_i$.

**Definition 5.2** $FSP \triangleq \{i \mid 0 \leq i < n_f\}$;
$MSG \triangleq \{\mathbf{ok}, \mathbf{er}\}$;
$DAT \triangleq \{i \mid 0 \leq i < n_d\}$;
$CMD \triangleq \{\mathbf{rd}, \mathbf{wr}\}$. △

The system has three kinds of event. The first two are 'commands' (from the set $CMD$) engaged by a user along the input channel ($i.in$ for user $u_i$) for that user.

$(\mathbf{rd}, f, x, j)$ A request to read the contents of file $f \in FSP$ belonging to user $u_j \in USR$ ($x$ is an arbitrary value).

$(\mathbf{wr}, f, d, j)$ A request to write the data $d \in DAT$ to file $f \in FSP$ belonging to user $u_j \in USR$.

The third kind of event occurs, in reply to one of the events above, along a second channel ($i.out$) and is one of these two messages:

**ok** The reply to a successful write;

**er** The last request was unsuccessful. This may have been caused by reading a non–existent file or by the system denying access to a file because the request breached the security policy.

Additionally, an item of data from $DAT$ may be communicated on $i.out$ in reply to a **rd** command.

The set $OP$ contains all possible triples of commands, file names and data, and is used to give later definitions.

**Definition 5.3** $OP \triangleq CMD \times FSP \times DAT$. △

So, the events user $u_i$ may perform are given by

**Definition 5.4**

$$u_i \triangleq \{i.in.(c, f, d, j) \mid (c, f, d) \in OP \wedge 0 \leq j < n_u\} \cup \{i.out.m \mid m \in MSG \cup DAT\}.$$ △

Events performed by $u_i$ are either requests to the system (the $i.in$ form) or responses to requests (the $i.out$ form).

We call the system we are to develop $\mathsf{S}$ and its alphabet is the union of its users.

**Definition 5.5** $\alpha\mathsf{S} \triangleq \bigcup USR$. △

We define a preorder, $<$, on users which we use to state the security property. Users are ordered by reference to their subscript.

**Definition 5.6** $\forall u_i, u_j \in USR \bullet u_i < u_j \Leftrightarrow i < j$. △

Then the security policy (an MLS–type specification) is given by Definition 5.7. In this instance users higher in the order $<$ are unable to interfere with lower users.

**Definition 5.7** $\forall\, u_i, u_j \in USR \bullet u_i < u_j \Rightarrow\; u_j \not\rightsquigarrow\; u_i : \mathsf{S}.$ $\triangle$

We define a function *lw* (read as 'last write') which is used to relate a trace (of $\mathsf{S}$) to the current state of a particular user's file–store; *lw j t f* is the set of possible values of file $f$ belonging to user $j$ after trace $t$.

**Definition 5.8**

$$\begin{aligned} &lw\;\; j\; \langle\rangle\; f \triangleq \emptyset; \\ &lw\;\; j\; (t^\frown\langle i.in.(\mathbf{wr}, f, d, j)\rangle)\; f \triangleq (lw\; j\;\; t\;\, f) \cup \{d\}, \qquad i \leq j; \\ &lw\;\; j\; (t^\frown\langle e\rangle)\; f \triangleq lw\; j\;\; t\; f, \qquad e \neq i.in.(\mathbf{wr}, f, d, j) \vee j < i. \end{aligned} \qquad \triangle$$

We define a property that expresses the fact that values returned by **rd** events (that do not violate the restriction 'no read–up') return the correct values. That is, they return a value previously written (by a **wr** that does not violate 'no write–down') to the file being examined, although not necessarily the value of the last write. The intended implementation is a distributed database and we have not given any indication of how the update of such a database should be performed—hence the specification only states that a read must reflect some previous write. In that way the database is consistent; it never returns a value not previously written.

**Definition 5.9** *A trace $t$ (of $\mathsf{S}$) is* retrospective *if for all $u$, $v$ and $w$ such that*

$$t = u^\frown\langle i.in.(\mathbf{rd}, f, x, j)\rangle^\frown v^\frown\langle i.out.m\rangle^\frown w$$

*and* $u_i \cap \mathbf{set}\;\, v = \emptyset$ *then*

$$\begin{aligned} &(m = \mathbf{er}) \\ &\vee \\ &j \leq i \wedge (lw\;\, j\;\, (u^\frown v)\;\, f \neq \emptyset) \Rightarrow m \in lw\;\, j\;\, (u^\frown v)\;\, f. \end{aligned} \qquad \triangle$$

Note how, in Definition 5.9, **rd** events return **er** if no value has been written to the relevant file, or one of the previously written values if at least one write has been engaged in.

Lemma 5.1 shows that if $t$ is a retrospective trace then restricting to the elements of $\alpha\mathsf{S}$ does not affect the property (we use this fact later when hiding the internal construction of $\mathsf{S}$, there the traces of a system composed of subsystems are restricted by hiding the internal events, using subordination):

**Lemma 5.1** *If $t$ is retrospective then so is $t \restriction \alpha \mathsf{S}$.*

**Proof** By inspection of Definition 5.8 we find that $lw\ j\ t = lw\ j\ (t \restriction \alpha \mathsf{S})$.

Consider a retrospective trace $t$ such that

$$t = u \frown \langle i.in.(\mathbf{rd}, f, x, j) \rangle \frown v \frown \langle i.out.m \rangle \frown w$$

and

$$t \restriction \alpha \mathsf{S} = u \restriction \alpha \mathsf{S} \frown \langle i.in.(\mathbf{read}, f, x, j) \rangle \frown v \restriction \alpha \mathsf{S} \frown \langle i.out.m \rangle \frown w \restriction \alpha \mathsf{S}.$$

$t \restriction \alpha \mathsf{S}$ is retrospective since $lw\ j\ (u \restriction \alpha \mathsf{S} \frown v \restriction \alpha \mathsf{S}) = lw\ j\ (u \frown v)$ and $t$ is retrospective. □

The property 'retrospective' is closed under prefix.

**Lemma 5.2** *If $s \frown t$ is retrospective then $s$ is retrospective.* □

Notice how Definitions 5.8 and 5.9 include the restrictions 'no read–up' and 'no write–down'. As we will use Definition 5.9 as the functional specification the inclusion of those two properties is not part of a security specification. Rather they are included as design decisions which will help us to achieve the security specification of Definition 5.7. We do not, for the purposes of development, view those requirements as security policies but as security techniques. Our security policy is more abstract: it is expressed in terms of permitted and unpermitted flows of information through the use of non–interference. In other examples 'no read–up' and 'no write–down' may form the security policy (see, for example, [10]).

If we do not make that design decision it becomes easy to demonstrate traces of $\mathsf{S}$ which violate the non–interference property given in Definition 5.7: any trace in which a user reads a file belonging to a user higher in the order violates Definition 5.7.

The functional specification of $\mathsf{S}$ is that, in the sense of [39], $\mathsf{S}$ satisfies *retrospective*.

The complete specification of $\mathsf{S}$ is the conjunction of Definition 5.7 and *retrospective*. In the following section we develop system $\mathsf{S}$ and show that it does satisfy the security specification of Definition 5.7 and the functional specification *retrospective*.

## 5.3 Development

Here a process $\mathsf{S}$ satisfying the functional specification (*retrospective*) and the security specification (Definition 5.7) is developed. That system

is a distributed database with each node holding all the files belonging to a particular named user. Requests by one user to access files belonging to another user are communicated to the appropriate (remote) file–store. $\mathsf{S}$ is given algebraically using CSP and, in Section 5.4, is coded into occam.

The development starts with a user's file–store (called $\mathsf{F}_i$ for user $u_i$) which is entirely local to $u_i$. Next the process $\mathsf{P}_i$ is developed to handle requests from any user for access to files in $\mathsf{F}_i$. An interface $\mathsf{I}_i$ for each user is produced and the total system consists of all file–stores, processes $\mathsf{P}_i$ and interfaces in parallel.

We start by considering one of the local file–stores. Each file–store is associated with a user and stores the files that belong to that user. The process $\mathsf{F}_i$ is the store appropriate for user $u_i$ and its alphabet consists of communications along channels: $i.j.Fin$ and $i.j.Fout$.

**Definition 5.10**

$$\begin{aligned} \alpha\mathsf{F}_i \triangleq\ & \{i.j.Fin.(c, f, d) \mid (c, f, d) \in OP \wedge u_j \in USR\} \\ & \cup \\ & \{i.j.Fout.m \mid m \in DAT \cup MSG \wedge u_j \in USR\}. \end{aligned} \qquad \triangle$$

The communications along $i.j.Fin$ are **rd** or **wr** commands with a file–name (from $FSP$) and, in the case of **wr**, a data item from $DAT$.

On $i.j.Fout$ messages **ok**, **er** or data items from $DAT$ are communicated in reply to one of the commands.

Initially the store of files, represented by the partial function $s : FSP \nrightarrow DAT$, is empty and each **wr** command alters the appropriate part of that function. The **rd** command returns values dependent on the state of $s$. Functional overriding $\oplus$ is used to denote the update to $s$ (see [92, page 108] and Definition 2.10).

**Definition 5.11** *The local file–store for user $u_i \in USR$ is*

$$\mathsf{F}_i \triangleq \mathsf{F}_i^{\emptyset};$$

$$\mathsf{F}_i^s \triangleq \mathop{[\!]}_{u_j \in USR} i.j.Fin?(c, f, d) \rightarrow \mathsf{F}_i^s(c, f, d, j);$$

$$\mathsf{F}_i^s(\mathbf{rd}, f, d, j) \triangleq (i.j.Fout!s(f) \lhd f \in \mathbf{dom}\ s \rhd i.j.Fout!\mathbf{er}) \rightarrow \mathsf{F}_i^s;$$

$$\mathsf{F}_i^s(\mathbf{wr}, f, d, j) \triangleq i.j.Fout!\mathbf{ok} \rightarrow \mathsf{F}_i^{(s \oplus \{f \mapsto d\})}. \qquad \triangle$$

The set $f_j^i$, of events from $\alpha\mathsf{F}_i$, is the set of events used to *read* files if $u_j > u_i$ and the set used to *write* files if $u_j < u_i$. We use it in Lemma 5.3 to show that transactions that read files in $\mathsf{F}_i$ do not interfere with events used to write files.

$F_j^i$ is the set of completions to transactions in $\mathsf{F}_i$.

**Definition 5.12** $F_j^i \triangleq \{\langle i.j.Fout.m\rangle \mid m \in DAT \cup MSG\} \cup \{\langle\rangle\}$. △

**Definition 5.13**

$$
\begin{aligned}
f_j^i \triangleq\ & \{i.j.Fin.(c,f,d) \mid (c,f,d) \in OP \\
& \qquad\qquad\qquad\qquad \wedge \\
& \qquad\qquad\qquad\qquad (u_j > u_i \Rightarrow c = \mathbf{rd}) \\
& \qquad\qquad\qquad\qquad \wedge \\
& \qquad\qquad\qquad\qquad (u_j < u_i \Rightarrow c = \mathbf{wr})\} \\
& \cup \\
& \{i.j.Fout.m \mid m \in DAT \cup MSG\}.
\end{aligned}
$$
△

When $u_j > u_i$ transactions from $f_j^i$ do not interfere with any other events performed by $\mathsf{F}_i$; similarly when $u_j < u_i$ transactions performed by $\mathsf{F}_i$ do not interfere with events from $f_j^i$.

**Lemma 5.3** *If $u_j > u_i$ then*

$$F_j^i, f_j^i \not\rightsquigarrow (\alpha\mathsf{F}_i \setminus f_j^i) : \mathsf{F}_i$$

*and if $u_j < u_i$ then*

$$F_j^i, (\alpha\mathsf{F}_i \setminus f_j^i) \not\rightsquigarrow f_j^i : \mathsf{F}_i.$$

**Proof** The first assertion follows from Law 4.14 since when $u_j > u_i$ $\mathsf{F}_i$ can be written

$$\mu\, \underline{\mathsf{X}} \bullet (i.j.Fin?(\mathbf{rd}, f, d) \rightarrow \mathsf{F}_i^s(\mathbf{rd}, f, d, j) [\![ \mathsf{G}),$$

for suitable $\mathsf{G}$. The second case follows in a similar manner since complete transactions from $(\alpha\mathsf{F}_i \setminus f_j^i)$ do not interfere with events from $f_j^i$ as $f_j^i$ only contains **wr** events and the return message **ok**. □

We verify that $\mathsf{F}_i$ performs correctly and define the function $lw_{\mathsf{F}_i}$ to reflect the relationship between a trace containing **wr** commands and the function $s$. Compare Definition 5.14 with Definition 5.8 where **wr** events for a file $f$ update a set of possible values; here **wr**s update a single value for each file.

**Definition 5.14**

$$
\begin{aligned}
&lw_{\mathsf{F}_i} &&: \tau\mathsf{F}_i \rightarrow FSP \nrightarrow DAT;\\
&lw_{\mathsf{F}_i} &&\langle\rangle \triangleq \emptyset;\\
&lw_{\mathsf{F}_i} &&(t^\frown\langle i.j.Fin.(\mathbf{wr}, f, d), i.j.Fout.\mathbf{ok}\rangle) \triangleq (lw_{\mathsf{F}_i}\ t) \oplus \{f \mapsto d\};\\
&lw_{\mathsf{F}_i} &&(t^\frown\langle e\rangle) \triangleq lw_{\mathsf{F}_i}\ t,\\
& && \qquad t^\frown\langle e\rangle \neq \mathbf{init}\ t^\frown\langle i.j.Fin.(\mathbf{wr}, f, d), i.j.Fout.\mathbf{ok}\rangle. \quad \triangle
\end{aligned}
$$

The following lemma shows the relationship between $lw_{\mathsf{F}_i}$ and $s$ in the definition of $\mathsf{F}_i$.

**Lemma 5.4** *For all $t \in \tau\mathsf{F}_i$ and some $k$*

$$(\mathsf{F}_i/t = \mathsf{F}_i^s)\ \mathrm{C}\ \#t = 2k\ \mathrm{B}\ (\mathsf{F}_i/t = \mathsf{F}_i^s(c, f, d, j)), \qquad (3)$$

*where $\#t = 2k+1 \Rightarrow$ **last** $t = i.j.Fin.(c, f, d)$ and $s = lw_{\mathsf{F}_i}\ t$.*

**Proof** By induction on the length of a trace $t$.

**(base case)** $t = \langle\rangle$ has even length and $lw_{\mathsf{F}_i}\ \langle\rangle = \emptyset$. The result holds since $\mathsf{F}_i/\langle\rangle = \mathsf{F}_i^{\emptyset}$.

**(inductive step)** Suppose that Equation (3) holds for each trace $t$ such that $\#t \leq n$ (for some $n$), and consider an event $e \in \alpha\mathsf{F}_i$ such that $t' = t^\frown\langle e\rangle$ is a trace of $\mathsf{F}_i$. Examine the cases when $t$ is of even and odd length separately.

**($t$ even)** By hypothesis $\mathsf{F}_i/t = \mathsf{F}_i^s$, with $s = lw_{\mathsf{F}_i}\ t$. Then, for some $c$, $f$, $d$ and $j$, $e = i.Fin.(c, f, d)$, and by examining $\mathsf{F}_i^s$ we find that

$$\mathsf{F}_i/t' = \mathsf{F}_i^s(c, f, d, j),$$

and Equation (3) holds for $t'$, since $lw_{\mathsf{F}_i}\ t' = lw_{\mathsf{F}_i}\ t = s$ in this case.

**($t$ odd)** By hypothesis $\mathsf{F}_i/t = \mathsf{F}_i^s(c, f, d, j)$ and $e = i.j.Fout.m$ for some $m \in MSG \cup DAT$.

If $c = \mathbf{rd}$ then $\mathsf{F}_i/t' = \mathsf{F}_i^s$ by definition of $\mathsf{F}_i^s(\mathbf{rd}, f, d, j)$ and Equation (3) holds since $lw_{\mathsf{F}_i}\ t' = lw_{\mathsf{F}_i}\ t$ (see last line of Definition 5.14).

If $c = \mathbf{wr}$ then by inspection of $\mathsf{F}_i^s(\mathbf{wr}, f, d, j)$ we find

$$\mathsf{F}_i/t' = \mathsf{F}_i^{s\oplus\{f\mapsto d\}} \quad \text{and} \quad m = \mathbf{ok},$$

and by Definition 5.14 (and hypothesis)

$$\begin{aligned} lw_{\mathsf{F}_i}\ t' &= lw_{\mathsf{F}_i}\ (\mathbf{init}\ t)^\frown\langle i.j.Fin.(\mathbf{wr}, f, d), i.j.Fout.\mathbf{ok}\rangle \\ &= lw_{\mathsf{F}_i}\ t \oplus \{f \mapsto d\} \\ &= s \oplus \{f \mapsto d\}. \end{aligned}$$

The result follows by induction. □

We say that a trace is $\mathsf{F}_i$–serial if reading file $f$ returns the last value written to $f$, or **er** if that file has not been written to at all. Compare this with Definition 5.9 where **rd** commands return *one* of the values previously written; the **rd**s in an $\mathsf{F}_i$–serial trace return the value *last* written.

**Definition 5.15** *A trace $t$ is* $\mathsf{F}_i$–serial *if for all $u, v$ such that*

$$t = u^\frown\langle i.j.Fin.(\mathbf{rd}, f, d), i.j.Fout.m\rangle^\frown v$$

*then*

$$m = (lw_{\mathsf{F}_i}\ u\ f\ \mathrm{C}\ f \in \mathbf{dom}\ (lw_{\mathsf{F}_i}\ u)\ \mathrm{B}\ \mathbf{er}).$$ △

A prefix of an $\mathsf{F}_i$–serial trace is also $\mathsf{F}_i$–serial; the proof is evident.

**Lemma 5.5** *If $s^\frown t$ is $\mathsf{F}_i$–serial then $s$ is $\mathsf{F}_i$–serial.* □

Using Lemma 5.4 we show that the traces of $\mathsf{F}_i$ are $\mathsf{F}_i$–serial, i.e. we show that $\mathsf{F}_i$ behaves as we expect.

**Lemma 5.6** $\mathsf{F}_i$ **sat** $\mathsf{F}_i$*–serial.*

**Proof** By induction on the length of a trace $t$.

**(base case)** $\langle\rangle$ is $\mathsf{F}_i$–serial and $\langle\rangle \in \tau\mathsf{F}_i$. Hence the base case holds.

**(inductive step)** The hypothesis is: if $t \in \tau\mathsf{F}_i$ such that $\#t \leq n$ (for some $n$) then $t$ is $\mathsf{F}_i$–serial. Consider the case when $t$ is even and odd separately and put $t' = t^\frown\langle e\rangle$ where $e \in \alpha\mathsf{F}_i$.

- **($t$ even)** By Lemma 5.4 $\mathsf{F}_i/t = \mathsf{F}_i^s$ and $s = lw_{\mathsf{F}_i}\ t$. $t' \in \tau\mathsf{F}_i$ and so $e = i.Fin.(c, f, d)$ for some $c$, $f$ and $d$. Hence $t'$ is trivially $\mathsf{F}_i$–serial by hypothesis.
- **($t$ odd)** By Lemma 5.4 $\mathsf{F}_i/t = \mathsf{F}_i^s(c, f, d, j)$ for some $c$, $f$ and $d$ (with $s = lw_{\mathsf{F}_i}\ t$). By inspection $e = i.j.Fout.m$. If $c = \mathbf{rd}$ then when $f \notin \mathbf{dom}\ s$ we have $m = \mathbf{er}$ and when $f \in \mathbf{dom}\ s$ the message $m = s(f) = lw_{\mathsf{F}_i}\ t$. Hence $t'$ is $\mathsf{F}_i$–serial.

The result follows by induction. $\square$

So far we have developed a simple file–store which services requests along a pair of channels. We now develop a process $\mathsf{P}_i$ which takes requests from any user $u_j$ along channel $i.j.in$ and returns replies along $i.j.out$. Requests which are disallowed, because of the restrictions 'no read–up' and 'no write–down', return the message **er**; but valid requests are passed to the local file–store $\mathsf{F}_i$ and its replies are forwarded to the user who made the request.

**Definition 5.16**

$$\begin{aligned}
\alpha\mathsf{P}_i \triangleq\ & \alpha\mathsf{F}_i \\
& \cup \\
& \{i.j.in.(c, f, d) \mid (c, f, d) \in OP \wedge u_j \in USR\} \\
& \cup \\
& \{i.j.out.m \mid m \in DAT \cup MSG \wedge u_j \in USR\};
\end{aligned}$$

$$\mathsf{P}_i \triangleq \mathop{\square}_{u_j \in USR} i.j.in?(c, f, d) \rightarrow (\mathsf{R}_i^j(f, d) \mathrel{\mathrm{C}} c = \mathbf{rd} \mathrel{\mathrm{B}} \mathsf{W}_i^j(f, d));$$

$$\mathsf{R}_i^j(f, d) \triangleq (i.j.out!\mathbf{er} \rightarrow \mathsf{P}_i \mathrel{\mathrm{C}} j < i \mathrel{\mathrm{B}} i.j.Fin!(\mathbf{rd}, f, d) \rightarrow \mathsf{P}_i^j);$$

$$\mathsf{W}_i^j(f, d) \triangleq (i.j.out!\mathbf{er} \rightarrow \mathsf{P}_i \mathrel{\mathrm{C}} j > i \mathrel{\mathrm{B}} i.j.Fin!(\mathbf{wr}, f, d) \rightarrow \mathsf{P}_i^j);$$

$$\mathsf{P}_i^j \triangleq i.j.Fout?m \rightarrow i.j.out!m \rightarrow \mathsf{P}_i. \qquad \triangle$$

We now begin the development of the system from the viewpoint of security. We define the set $t_j^i$ which contains the events possible along channels $i.j.in$ and $i.j.out$. It represents the channels along which user $u_j$ makes requests of $u_i$.

**Definition 5.17**

$$\begin{aligned}
t_j^i \triangleq\ & \{i.j.in.(c, f, d) \mid (c, f, d) \in OP\} \\
& \cup \\
& \{i.j.out.m \mid m \in DAT \cup MSG\}. \qquad \triangle
\end{aligned}$$

Using that definition we prove that transactions (see Section 4.3.2) with alphabet $(t_j^i \cup f_j^i)$ do not interfere with the events from $t_k^i$, when $k < j$, through the process $\mathsf{P}_i$. These transactions are traces of the form

$$\langle i.j.in.(c, f, d), i.j.Fin.(c, f, d), i.j.Fout.m, i.j.out.m\rangle$$

and

$$\langle i.j.in.(c, f, d), i.j.out.m\rangle,$$

representing the cases of valid and invalid requests (i.e. requests disallowed because of the restrictions on reading and writing).

Definition 5.18 defines the set $T_j^i$, the set of completions to transactions from user $u_j$ using $\mathsf{P}_i$.

**Definition 5.18**

$$\begin{aligned} T_j^i \triangleq\ & \{\langle i.j.Fin.(c,f,d), i.j.Fout.m, i.j.out.m\rangle \mid (c,f,d) \in OP \\ & \qquad \wedge \\ & \qquad (u_j > u_i \Rightarrow c = \mathbf{rd}) \\ & \qquad \wedge \\ & \qquad (u_j < u_i \Rightarrow c = \mathbf{wr}) \wedge m \in DAT \cup MSG\} \\ & \cup \\ & \{\langle i.j.Fout.m, i.j.out.m\rangle \mid m \in DAT \cup MSG\} \\ & \cup \\ & \{\langle i.j.out.m\rangle \mid m \in DAT \cup MSG\} \\ & \cup \\ & \{\langle\rangle\}. \end{aligned}$$ △

Transactions from $(t_j^i \cup f_j^i)$ do not interfere with $t_k^i$ when $u_k < u_j$.

**Lemma 5.7** *For all $u_i, u_j, u_k \in USR$ such that $u_k < u_j$*

$$T_j^i, (t_j^i \cup f_j^i) \not\leadsto t_k^i : \mathsf{P}_i.$$

**Proof** The result follows from Law 4.14 since $\mathsf{P}_i$ can be written (for suitable $\mathsf{Q}$):

$$\mu\, \underline{\mathsf{X}} \bullet (i.j.in?(c,f,d) \rightarrow (\mathsf{R}_i^j(f,d) \lhd c = \mathbf{rd} \rhd \mathsf{W}_i^j(f,d)) [\!] \mathsf{Q}),$$

and $T_j^i, (t_j^i \cup f_j^i) \not\leadsto t_k^i : \mathsf{Q}$ by Lemma 4.22 part 6. □

Process $\mathsf{P}_i$ runs in parallel with the file–store $\mathsf{F}_i$ to form the process $\mathsf{C}_i$. Below we hide the synchronization between $\mathsf{F}_i$ and $\mathsf{P}_i$ leaving only the 'external' communications exposed, but first $\mathsf{C}_i$ is used to verify that the combination of $\mathsf{F}_i$ and $\mathsf{P}_i$ work as expected.

**Definition 5.19** $\mathsf{C}_i \triangleq \mathsf{F}_i \| \mathsf{P}_i$. △

We showed in Lemma 5.6 that the traces of $\mathsf{F}_i$ are $\mathsf{F}_i$–serial and are able to prove a similar result about $\mathsf{C}_i$. In $\mathsf{C}_i$'s case, however, the serial nature of the traces is complicated by the fact that some **wr** commands are disallowed by $\mathsf{P}_i$.

We start by defining $lw_{\mathsf{C}_i}$ the function which relates a trace from $\tau\mathsf{C}_i$ to the effect that trace has on the initial (empty) file–store. That function is closely related to the 'last write' function for $\mathsf{F}_i$, see below.

**Definition 5.20**

$$\begin{array}{ll}
lw_{\mathsf{C}_i} & : \tau\mathsf{C}_i \to FSP \nrightarrow DAT; \\
lw_{\mathsf{C}_i} & \langle\rangle \triangleq \emptyset; \\
lw_{\mathsf{C}_i} & (t^\frown\langle i.j.in.(\mathbf{wr}, f, d), i.j.Fin.(\mathbf{wr}, f, d), i.j.Fout.\mathbf{ok}\rangle), \\
& \qquad \triangleq (lw_{\mathsf{C}_i}\ t) \oplus \{f \mapsto d\}, \qquad j \leq i; \\
lw_{\mathsf{C}_i} & (t^\frown\langle e\rangle) \triangleq lw_{\mathsf{C}_i}\ t, \qquad \textit{otherwise.}
\end{array} \quad \triangle$$

Clearly the functions $lw_{\mathsf{C}_i}$ and $lw_{\mathsf{F}_i}$ are related. Lemma 5.8 shows how:

**Lemma 5.8** $lw_{\mathsf{C}_i}\ t = lw_{\mathsf{F}_i}\ (t \upharpoonright \alpha\mathsf{F}_i)$.

**Proof** Trivial observation of Definitions 5.14 and 5.20. □

We use Definition 5.20 to define a $\mathsf{C}_i$–serial trace. Such a trace must reflect **wr** commands in the way that an $\mathsf{F}_i$–serial trace does, whilst allowing for the restrictions on access from unauthorized users.

**Definition 5.21** *A trace $t$ is* $\mathsf{C}_i$–serial *if for all $u, v$ such that* $t = u^\frown\langle i.j.in.(\mathbf{rd}, f, d), i.j.out.m\rangle^\frown v$ *then*

$$m = \mathbf{er} \quad \wedge \quad j < i$$

*and if for all $u, v$ such that*

$$t = u^\frown\langle i.j.in.(\mathbf{rd}, f, d), i.j.Fin.(\mathbf{rd}, f, d), i.j.Fout.m, i.j.out.m\rangle^\frown v$$

*then*

$$j \geq i \wedge m = (lw_{\mathsf{C}_i}\ u\ f\ \mathrm{C}\ f \in \mathbf{dom}\ (lw_{\mathsf{C}_i}\ u)\ \mathrm{B}\ \mathbf{er}). \quad \triangle$$

If a trace is $\mathsf{C}_i$–serial then so are all its prefixes.

**Lemma 5.9** *If $s^\frown t$ is $\mathsf{C}_i$–serial then $s$ is $\mathsf{C}_i$–serial.* □

Lemma 5.10 demonstrates that the traces of $\mathsf{C}_i$ are $\mathsf{C}_i$–serial.

**Lemma 5.10** $\mathsf{C}_i$ **sat** $\mathsf{C}_i$*–serial.*

**Proof** By induction on the length of a trace $t$.

**(base case)** As $\langle\rangle$ is $\mathsf{C}_i$–serial, $\langle\rangle \in \tau\mathsf{C}_i$ the base case holds.

**(inductive step)** The hypothesis is: if $t \in \tau\mathsf{C}_i$ such that $\#t \leq n$ (for some $n$) then $t$ is $\mathsf{C}_i$–serial. Consider three possible cases based on the last event in the trace $t$ and let $t' = t ^\frown \langle e \rangle$ be a trace of $\mathsf{C}$.

**(last** $t = i.j.in.(c, f, d)$**)** If $c = \mathbf{rd}$ then, by examining $\mathsf{P}_i$, we find $e = i.j.out.\mathbf{er}$ when $i > j$ and when $i \leq j$ $e$ is a communication with $\mathsf{F}_i$ (i.e. $e = i.j.Fin.(c, f, d)$) and so $t'$ is $\mathsf{C}_i$–serial by hypothesis.

**(last** $t = i.j.out.m$ or **last** $t = i.j.Fin.(c, f, d)$**)** In both cases $t'$ is $\mathsf{C}_i$–serial by hypothesis since neither of the next events possible for $e$ (either $i.j.in.(c, f, d)$ or $i.j.Fout.d$) add any cases to $t$ which need checking against Definition 5.21.

**(last** $t = i.j.Fout.m$**)** By examining $\mathsf{P}_i$ and elementary laws of $\|$ we find that $e = i.j.out.m$ and so (for some $s$; $s$ is $\mathsf{C}_i$–serial by Lemma 5.9) $t'$ is

$$s ^\frown \langle i.j.in.(c, f, d), i.j.Fin.(c, f, d), i.j.Fout.m, i.j.out.m \rangle.$$

Hence $t' \upharpoonright \alpha\mathsf{F}_i \in \tau\mathsf{F}_i$ since $t' \in \tau\mathsf{C}_i$ and $t' \upharpoonright \alpha\mathsf{F}_i$ is $\mathsf{F}_i$–serial by Lemma 5.6. So by Definitions 5.15, 5.20 and 5.21 and Lemma 5.8 $t'$ is $\mathsf{C}_i$–serial.

The result follows by induction. □

Lemma 5.7 showed that transactions for a particular user did not interfere with other users in $\mathsf{P}_i$; Lemma 5.11 shows the same fact for $\mathsf{C}_i$—the synchronisation with $\mathsf{F}_i$ has not introduced any new communication.

**Lemma 5.11** *For all* $u_i, u_j, u_k \in USR$ *such that* $u_k < u_j$

$$T^i_j, (t^i_j \cup f^i_j) \not\leadsto t^i_k : \mathsf{C}_i.$$

**Proof** By Law 4.9 (applied twice, once for each of the cases $u_i > u_j$ and $u_j > u_i$) and Lemmas 5.3 and 5.7. □

The communication between $\mathsf{P}_i$ and $\mathsf{F}_i$ in $\mathsf{C}_i$ is entirely local as $\alpha\mathsf{F}_i$ is a subset of $\alpha\mathsf{P}_i$ and hence we define a process $\mathsf{D}_i$, which is $\mathsf{C}_i$ with the alphabet of $\mathsf{F}_i$ hidden (shown here by subordination [39, page 161]).

**Definition 5.22** $\mathsf{D}_i \triangleq \mathsf{F}_i \,//\, \mathsf{P}_i$. △

Hiding the alphabet of $\mathsf{F}_i$ reduces the length of the transactions performed in $\mathsf{C}_i$ to pairs of events along $i.j.in$ and $i.j.out$ in $\mathsf{D}_i$. Consequently the non–interference assertion in Lemma 5.11 is similarly simplified.

**Lemma 5.12** *For all $u_i, u_j, u_k \in USR$ such that $u_k < u_j$*

$$(T^i_j \setminus \alpha\mathsf{F}_i), t^i_j \not\rightsquigarrow t^i_k : \mathsf{D}_i.$$

**Proof** By Law 4.5 and Lemma 5.11, since $t^i_k \setminus \alpha\mathsf{F}_i = t^i_k$,

$$(t^i_j \cup f^i_j) \setminus \alpha\mathsf{F}_i = t^i_j$$

and $\mathsf{D}_i = \mathsf{C}_i \setminus \alpha\mathsf{F}_i$. □

$t_j$ is the union of the $t^i_j$ over all $i$:

**Definition 5.23** $t_j \triangleq \bigcup_{u_i \in USR} t^i_j$. △

The interleaving of transactions from $T^i_j \setminus \alpha\mathsf{F}_i$ (over all $i$) is called $T_j$ (recall Definition 4.8).

**Definition 5.24** $T_j \triangleq \mathop{|}_{u_i \in USR} (T^i_j \setminus \alpha\mathsf{F}_i)$. △

The parallel composition of the $\mathsf{D}_i$ is called $\mathsf{D}$.

**Definition 5.25** $\mathsf{D} \triangleq \mathop{\|}_{u_i \in USR} \mathsf{D}_i$. △

Putting the $\mathsf{D}_i$ in parallel gives transaction non–interference between appropriate $u_j$ and $u_k$.

**Lemma 5.13** *For all $u_j, u_k \in USR$ such that $u_k < u_j$: $T_j, t_j \not\rightsquigarrow t_k : \mathsf{D}$.*

**Proof** By Law 4.7 and Lemma 5.12 we have

$$(T^0_j \setminus \alpha\mathsf{F}_0) \mid (T^1_j \setminus \alpha\mathsf{F}_1), (t^0_j \cup t^1_j) \not\rightsquigarrow (t^0_k \cup t^1_k) : (\mathsf{D}_0 \| \mathsf{D}_1).$$

Continue using Law 4.7 to obtain

$$T_j, t_j \not\rightsquigarrow t_k : \mathsf{D}$$

(see Definitions 5.23 and 5.24). □

Here we have (transaction) non–interference between the channels used by one user ($u_j$) to make requests and the channels used by another ($u_k$). We do not have non–interference between these two sets as the transactions from $T_j$ are not of length zero (see Lemma 4.20). To overcome this problem we add the interface described below. Then the system will exhibit non–interference, the interface design having removed interference by denial of service without using an infinite buffer (see comments on non–interference and infinite buffers in [64]). Infinite buffers can be used to assure non–interference by providing a buffer which, as it is never full, does not cause communication from receiver to sender as the receiver can never prevent the sender from placing more information in the buffer.

The final constituent of the system is the interface provided to a user $u_i$. We have already given (in Definition 5.4) the set of events that user $u_i$ expects to be able to engage in at his interface.

The process $\mathsf{I}_i$ provides the interface for user $u_i$, engaging in input events on $i.in$ and returning results on $i.out$. It communicates with the appropriate user (via the processes $\mathsf{D}_j$) and communicates the result of those events to the user.

**Definition 5.26**

$$\mathsf{I}_i \triangleq i.in?(c, f, d, j) \to j.i.in!(c, f, d) \to j.i.out?m \to i.out!m \to \mathsf{I}_i;$$

$$\alpha\mathsf{I}_i \triangleq u_i \ \cup \ t_i. \qquad \triangle$$

We next define the process which represents an individual user $u_i$: $\mathsf{T}_i$ combines the file–store (and associated machinery of $\mathsf{P}_i$) with the interface $\mathsf{I}_i$.

**Definition 5.27** $\mathsf{T}_i \triangleq \mathsf{I}_i \| \mathsf{D}_i$. $\triangle$

The parallel combination of these users will form the system ($\mathsf{S}$) with the internal communications exposed (see Definition 5.31).

**Definition 5.28** $\mathsf{T} \triangleq \mathop{\|}\limits_{u_i \in USR} \mathsf{T}_i$. $\triangle$

Function $lw_{\mathsf{T}}$ relates a trace to its effect on the file–store of a particular user.

**Definition 5.29**

$$lw_{\mathsf{T}} \quad : \tau\mathsf{T} \to USR \to FSP \nrightarrow DAT;$$

$$\begin{array}{lll} lw_{\mathsf{T}}\ j & \langle\rangle \triangleq \emptyset; & \\ lw_{\mathsf{T}}\ j & (t^\frown\langle i.j.in.(\mathbf{wr}, f, d)\rangle^\frown t'^\frown\langle i.j.out.\mathbf{ok}\rangle) & \\ & \quad \triangleq (lw_{\mathsf{T}}\ (t^\frown t')) \oplus \{f \mapsto d\}, & j \le i \wedge t^i_j \cap \mathbf{set}\ t' = \emptyset; \\ lw_{\mathsf{T}}\ j & (t^\frown\langle e\rangle) \triangleq lw_{\mathsf{T}}\ j\ t, & e \ne i.j.out.\mathbf{ok}. \quad \triangle \end{array}$$

Function $(lw_{\mathsf{T}}\ j)$ is equivalent to $lw_{\mathsf{C}_j}$ and $(lw_{\mathsf{T}}\ j\ t\ f)$ is a member of $(lw\ j\ t\ f)$ if $(lw\ j\ t\ f)$ is non–empty. (When $(lw\ j\ t\ f)$ is empty $(lw_{\mathsf{T}}\ j\ t\ f)$ is undefined).

**Lemma 5.14** *$lw_{\mathsf{T}}\ j = lw_{\mathsf{C}_j}$ and*

$$f \in \mathbf{dom}\ lw_{\mathsf{T}}\ j\ t \Rightarrow (lw_{\mathsf{T}}\ j\ t\ f) \in (lw\ j\ t\ f).$$

**Proof** Obvious from Definitions 5.20 and 5.29 and from Definitions 5.8 and 5.29. □

Using Definition 5.29 we say that a trace of $\mathsf{T}$ is *serial* if each read reflects the value last written to that file, for each user. This definition is stronger then the definition of 'retrospective' as here the value returned from a read is the value of the last write; that fact is reflected in Lemma 5.17.

**Definition 5.30** *A trace $t$ is* serial *if for all $u$, $v$ and $w$ such that*

$$t = u^\frown\langle i.j.in.(\mathbf{rd}, f, x)\rangle^\frown v^\frown\langle i.j.out.m\rangle^\frown w$$

*and $\alpha\mathsf{I}_i \cap \mathbf{set}\ v = \emptyset$ then*

$$\begin{array}{l} m = \mathbf{er} \\ \vee \\ j \ge i \wedge (f \in \mathbf{dom}\ lw_{\mathsf{T}}\ j\ (u^\frown v) \Rightarrow m = lw_{\mathsf{T}}\ j\ (u^\frown v)\ f). \end{array} \quad \triangle$$

The traces of $\mathsf{T}$ are serial.

**Lemma 5.15** $\mathsf{T}$ **sat** *serial.*

**Proof** By induction on a trace $t$ of $\tau\mathsf{T}$.

**(base case)** $\langle\rangle$ is serial and $\langle\rangle$ is in $\tau\mathsf{T}$.

**(inductive step)** The hypothesis is that for some $n$

$$\forall\, t \in \tau\mathsf{T} \bullet \#t \le n \Rightarrow t \text{ is serial;}$$

and put $t' = t^\frown\langle e\rangle$. Assume $t'$ is a trace of $\mathsf{T}$. There are two cases to consider.

($e = i.j.out.m$) As $t'$ is a trace of $\mathsf{T}$ there exist $u$ and $v$ such that

$$t' = u^\frown\langle i.j.in.(\mathbf{rd}, f, d)\rangle^\frown v^\frown\langle i.j.out.m\rangle$$

and $t^i_j \cap \mathbf{set}\ v = \emptyset$. $t' \upharpoonright \alpha\mathsf{C}_i$ is a trace of $\mathsf{C}_i$ and by Lemma 5.10 it is $\mathsf{C}_i$–serial. So $t'$ is serial by Definition 5.21 and Lemma 5.14.

($e \neq i.j.out.m$) In this case the fact that $t'$ is serial follows from the hypothesis.

The result follows by induction. □

Note that

$$\mathsf{T} = \mathop{\|}_{u_i \in USR} \mathsf{T}_i = \mathsf{D} \| (\mathop{\|}_{u_i \in USR} \mathsf{I}_i)$$

and we have already shown a non–interference property for $\mathsf{D}$ in Lemma 5.13. We show in Lemma 5.16 the effect of adding the interfaces $\mathsf{I}_i$ to $\mathsf{D}$.

**Lemma 5.16** *For all $u_j, u_k \in USR$ such that $u_k < u_j$*

$$T_j, (t_j \cup u_j) \not\rightsquigarrow (t_k \cup u_k) : \mathsf{T}.$$

**Proof** By Lemma 5.13 and Law 4.8 we obtain

$$T_j, t_j \not\rightsquigarrow (t_k \cup u_k) : (\mathsf{D} \| \mathsf{I}_k)$$

since $\alpha\mathsf{I}_k \cap \alpha\mathsf{D} \subseteq \alpha\mathsf{D} \setminus t_j$ (condition (4.8.7)).

Then use Law 4.8 again to get

$$T_j, (t_j \cup u_j) \not\rightsquigarrow (t_k \cup u_k) : (\mathsf{D} \| \mathsf{I}_j \| \mathsf{I}_k),$$

since $\alpha\mathsf{D} \cap \alpha\mathsf{I}_j \subseteq t_j$ (condition (4.8.3)) and

$$\forall\, t \in \tau\mathsf{D}\, \exists\, r \in T_j \bullet \mathbf{com}_{\mathsf{D}}(t, r) \Rightarrow (t^\frown r) \upharpoonright \alpha\mathsf{I}_j \in \tau\mathsf{I}_j$$

(condition (4.8.4)).

Lemma 4.22 part 1 gives $\emptyset \not\rightsquigarrow \alpha\mathsf{I}_i : \mathsf{I}_i$, so use Law 4.9 for each $\mathsf{I}_i$ with $u_i \neq u_k$ and $u_i \neq u_j$ to get $T_j, (t_j \cup u_j) \not\rightsquigarrow (t_k \cup u_k) : \mathsf{T}$ as required, since $(t_j \cup u_j) \cap \alpha\mathsf{I}_i = \emptyset$ and $\alpha\mathsf{I}_i \setminus (t_j \cup u_j) = \alpha\mathsf{I}_i$. □

If a trace is serial then it is also retrospective. That is, if in a trace each read reflects the last write then each read also reflects at least one of the preceding writes.

**Lemma 5.17** *If $t \in \tau\mathsf{T}$ is serial then it is retrospective.*

**Proof** Take a trace $t$ of $\mathsf{T}$. By Lemma 5.15 we know that $t$ is serial. Consider any $u$, $v$ and $w$ such that

$$t = u^\frown\langle i.j.in.(\mathbf{rd}, f, x)\rangle^\frown v^\frown\langle i.j.out.m\rangle^\frown w$$

and the conditions on $m$ from Definition 5.30 hold.

$t$ is retrospective as $lw_{\mathsf{T}}\ j\ (u^\frown v)\ f \in lw\ j\ (u^\frown v)\ f$ (by Lemma 5.14) when $lw_{\mathsf{T}}\ j\ (u^\frown v)\ f$ is defined and $m = \mathbf{er}$ otherwise. □

The system consists of the users operating in parallel with the communication along the channels $i.j.in$ and $i.j.out$ concealed. That leaves the alphabet of $\mathsf{S}$ as the collection of user interfaces (see Definition 5.5).

**Definition 5.31** $$\mathsf{S} \triangleq \mathsf{T} \setminus (\bigcup_{u_i \in USR} \alpha\mathsf{D}_i).$$ △

Finally, we show that $\mathsf{S}$ acts as the database we originally specified.

**Theorem 5.1** $\mathsf{S}$ **sat** *retrospective.*

**Proof** For any trace $t$ of $\mathsf{S}$ there exists $s \in \tau\mathsf{T}$ such that $s$ is serial (see Lemma 5.15) and $s \restriction \alpha\mathsf{S} = t$. So by Lemma 5.17 $s$ is retrospective and by Lemma 5.1 $t$ is retrospective. □

And we show that $\mathsf{S}$ satisfies the security property of Definition 5.7.

**Theorem 5.2** *For all $u_j, u_k \in USR$ such that $u_k < u_j$: $u_j \not\rightsquigarrow u_k : \mathsf{S}$.*

**Proof** Repeated application of Law 4.5 to the result of Lemma 5.16 gives

$$\{\langle\rangle\}, u_j \not\rightsquigarrow u_k : \mathsf{S}$$

and the result follows from Lemma 4.20. □

## 5.4 occam Implementation

The translation of $\mathsf{S}$ from CSP into occam (see [41, 56]) is standard. Each of the processes $\mathsf{D}_i$, $\mathsf{F}_i$, $\mathsf{I}_i$, $\mathsf{P}_i$, $\mathsf{S}$ and $\mathsf{T}_i$ are given a process definition and the overall structure of the code is shown in Figure 3.

Note the nesting of `procI` and `procD` within `procT`, and `procF` and `procP` within `procD`, so that the hidden channels from the original CSP

```
...definitions

PROC procT(CHAN in, out, rep[], req[], VALUE i)

  PROC procI(CHAN in, out, rep[], req[], VALUE i)
    ...interface for user i

  PROC procD(CHAN rep[], req[], VALUE i)

    PROC procF(CHAN in[], out[])
      ...the local file-store for user i

    PROC procP(CHAN rep[], req[], in[], out[], VALUE i)
      ...the command processor for user i

    ...process commands and access local file-store for i

  ...user i is interface and command processor in parallel

PROC procS(CHAN in[], out[])
  ...all users in parallel
```

Figure 3: Outline of occam version of S.

processes are hidden within the process definitions in the implementation. The communication between users is hidden within `procS` giving the external channels `in[]` and `out[]` as the communication points with the system.

The standard CSP–to–occam translation of ? to ?, ! to !, $\rightarrow$ to `SEQ`, $\|$ to `PAR`, $[\!]$ to `ALT` (although see Section 5.5.2) and $\mu \mathsf{X} \bullet \mathsf{F}(\mathsf{X})$ to `WHILE TRUE` is shown below. The channels $i.j.in$ and $i.j.out$ are replaced by two arrays of channels

$$\texttt{req}[\texttt{nu} * \texttt{nu}] \quad \text{and} \quad \texttt{rep}[\texttt{nu} * \texttt{nu}],$$

with the correspondence

$$i.j.in \leftrightarrow \texttt{req}[\texttt{nu} * \texttt{i} + \texttt{j}] \quad \text{and} \quad i.j.out \leftrightarrow \texttt{rep}[\texttt{nu} * \texttt{i} + \texttt{j}].$$

The correctness of that translation is verified by the following lemma.

**Lemma 5.18** *If $i.j.in$ and $k.l.in$ are channels from* S *with $i \neq k$ or $j \neq l$ then, in the* occam*,*

$$\texttt{req}[\texttt{nu} * \texttt{i} + \texttt{j}] \neq \texttt{req}[\texttt{nu} * \texttt{k} + \texttt{l}].$$

*Similarly for* `rep`.

**Proof** If $i \neq k$ or $j \neq l$ then $n_u * i + j \neq n_u * k + l$ and the result follows. Similarly for `rep`. □

A similar translation of the channels $i.in$, $i.out$, $i.j.Fin$ and $i.j.Fout$ is made in giving the definition of `procS` and `procT` using the arrays

$$\texttt{in}[\texttt{nu}], \quad \texttt{out}[\texttt{nu}], \quad \texttt{F.in}[\texttt{nu}], \quad \text{and} \quad \texttt{F.out}[\texttt{nu}]$$

with the correspondences

$$i.in \leftrightarrow \texttt{in}[\texttt{i}], \qquad i.out \leftrightarrow \texttt{out}[\texttt{i}],$$

$$i.j.Fin \leftrightarrow \texttt{F.in}[\texttt{j}] \quad \text{and} \quad i.j.Fout \leftrightarrow \texttt{F.out}[\texttt{j}].$$

As in Lemma 5.18 the correspondence for channels $i.in$, $i.out$, $i.j.Fin$ and $i.j.Fout$ is verified as correct.

**Lemma 5.19** $\texttt{in}[\texttt{i}] \neq \texttt{in}[\texttt{j}]$, $\texttt{out}[\texttt{i}] \neq \texttt{out}[\texttt{j}]$, $\texttt{F.in}[\texttt{i}] \neq \texttt{F.in}[\texttt{j}]$ *and* $\texttt{F.out}[\texttt{i}] \neq \texttt{F.out}[\texttt{j}]$ *when* $i \neq j$. □

As occam does not provide a data structure more complex than a vector and our definition of the events is structured we translate both of the requests to a vector of natural numbers as follows.

| request | occam slice |
|---|---|
| $(\mathbf{rd}, f, x, j)$ | $\langle \texttt{Rd}, f, x, j \rangle$ |
| $(\mathbf{wr}, f, d, j)$ | $\langle \texttt{Wr}, f, d, j \rangle$ |

with the definitions `Rd = 0` and `Wr = 1` given in the implementation.

The messages are defined in a similar way.

| message | occam DEF |
|---|---|
| **ok** | `Ok = -1` |
| **er** | `Er = -2` |

These definitions are then used with slice communication to perform the communications required in the CSP in a single event. (See Section 5.5 below for further comments on the choice of slice communication over ; for multiple sequential I/O.)

The partial function $s$ from Section 5.3 used to store values is represented by the array `s[nm]` and an undefined element of the domain of $s$, $i \in FSP$ say, is represented by `s[i] = -1`.

The correctness of that representation is verified by Lemma 5.20, which checks the translation of the initialisation and update of $s$. The relationship between `s[i]` and $s(i)$ is given by

$$\forall\, i \in \mathit{FSP} \bullet \mathtt{s}[\mathtt{i}] = (s(i) \lhd i \in \mathbf{dom}\ s \rhd -1).$$

**Lemma 5.20** *Under the representation above*

- $s := \emptyset$ *(see Definition* 5.11*) is correctly implemented by*

```
SEQ f = [0 FOR nm]
  s[f] := -1,
```

- *and* $s := s \oplus \{f \mapsto d\}$ *(see* $\mathsf{F}_i^s(\mathbf{wr}, f, d, j)$*) is correctly implemented by* `s[f] := d`.

## Proof

- $s := \emptyset$ achieves $\forall\, i \in \mathit{FSP} \bullet i \in \mathbf{dom}\ s$, as does the `SEQ` loop given.
- $s := s \oplus \{f \mapsto d\}$ only changes $s(f)$ to $d$ as does `s[f] := d`. □

---

```
DEF nf = 20,
    nu = 10,
    nd = 100 :

DEF Rd = 0,
    Wr = 1,
    Ok = -1,
    Er = -2 :

PROC procT(CHAN in, out, rep[], req[], VALUE i) =

  PROC procI(CHAN in, out, rep[], req[], VALUE i) =

    VAR cmd[4],
        msg,
        adr :

    WHILE TRUE
      SEQ
        in ? cmd[0 FOR 3]
        adr := (nu * cmd[3]) + i
        IF
```

```
        (cmd[0] <> Rd) AND (cmd[0] <> Wr)
          out ! Er
        (cmd[1] < 0) OR (cmd[1] >= nf)
          out ! Er
        (cmd[2] < 0) OR (cmd[2] >= nd)
          out ! Er
        (cmd[3] < 0) OR (cmd[3] >= nu)
          out ! Er
        TRUE
          SEQ
            req[adr] ! cmd[0 FOR 2]
            rep[adr] ? msg
            out ! msg :

PROC procD(CHAN rep[], req[], VALUE i) =

  CHAN F.in[nu], F.out[nu] :

  PROC procF(CHAN in[], out[]) =

    VAR s[nf],
        cmd[3] :

    SEQ
      SEQ f = [0 FOR nf]
        s[f] := -1
      WHILE TRUE
        ALT j = [0 FOR nu]
          in[j] ? cmd[0 FOR 2]
            IF
              cmd[0] = Rd
                IF
                  s[cmd[1]] < 0
                    out[j] ! Er
                  TRUE
                    out[j] ! s[cmd[1]]
              cmd[0] = Wr
                SEQ
                  s[cmd[1]] := cmd[2]
                  out[j] ! Ok :

  PROC procP(CHAN rep[], req[], in[], out[], VALUE i) =

    VAR cmd[3],
        msg,
```

```
        adr :

      PAR
        WHILE TRUE
          ALT j = [0 FOR nu]
            req[(nu * i) + j] ? cmd[0 FOR 2]
              SEQ
                adr := (nu * i) + j
                IF
                  cmd[0] = Rd
                    IF
                      j < i
                        rep[adr] ! Er
                      j >= i
                        SEQ
                          in[j] ! cmd[0 FOR 2]
                          out[j] ? msg
                          rep[adr] ! msg
                  cmd[0] = Wr
                    IF
                      j > i
                        rep[adr] ! Er
                      j <= i
                        SEQ
                          in[j] ! cmd[0 FOR 2]
                          out[j] ? msg
                          rep[adr] ! msg :

    PAR
      procF(F.in, F.out)
      procP(rep, req, F.in, F.out, i) :

  PAR
    procI(in, out, rep, req, i)
    procD(rep, req, i) :

PROC procS(CHAN in[], out[]) =

  CHAN rep[nu * nu], req[nu * nu] :

  PAR i = [0 FOR nu]
    procT(in[i], out[i], rep, req, i) :
```

occam implementation of process S.

## 5.5 Possible Flaws

We know, through the development given in Section 5.3, that the process S satisfies the MLS property given by Definition 5.7. That development, while giving *an* assurance of security, only gives an assurance as far as our model of security goes (e.g. as we have not considered time we do not expect the assurance to include time). By examining the code above and the process of its translation from CSP we can make three major observations about possible security breaches which may occur if this occam process were used. Section 5.5.1 covers the process of translation and its inherent problems and Section 5.5.2 deals with specific areas where this system (process S or procS) may be considered 'insecure.'

The third possible flaw concerns the extent of the model. Whilst we can be sure of the security properties of the CSP process S (and to some extent those of the occam process procS) the model (and the development of the system) has not encompassed the occam compiler or the system on which the code may be executed. For full security assurance they must be taken into account too.

### 5.5.1 Errors of Translation

This section describes possible errors in the translation from CSP to occam. In fact, the code given in Section 5.4 compiles and runs as expected.

**slice communication** As noted above the requests are converted to slice communications, defining the types as the numbers 0 and 1. An alternative would have been to translate $i.j.in?(c, f, d)$ to

```
req[nu * i + j] ? c ; f ; d
```

which is the same as

```
SEQ
  req[nu * i + j] ? c
  req[nu * i + j] ? f
  req[nu * i + j] ? d.
```

This would be incorrect as it would be the same as changing the CSP process by replacing $i.j.in?(c, f, d)$ with

$$i.j.in?c \rightarrow i.j.in?f \rightarrow i.j.in?d.$$

That change would require a redevelopment of S, as a single event has been replaced by three events, and would invalidate the proof and conclusion of security given by Theorem 5.2.

**type checking** In the example above there is explicit checking of input data in `procI` which is implied in the definitions of the types *CMD*, *USR*, *FSP* and *DAT* given in Definition 5.2. This addition to `procI` causes the reply `Er` to be returned on occasions when incorrect data is entered. This change adds to the interactions that are possible along $i.in$ and $i.out$ as the CSP constrains the values to those defined in the types. occam makes no such constraint and thus checking of the values is required.

Although in this case the additions are necessary to prevent the system from operating incorrectly due to incorrect input data, such type–checking needs to be added carefully to prevent the introduction of security problems. These checks must be added to the development earlier on—i.e. in the CSP process S.

Careful checking of the way in which types are implemented is required.

**structure** The channels $i.j.in$, $i.j.out$, $i.in$, $i.out$, $i.j.Fin$ and $i.j.Fout$ have been translated into vectors of channels as described in Section 5.4. The correspondence between them is demonstrated in Lemmas 5.18 and 5.19 and the correctness of the representation of $s$ is verified by Lemma 5.20. The establishment of such correspondences is vital since the implementation cannot be considered correct without them.

**transliteration** In such a short process it is easy to examine each line and check for errors in translation. In larger, and more realistic, examples such checking is close to impossible and many bugs may be introduced (see [5], particularly Chapter 2).

### 5.5.2 Security Breaches

The model we are using has not mentioned time; it has been concerned with untimed observations (although see Chapter 6). Many authors have noted concern about security problems caused by not examining time (see, for an example of response–time causing security problems, [95, pages 297–298]). We agree with that concern and, for that reason, specifically give the analysis of `procS` shown below. We have separated the formal development from the analysis as our development does not encompass timed observations.

Our implementation can exhibit covert communication from a high user to a low user due to time, caused by delaying a user's access to a particular file–store. Example 5.1 illustrates the problem.

**Example 5.1** Consider the case when S has only two users: $u_0$ and $u_1$. In that case we have that $u_1 \not\rightsquigarrow u_0 : \mathsf{S}$ as guaranteed by the development of Section 5.3.

But repeated attempts by $u_1$ to read a file belonging to $u_0$ can result in covert communication from $u_1$ to $u_0$ in the form of the delayed response $u_0$ gets from his own file–store. So, if $u_1$ repeatedly engages in the event $1.in.(\mathbf{rd}, 1, x, 0)$ for example, he may be able to delay $u_0$'s access to his file–store.

User $u_0$ is only able to detect the covert communication by reference to a real–time clock and as our model does not reflect time such communication has not been ruled out by the development. △

The problem of covert communication through delayed response is not captured in the development of process S as there is no mention of time in our model of security. It appears primarily because we now make timed observations of the occam (with our attention on the role of ALT), whereas in the CSP we made untimed observations (and did not consider the role of $\|$ with respect to time). The ALT gives equal priority to requests coming from each user. As each request may take some time to perform, the delay exhibited in Example 5.1 is realized.

In order to solve this problem it is sufficient to do one of two things: we could rework the model of security to include time and redevelop S; or we could give a condition of fairness on the choice made by ALT. The latter solution is the most practical.

The fairness we require is that each user is given equal access to the file–stores in the limit. The limit equation, for two users $u_0$ and $u_1$, is

$$\lim_{\#t \to \infty} (t \downarrow 0.out - t \downarrow 1.out) = 0 \qquad (4)$$

and expresses the fact that in the limit $u_0$ and $u_1$ must have received an equal number of replies to their requests.

Achieving this type of requirement is a concern of scheduling algorithms (see [95]) and we suggest three occam–specific solutions.

In order to understand how one user can 'starve' another of access we need to examine the way in which the ALT operator works in occam. In [4, page 42] the authors give the following equivalence (based on the semantics of the operators):

```
PAR
  PRI PAR 1 <= i <= n          PAR
    ci!                          PRI PAR 2 <= i <= n
  PRI ALT 1 <= i <= n     ≡        c1!
    ci?                          P1
      Pi
```

and the authors note that

> The PRI PAR may be explicit in the program, or it may be implemented by the user who interacts with the program.

We use the fact that the PRI PAR occurs when a user repeatedly attempts access of a file–store to give users access in a specific order in the following examples. That equivalence shows how the two PRI ALT's below achieve the fairness we require. The limit holds as we are able to give specific users access in a specific order, by choosing the order correctly we can achieve the limit.

**random selection** Suppose that the channel random provides random numbers in the range 0 to $n_u - 1$ with the condition that each number is equally likely to occur. Then the following piece of occam can replace the ALT of procS to provide fair distribution of requests.

```
random ? j
PRI ALT j1 = [0 FOR nu]
  req[nu * i + (j + j1) \ nu] ? cmd
    ...service request
```

Its primary difficulty lies in developing a suitable random number generator. In practice such a generator would have to be shown to be sufficiently random to prevent the undesired communication by time; a uniform distribution of the numbers ensures that the limit equation (4) holds. Exactly what is meant by 'sufficiently random' will depend on the acceptable bandwidth of the channel caused by time.

**round robin** This solution works by changing the ALT to the following code which gives each user priority of access in turn.

```
SEQ j = [0 FOR nu]
  PRI ALT j1 = [0 FOR nu]
    req[nu * i + (j + j1) \ nu] ? cmd
      ...service request
```

That will prevent one user being able to starve the others of access by servicing requests in order of user. In this case (and in the last) users still slow the system down by their use of the file–stores, but Equation (4) obviously holds.

**insertion of delays** A more complex version is achieved by inserting delays to simulate requests when there have been none. The requests for users are serviced (as in round robin, above) in order of user and if a particular user does not make a request within some fixed time the next user is polled.

```
VAR clock :
DEF delay = 10 :

...definitions

SEQ j = [0 FOR nu]
  SEQ
    clock := NOW
    ALT
      req[nu * i + j] ? cmd
        ...service request
      WAIT NOW AFTER clock + delay
        SKIP
```

This solution has the disadvantage of adding extra delay to the system and finding the correct value of `delay` will require analysis of the system in operation.

The limit holds as each user is polled in turn (much in the same way as in round robin).

The solution to the problem of channels outside the security specification or development is dependent on the acceptable bandwidth of those channels and such channels need to be carefully analysed in order to determine their bandwidth (see classes B2 and A1 in [76]).

# Chapter 6

# Timed Non–interference

## 6.1 Introduction

In Section 5.5.2 we saw that a system proved secure using the laws from Section 4.3.3 could, in practice, be insecure if observations outside the security model were made. In particular we focussed on the role of timed observations in the security of a system. In this chapter we show how to incorporate time into the formalism for non–interference. Thus $v$–equivalence (Definition 3.7) and non–interference (Definition 3.9) are extended to include timed observations. As a result a system proved non–interfering using the laws of this chapter will not exhibit interference even when timed observations are made.

To introduce time to the definition of non–interference we use the semantic models of Timed CSP (TCSP), choosing to use timed traces to define information flow and non–interference. TCSP is described below and in [79] and [80, 81].

We develop the necessary theory to capture timed non–interference; prove several laws; and apply the laws to reason about an example system.

## 6.2 Timed Security

In Section 6.2.1 two examples motivate the definition of information flow in TCSP. In order to give those examples, we present a little of the notation of TCSP (more can be found in [89]).

The abstract syntax of TCSP (Definition 6.1) is almost that of untimed CSP (Definition 2.11). The parallel operator now has two forms: $\|$ for the composition of processes which must synchronize on all events; and ${}_X\|_Y$ where the processes only synchronize on events from $X \cap Y$.

In addition there is the *delay operator*, $\mathrm{Wait}\ \ $, which is used to give a specific delay.

**Definition 6.1** *TCSP has the abstract syntax:*

$$\mathsf{S} \ ::= \ \mathrm{Stop}_{\ A} \mid \mathrm{Wait}\ \ t \mid (e \rightarrow \mathsf{S}) \mid (b : B \rightarrow \mathsf{S}_b) \mid \ \mathsf{S} \| \mathsf{T} \mid \mathsf{S}\ {}_X\|_Y\ \mathsf{T} \mid \\ \mathsf{S} \sqcap \mathsf{T} \mid \mathsf{S} [\!] \mathsf{T} \mid \mathsf{S} \setminus C. \qquad \triangle$$

As in CSP, TCSP events are instantaneous, occurring in co–operation with the environment. We note the four assumptions in [21] concerning timing in a distributed system: there is a non–zero lower bound, $\delta$, on the length of time interval between any two events in the history of a sequential process; there is no lower bound on the time interval between two independent actions, such as those performed by two processes running asynchronously in parallel; the times at which events occur in the system relate to a conceptual global clock: time passes at the same rate in each process; and hidden events occur as soon as they become available.

TCSP is unalphabetized; the set of all events is denoted $\Sigma$ and parallel composition is handled by the process operators $\|$ and ${}_X\|_Y$. $\mathsf{S}\|\mathsf{T}$ is the parallel composition of $\mathsf{S}$ and $\mathsf{T}$ with synchronization on all events; $\mathsf{S}\ {}_X\|_Y\ \mathsf{T}$ means that $\mathsf{S}$ can only perform events from $X$ and $\mathsf{T}$ can only perform events from $Y$ and the processes synchronize on events in $X \cap Y$.

The traces semantics of CSP (Definition 2.14) is easily extended to include time. In untimed CSP the elements of a trace are events; in timed CSP they are (*time*, *event*) pairs. The *time* is the time at which the *event* occurs; times in traces are non–decreasing. Additionally an event which occurs at the first time it is available is flagged by placing a hat on it: the event $e$ becomes $\hat{e}$.

The set of all (timed) traces drawn from the set $\Sigma$ is denoted $T\Sigma^*_{\leq}$.

Some notation is used in later definitions; these definitions are taken from [79, 89].

**Definition 6.2** *A trace can be* delayed *by a time t; because the resulting operator is strict and distributive it suffices to define it on singleton traces:*

$$\langle (t_1, e) \rangle + t \triangleq \langle (t_1 + t, e) \rangle.$$

*The* temporal shift *of a trace is given by the strict and distributive operator defined on the singleton trace by:*

$$\langle (t_1, e) \rangle \dot{-} t \triangleq (\langle (t_1 - t, e) \rangle \ \mathrm{C}\ t_1 \geq t \ \mathrm{B}\ \langle\rangle).$$

*A trace is* unhatted *if none of the events in it are hatted; the strict and distributive operator* $\tilde{}$ *removes hats from events in a trace. Its effect on the singleton trace is:*

$$\langle(t,\hat{e})\rangle\tilde{} = \langle(t,e)\rangle\tilde{} \triangleq \langle(t,e)\rangle.$$

*The* set of timed events *in which a process* $\mathsf{S}$ *can engage is:*

$$\sigma\mathsf{S} \triangleq \{e \mid \exists\, s \in \tau\mathsf{S}\, \exists\, t \in [0,\infty) \bullet \langle(t,e)\rangle \textbf{ in } \tilde{s}\}.$$

*The* parallel composition *of traces is given by* ${}_X\|_Y$*:*

$$s_0 \;{}_X\|_Y\; s_1 \triangleq \{s \mid s \upharpoonright X = s_0 \wedge s \upharpoonright Y = s_1 \wedge s \upharpoonright (X \cup Y) = s\}.$$

*When traces* $s_0$ *and* $s_1$ *are such that* $\tilde{s}_0 = \tilde{s}_1$ *then* $s \triangleq s_0 \wr s_1$ *is such that* $\tilde{s} = \tilde{s}_0$ *and the* $n$*th element of* $s$ *is hatted iff the* $n$*th element of* $s_0$ *or* $s_1$ *is hatted.*

*The time at which the first event occurs in a trace is:*

$$\textbf{begin}\;(\langle(t,e)\rangle^\frown s) \triangleq t. \qquad \triangle$$

Definition 6.3 summarises the parts of the timed traces model that we use; these definitions are taken from [79, pages 43–44]. We reuse the notation $\tau\mathsf{S}$ (Definition 2.14) to denote the set of timed traces of system $\mathsf{S}$.

After an event has occurred there is a fixed time $\delta$ before any further events can occur. In Definition 6.3 that $\delta$ delay is specifically included in the definitions of prefixing.

**Definition 6.3**

$$\begin{array}{lcl}
\tau\mathtt{Stop}_{\;A} & \triangleq & \{\langle\rangle\}; \\
\tau(\mathtt{Wait}\quad t\,;\mathsf{S}) & \triangleq & \{s + t \mid s \in \tau\mathsf{S}\}; \\
\tau(e \to \mathsf{S}) & \triangleq & \{(0,\langle\rangle)\} \\
 & & \cup \\
 & & \{\langle(0,\hat{e})\rangle^\frown(s+\delta) \mid s \in \tau\mathsf{S}\} \\
 & & \cup \\
 & & \{\langle(t,e)\rangle^\frown(s+t+\delta) \mid s \in \tau\mathsf{S} \wedge t \geq 0\}; \\
\tau(b : B \to \mathsf{S}_b) & \triangleq & \{(0,\langle\rangle)\} \\
 & & \cup \\
 & & \{\langle(0,\hat{b})\rangle^\frown(s+\delta) \mid s \in \tau\mathsf{S} \wedge b \in B\} \\
 & & \cup \\
 & & \{\langle(t,b)\rangle^\frown(s+t+\delta) \mid s \in \tau\mathsf{S} \wedge b \in B \wedge t \geq 0\};
\end{array}$$

$$
\begin{array}{lll}
\tau(\mathsf{S} [\!] \mathsf{T}) & \triangleq & \tau\mathsf{S} \cup \tau\mathsf{T}; \\
\tau(\mathsf{S} \sqcap \mathsf{T}) & \triangleq & \tau\mathsf{S} \cup \tau\mathsf{T}; \\
\tau(\mathsf{S} \| \mathsf{T}) & \triangleq & \{s_0 \wr s_1 \mid s_0 \in \tau\mathsf{S} \wedge s_1 \in \tau\mathsf{T} \wedge \tilde{s}_0 = \tilde{s}_1\}; \\
\tau(\mathsf{S}_X \|_Y \mathsf{T}) & \triangleq & \{s \mid \exists\, s_0 \in \tau\mathsf{S}\, \exists\, s_1 \in \tau\mathsf{T} \bullet s \in (s_0\ {}_X\|_Y\ s_1)\}; \\
\tau f(\mathsf{S}) & \triangleq & \{f^*(s) \mid s \in \tau\mathsf{S}\}; \\
\tau(\mathsf{S} \setminus C) & \triangleq & \{s|_C \mid s \in \tau\mathsf{S} \wedge s \text{ is } C\text{–active}\};
\end{array}
$$

*where $s$ is $C$–*active *if there are no events of the form $(t, e)$ in $s$; all events in $s$ must be hatted: $(t, \hat{e})$.* △

## 6.2.1 Examples

In this section we provide two examples to motivate the definition of timed non–interference. The first shows how a system considered secure when timed observations are not made can be insecure, in practice, if timed observations are made. The second example is used later to illustrate the laws, it demonstrates a simple system for encrypting data and shows how the time taken for encryption can allow an eavesdropper to determine the type of message passing along a channel.

**Example 6.1** Consider again the system L in Example 3.1. Example 3.3 showed that $a \not\rightsquigarrow c : \mathsf{L}$ where $a = \{a.0, a.1\}$ and $c = \{ce\}$. Suppose that after any switch is pressed there is a delay of $\delta$ during which no other switch may be depressed, i.e. both the other switches are locked for time $\delta$. Then the situation is described by system L in TCSP:

$$
\begin{array}{lll}
\mathsf{L} & \triangleq & \mathsf{L}_0 \\
\mathsf{L}_i & \triangleq & (a.i \rightarrow \mathsf{L}_{\neg i} \mid b.i \rightarrow \mathsf{L}_{\neg i} \mid ce \rightarrow \mathsf{L}_{\neg i}).
\end{array}
$$

Consider the (timed) traces $\langle\rangle$ and $\langle(1, a.0)\rangle$; the first represents no activity, the second the case when $a$ presses his switch at time 1.

In the case of $\langle\rangle$ user $c$ can perform the event $ce$ at any time; after the trace $\langle(1, a.0)\rangle$ $c$ can only perform $ce$ after time $1 + \delta$. Hence, by examining timing information, $c$ can tell the difference between $a$ having performed the event $a.0$ and not having performed any events. So when time is taken into account, i.e. in TCSP, L should not be non–interfering: $a$ can communicate with $c$. △

Our second example describes an encryption system where encrypted messages are sent across channel *chan*. Encryption takes time $t$ (and is explicitly included using the Wait construction); when there are no messages to send, the channel is stuffed with a bogus message. Bogus

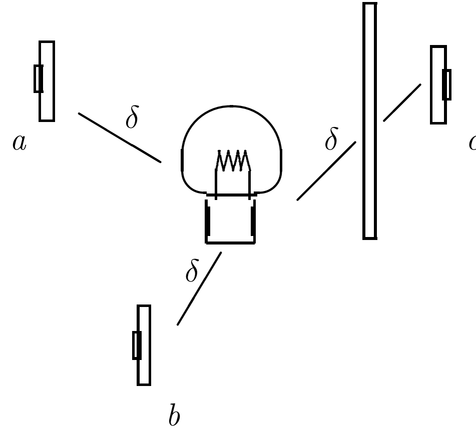


Figure 4: Process L with delays.

messages do not take time $t$ to generate and Example 6.2 shows how an eavesdropper can use that fact to distinguish genuine and bogus messages.

**Example 6.2** Suppose that $M$ is the set of possible messages and $C$ is the set of possible encrypted messages. The function $f : M \rightarrow C$ is used to encrypt messages. We assume encryption takes time $t$. When no messages need be sent a random encrypted message from $C$ is selected without delay.

S describes the behaviour of such a system.

$$\begin{aligned} \mathsf{S} \;\triangleq\; & in?x \rightarrow \mathrm{Wait}\ \ t\ ; chan!f(x) \rightarrow \mathrm{Stop}_{\ \alpha \mathsf{S}} \\ & \llbracket \\ & \sqcap_{c \in C}\ chan!c \rightarrow \mathrm{Stop}_{\ \alpha \mathsf{S}}. \end{aligned}$$

Consider the view of an eavesdropper who has tapped *chan* and can see all the messages passing along it. Assuming the eavesdropper cannot reverse the encryption $f$, consider whether he can gain any information from the messages passed along *chan*.

The insertion of messages when there are no messages to encrypt guarantees that the eavesdropper doesn't gain information from knowing that a message was sent; the channel always has traffic on it, but unfortunately consideration of timing can allow him to determine when a 'real' message is being sent. Suppose that at time 0 a message $m \in M$ is sent along *in* and then at time $t + \delta$ the event $chan.f(m)$ occurs; i.e. S performs the trace

$$\langle (0, in.m), (t + \delta, chan.f(m)) \rangle.$$

If $(0, in.m)$ has not happened the trace $\langle(0, chan.c)\rangle$ can occur and the eavesdropper can tell that the message $c$ is bogus as no encrypted message can occur before time $t + \delta$. $\triangle$

### 6.2.2 Timed $v$–equivalence

In order to define non–interference in TCSP we first define when two timed traces appear identical to a user. A user is (cf. Definition 3.1) a collection of events that form the user's interface; as there is no alphabet for a process those events are drawn from the universal alphabet.

**Definition 6.4** *A* user $v$ *is a set* $v \subseteq \Sigma$*;* $v$ *is called the* interface. $\triangle$

Two timed traces are equivalent at the interface of user $v$ (cf. Definition 3.4) if there are no subsequent traces (at $v$) that can distinguish them.

**Definition 6.5** *If in system* $\mathsf{S}$ *with user* $v$ *and* $s$, $s'$ *in* $T\Sigma^*_\leq$

$$\forall\, r \in ([0, \infty) \times v^*) \bullet (s^\frown r \in \tau\mathsf{S} \Leftrightarrow s'^\frown r \in \tau\mathsf{S})$$

*then* $s$ *and* $s'$ *are* $v$–equivalent*, written* $s \approx^{\mathsf{S}}_v s'$. $\triangle$

In Section 3.2.2 we showed $\approx^{\mathsf{S}}_v$ (Definition 3.7) to be an equivalence relation (Lemma 3.2); the same applies for $\approx^{\mathsf{S}}_v$ as given by Definition 6.5.

**Lemma 6.1** $\approx^{\mathsf{S}}_v$ *is an equivalence relation.* $\square$

User $u$ does not interfere with $v$ if the presence or absence of $u$'s events does not affect the view of the system at $v$'s interface; that view now includes timing information (cf. Definition 3.9).

**Definition 6.6** *If system* $\mathsf{S}$ *has disjoint users* $u$ *and* $v$*,* $u$ *is* (timed) non–interfering *with* $v$ $(u \not\rightsquigarrow v : \mathsf{S})$ *iff* $\forall\, s \in \tau\mathsf{S} \bullet s \approx^{\mathsf{S}}_v s|_u$. $\triangle$

In Definitions 6.5 and 6.6 we have reused the symbols $\approx^{\mathsf{S}}_v$ and $\not\rightsquigarrow$ to to define $v$–equivalence and non–interference with reference to time; throughout the remainder of the chapter those symbols refer to the definitions given in this chapter, not the untimed versions.

In order to illustrate those definitions reconsider the encryption system in Example 6.2; Example 6.3 gives a version of it in which *in* $\not\rightsquigarrow$ *chan* and hence the eavesdropper listening to *chan* cannot gain information about whether a real or bogus message has been sent (recall

that we assumed that the eavesdropper cannot invert the encryption $f$).

**Example 6.3** Let the process $\mathsf{T}$ be the system that engages in a communication along *chan* at random:

$$\mathsf{T} \triangleq \bigsqcap_{c \in C} chan!c \rightarrow \mathtt{Stop}_{\alpha \mathsf{S}}.$$

Then the encryption system $\mathsf{S}$ (Example 6.2) can be rewritten:

$$\begin{array}{rl} \mathsf{S} \triangleq & in?x \rightarrow \bigsqcap_{t' \geq t} \mathtt{Wait}\ t' ; (chan!f(x) \rightarrow \mathtt{Stop}_{\alpha \mathsf{S}}) \sqcap \mathsf{T} \\ & \interleave \\ & \bigsqcap_{t' \geq t + \delta} \mathtt{Wait}\ t' ; \mathsf{T}. \end{array}$$

Now the system is only willing to engage in an event from $\mathsf{T}$ after a random delay, and the random delay before encrypting hides whether or not a real message has been sent; hence $in \not\rightsquigarrow chan : \mathsf{S}$ and the eavesdropper does not know whether a real or a bogus message has been sent. In Example 6.4 below we use the laws presented in the next section to verify that this definition of $\mathsf{S}$ is non–interfering. $\triangle$

### 6.2.3 Some Laws of Timed Non–interference

In this section we prove several laws for timed non–interfering systems. Those laws (in many cases) imply the laws of untimed non–interference in Section 4.3.3, but proofs are given for the timed versions for clarity.

Just as Lemma 4.22 highlighted some of the simple properties of untimed non–interference, so Lemma 6.2 gives similar simple properties of timed non–interference. Parts 1 and 2 show that if either user's interface is empty then non–interference is trivial; part 3 shows that $\mathtt{Stop}_A$ is non–interfering; part 4 shows that contracting the interface of a user does not introduce interference; parts 5 and 6 show that if $\mathsf{S}$ is never willing to engage in events from $u$ or from $v$ then $\mathsf{S}$ is non–interfering.

**Lemma 6.2**

1. $\emptyset \not\rightsquigarrow v : \mathsf{S}$;
2. $\forall\, t \in \tau\mathsf{S} \bullet t|_u \in \tau\mathsf{S} \Rightarrow u \not\rightsquigarrow \emptyset : \mathsf{S}$;
3. $u \not\rightsquigarrow v : \mathtt{Stop}_A$;

4. $(u \not\rightsquigarrow v : \mathsf{S}) \wedge (v_0 \subseteq v) \Rightarrow u \not\rightsquigarrow v_0 : \mathsf{S}$;

5. $u \cap \sigma\mathsf{S} = \emptyset \Rightarrow u \not\rightsquigarrow v : \mathsf{S}$;

6. $(v \cap \sigma\mathsf{S} = \emptyset) \wedge (\forall\, t \in \tau\mathsf{S} \bullet t|_u \in \tau\mathsf{S}) \Rightarrow u \not\rightsquigarrow v : \mathsf{S}$.

**Proof** Parts 1 to 4 follow in the same way as parts 1 to 4 of Lemma 4.22.

Part 5 follows in the same way as part 6 of Lemma 4.22 since

$$\forall\, t \in \tau\mathsf{S} \bullet t|_u \in \tau\mathsf{S}$$

follows from the fact that $u \cap \sigma\mathsf{S} = \emptyset$.

Part 6 follows in the same way as part 7 of Lemma 4.22 since

$$\forall\, t \in \tau\mathsf{S} \bullet \mathsf{S}/t \equiv_v^A \mathrm{Stop}_A$$

follows from the fact that $v \cap \sigma\mathsf{S} = \emptyset$. □

TCSP includes a specific construct for introducing a delay: Wait . Law 6.1 shows that delaying a system by an arbitrary time $t$ does not introduce interference.

**Law 6.1** (*wait*)

$$\frac{(6.1.1)\ u \not\rightsquigarrow v : \mathsf{S}}{u \not\rightsquigarrow v : (\mathrm{Wait}\ \ t\,; \mathsf{S})}$$

**Proof** We verify Definition 6.6 for $(\mathrm{Wait}\ \ t\,;\mathsf{S})$. If $s \in \tau(\mathrm{Wait}\ \ t\,;\mathsf{S})$ then $s \dot{-} t \in \tau\mathsf{S}$; take $r \in [0,\infty) \times v^*$:

$$\begin{array}{lll}
 & (s^\frown r) \in \tau(\mathrm{Wait}\ \ t\,;\mathsf{S}) & \\
\Leftrightarrow & (s^\frown r) \dot{-} t \in \tau\mathsf{S} & [\text{Definition 6.3}] \\
\Leftrightarrow & (s \dot{-} t)^\frown(r \dot{-} t) \in \tau\mathsf{S} & [\dot{-}\ \text{distributive}] \\
\Leftrightarrow & (s \dot{-} t)|_u{}^\frown(r \dot{-} t) \in \tau\mathsf{S} & [u \not\rightsquigarrow v : \mathsf{S}] \\
\Leftrightarrow & (s|_u \dot{-} t)^\frown(r \dot{-} t) \in \tau\mathsf{S} & [\ |_u\ \text{distributes through}\ \dot{-}] \\
\Leftrightarrow & (s|_u{}^\frown r) \dot{-} t \in \tau\mathsf{S} & [\dot{-}\ \text{distributive}] \\
\Leftrightarrow & (s|_u{}^\frown r) \in \tau(\mathrm{Wait}\ \ t\,;\mathsf{S}). & [\text{Definition 6.3}]
\end{array}$$
□

Law 4.1 showed that prefixing a system by an event not in $u$ does not introduce interference, the same is true in the case of timed non–interference; Law 6.2 is the relevant law.

**Law 6.2** (*timed choice–1*)

$$\begin{array}{l} \text{(6.2.1)}\ \forall\, b \in B \bullet u \not\rightsquigarrow v : \mathsf{S}_b \\ \text{(6.2.2)}\ B \cap u = \emptyset \\ \hline u \not\rightsquigarrow v : (b : B \rightarrow \mathsf{S}_b) \end{array}$$

**Proof** If $s \in \tau(b : B \rightarrow \mathsf{S}_b)$, $r \in [0, \infty) \times v^*$ and $s = \langle b \rangle ^\frown s'$ for some $s'$ then

$$\begin{array}{lll} & (s ^\frown r) \in \tau(b : B \rightarrow \mathsf{S}_b) & \\ \Leftrightarrow & (s' ^\frown r) \in \tau \mathsf{S}_b & \text{[Definition 6.3; some } b \in B] \\ \Leftrightarrow & (s'|_u {}^\frown r) \in \tau \mathsf{S}_b & [u \not\rightsquigarrow v : \mathsf{S}_b] \\ \Leftrightarrow & (s|_u {}^\frown r) \in \tau(b : B \rightarrow \mathsf{S}_b). & \text{[Definition 6.3; } B \cap u = \emptyset] \quad \Box \end{array}$$

Law 6.2 shows how to introduce events not in $u$'s interface and in the untimed case Law 4.2 showed how to introduce events from $u$. Law 4.2 does not apply when considering timed non–interference. Here is a counter–example.

Consider the system $\mathsf{T}$ where $u \not\rightsquigarrow v : \mathsf{S}$ and $ue \in u$.

$$\mathsf{T} \triangleq (ue \rightarrow \mathsf{S} [\!] \mathsf{S})$$

is non–interfering (untimed) by Law 4.2, but the traces $\langle (0, ue) \rangle$ and $\langle\rangle$ $(= \langle (0, ue) \rangle|_u)$ need not be $v$–equivalent when considering time since the delay of $\delta$ after $ue$ means there is a period during which events from $\mathsf{S}$ cannot be performed.

The problem of the $\delta$ delay is solved by causing the system to delay for a random time before performing any event from $\mathsf{S}$. Then user $v$ will not be able to tell whether the delay was caused by $u$ engaging in the event $ue$ or by the system delaying at random.

Note that the random delay is required, replacing $\mathsf{T}$ by

$$\mathsf{T}_0 \triangleq (ue \rightarrow \mathsf{S} [\!] \mathtt{Wait}\ \ \delta\, ; \mathsf{S})$$

would not solve the problem of interference. In $\mathsf{T}_0$ the traces $\langle (1, ue) \rangle$ and $\langle\rangle$ are not $v$–equivalent since during the period $[1, 1 + \delta)$ no $v$–events (from $\mathsf{S}$) are available, but after $\langle\rangle$ $v$ events are available during $[1, 1 + \delta)$.

**Definition 6.7** *The process* $\mathsf{W}_t$ *waits for any time greater than, or equal to,* $t$ *at random:*

$$\mathsf{W}_t \triangleq \bigsqcap_{t' \geq t} \mathtt{Wait}\ \ t'. \qquad \triangle$$

Note that $\text{Wait} \quad t_0 \,; \mathsf{W}_t = \mathsf{W}_{t+t_0} = \mathsf{W}_t \,; \text{Wait} \quad t_0$ for any $t$ and $t_0$.

Hence $\mathsf{T}$, above, can be made non–interfering by the inclusion of specific delays using the process $\mathsf{W}_t$:

$$\mathsf{T}' \triangleq (ue \rightarrow \mathsf{W}_0 \,; \mathsf{S} \talloblong \mathsf{W}_\delta \,; \mathsf{S}).$$

Law 6.3 incorporates that idea to give the timed version of Law 4.2.

**Law 6.3** (*timed choice–2*)

$$\begin{array}{l}
(6.3.1)\ \forall\, b \in B \bullet u \not\rightsquigarrow\ v : \mathsf{S}_b \\
(6.3.2)\ B \subseteq u \\
(6.3.3)\ \mathsf{R} = (\underset{b \in B}{\talloblong} \mathsf{S}_b) \\
(6.3.4)\ B \cap \iota\mathsf{R} = u \cap \iota\mathsf{R} = \emptyset \\
\hline
u \not\rightsquigarrow\ v : (b : B \rightarrow \mathsf{W}_0 \,; \mathsf{S}_b) \talloblong (\mathsf{W}_\delta \,; \mathsf{R})
\end{array}$$

**Proof** Put $\mathsf{T} = (b : B \rightarrow \mathsf{W}_0 \,; \mathsf{S}_b) \talloblong (\mathsf{W}_\delta \,; \mathsf{R})$. $B \cap \iota\mathsf{R} = \emptyset$ means that $(b : B \rightarrow \mathsf{W}_0 \,; \mathsf{S}_b)$ and $(\mathsf{W}_\delta \,; \mathsf{R})$ have no non–empty traces in common. Consider $s \in \tau\mathsf{T}$ and $r \in [0, \infty) \times v^*$:

(case $(s ^\frown r) \in \tau(b : B \rightarrow \mathsf{W}_0 \,; \mathsf{S}_b)$, $s = \langle b \rangle ^\frown s'$)

$$\begin{array}{lll}
& (s ^\frown r) \in \tau\mathsf{T} & \\
\Leftrightarrow & (s' ^\frown r) \in \tau(\mathsf{W}_0 \,; \mathsf{S}_b) & [\text{this case}] \\
\Leftrightarrow & (s'|_u ^\frown r) \in \tau(\mathsf{W}_0 \,; \mathsf{S}_b) & [u \not\rightsquigarrow\ v : \mathsf{S}] \\
\Leftrightarrow & (s'|_u ^\frown r) \in \tau(\mathsf{W}_\delta \,; \mathsf{S}_b) & [\textbf{begin}\ (s'|_u ^\frown r) \geq \delta] \\
\Leftrightarrow & (s|_u ^\frown r) \in \tau(\mathsf{W}_\delta \,; \mathsf{S}_b) & [B \subseteq u] \\
\Leftrightarrow & (s|_u ^\frown r) \in \tau\mathsf{T}. & [(6.3.4)]
\end{array}$$

(case $(s ^\frown r) \in \tau(\mathsf{W}_\delta \,; \mathsf{R})$)

$$\begin{array}{lll}
& (s ^\frown r) \in \tau\mathsf{T} & \\
\Leftrightarrow & (s ^\frown r) \in \tau(\mathsf{W}_\delta \,; \mathsf{R}) & [\text{this case}] \\
\Leftrightarrow & (s|_u ^\frown r) \in \tau(\mathsf{W}_\delta \,; \mathsf{R}) & [u \not\rightsquigarrow\ v : \mathsf{S}] \\
\Leftrightarrow & (s|_u ^\frown r) \in \tau\mathsf{T}. & [(6.3.4)] \quad \Box
\end{array}$$

The law for changing the alphabet of a system (Law 4.4) applies to TCSP systems as well; rather than changing the alphabet, the function $f$ changes the names of all events in which a system is willing to engage. Law 6.4 is the law for changing the name of events in TCSP; the proof follows in a similar manner to that of Law 4.4.

**Law 6.4** (*change alphabet*)

$$\frac{\begin{array}{l}(6.4.1)\ u \not\rightsquigarrow v : \mathsf{S} \\ (6.4.2)\ f\ \mathit{is\ injective}\end{array}}{f(u) \not\rightsquigarrow f(v) : f(\mathsf{S})}$$ □

Hiding events of a non–interfering system does not introduce interference when considering time; although events are hidden, changes in the timing information are not made since the time delays (either of $\delta$ or by Wait 's) are not removed.

**Law 6.5** (*hiding*)

$$\frac{\begin{array}{l}(6.5.1)\ u \not\rightsquigarrow v : \mathsf{S} \\ (6.5.2)\ C \subseteq \alpha\mathsf{S}\end{array}}{(u \setminus C) \not\rightsquigarrow (v \setminus C) : (\mathsf{S} \setminus C)}$$

**Proof** Take $s \in \tau(\mathsf{S} \setminus C)$ and $r \in [0, \infty) \times (v \setminus C)^*$; we check Definitions 6.5 and 6.6.

$$\begin{array}{lll} & (s \frown r) \in \tau(\mathsf{S} \setminus C) & \\ \Leftrightarrow & \exists (s' \frown r') \in \tau\mathsf{S} \bullet s'|_C = s \wedge r'|_C = r & \text{[Definition 6.3]} \\ \Leftrightarrow & (s'|_u \frown r') \in \tau\mathsf{S} & [u \not\rightsquigarrow v : \mathsf{S}] \\ \Leftrightarrow & (s'|_u \frown r')|_C \in \tau(\mathsf{S} \setminus C) & \text{[Definition 6.3]} \\ \Leftrightarrow & (s|_{(u \setminus C)} \frown r) \in \tau(\mathsf{S} \setminus C). & [s'|_C = s;\ r'|_C = r] \end{array}$$ □

The law for parallelism is identical for that for untimed non–interference (Law 4.6): if both S and T are non–interfering then $(\mathsf{S} \| \mathsf{T})$ is non–interfering.

Law 6.7 applies for alphabetized parallel. Note how the restrictions caused by using the alphabetized parallel operator: users $u$ and $v$ must be in the shared alphabet. That is the case in both Laws 4.6 and 6.7, it must be explicitly stated for the alphabetized operator.

**Law 6.6** (*parallel*)

$$\frac{\begin{array}{l}(6.6.1)\ u \not\rightsquigarrow v : \mathsf{S} \\ (6.6.2)\ u \not\rightsquigarrow v : \mathsf{T}\end{array}}{u \not\rightsquigarrow v : (\mathsf{S} \| \mathsf{T})}$$

**Proof** If $s \in \tau(\mathsf{S} \| \mathsf{T})$ and $r \in [0, \infty) \times v^*$ then

$$(s \frown r) \in \tau(\mathsf{S} \| \mathsf{T})$$

$$\begin{aligned}
&\Leftrightarrow \quad \exists\, s_0 \in \tau\mathsf{S}\, \exists\, s_1 \in \tau\mathsf{T} \bullet (s^\frown r) = s_0 \wr s_1 && \text{[Definition 6.3]}\\
&\Leftrightarrow \quad \exists\, s_0 \in \tau\mathsf{S}\, \exists\, s_1 \in \tau\mathsf{T} \bullet (s|_u{}^\frown r) = (s_0|_u) \wr (s_1|_u)\\
& && [u \not\rightsquigarrow v : \mathsf{S};\ u \not\rightsquigarrow v : \mathsf{T};\ r \in [0,\infty) \times v^*]\\
&\Leftrightarrow \quad (s|_u{}^\frown r) \in \tau(\mathsf{S} \| \mathsf{T}). && \text{[Definition 6.3]} \quad \square
\end{aligned}$$

**Law 6.7** (*alphabetized parallel*)

$$\begin{array}{l}
(6.7.1)\ u \not\rightsquigarrow v : \mathsf{S}\\
(6.7.2)\ u \not\rightsquigarrow v : \mathsf{T}\\
(6.7.3)\ u \cup v \subseteq X \cap Y\\
\hline
u \not\rightsquigarrow v : (\mathsf{S}_X \|_Y \mathsf{T})
\end{array}$$

**Proof** If $s \in \tau(\mathsf{S}_X \|_Y \mathsf{T})$ and $r \in [0, \infty) \times v^*$ then

$$\begin{aligned}
& \quad (s^\frown r) \in \tau(\mathsf{S}_X \|_Y \mathsf{T})\\
&\Leftrightarrow \quad (\exists\, s_0 \in \tau\mathsf{S}\, \exists\, s_1 \in \tau\mathsf{T} \bullet (s^\frown r) \in (s_0\ {}_X\|_Y\ s_1) && \text{[Definition 6.3]}\\
&\Leftrightarrow \quad (s|_u{}^\frown r) \in (s_0|_u)\ {}_X\|_Y\ (s_1|_u) && [u \not\rightsquigarrow v : \mathsf{S};\ u \not\rightsquigarrow v : \mathsf{T}]\\
&\Leftrightarrow \quad (s|_u{}^\frown r) \in \tau(\mathsf{S}_X \|_Y \mathsf{T}). && \text{[Definition 6.3]} \quad \square
\end{aligned}$$

The external choice of two processes is non–interfering if neither of them engages in an event from $u$ initially. The same applies for internal choice as internal and external choice are not distinguished by the timed traces model (see Definition 6.3).

**Law 6.8** (*external choice*)

$$\begin{array}{l}
(6.8.1)\ u \not\rightsquigarrow v : \mathsf{S}\\
(6.8.2)\ u \not\rightsquigarrow v : \mathsf{T}\\
(6.8.3)\ \forall\, s \in \tau\mathsf{S} \cap \tau\mathsf{T}\, \forall\, r \in [0,\infty) \times v^* \bullet (s^\frown r) \in \tau\mathsf{S}\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \Leftrightarrow\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad (s^\frown r) \in \tau\mathsf{T}\\
\hline
u \not\rightsquigarrow v : (\mathsf{S} \,\square\, \mathsf{T})\\
u \not\rightsquigarrow v : (\mathsf{S} \sqcap \mathsf{T})
\end{array}$$

**Proof** Take $s \in \tau(\mathsf{S} \,\square\, \mathsf{T})$ and $r \in [0, \infty) \times v^*$ and verify Definition 6.6 (we assume $s$ is non–empty as the case $s = \langle\rangle$ is trivial).

$$\begin{aligned}
& \quad (s^\frown r) \in \tau(\mathsf{S} \,\square\, \mathsf{T})\\
&\Leftrightarrow \quad (s^\frown r) \in \tau\mathsf{S} \vee (s^\frown r) \in \tau\mathsf{T} && \text{[Definition 6.3 and property of } \cup]
\end{aligned}$$

(case $s \in \tau\mathsf{S} \setminus \tau\mathsf{T}$)

$$\Leftrightarrow \quad (s|_u{}^\frown r) \in \tau\mathsf{S} \qquad [u \not\rightsquigarrow v : \mathsf{S}]$$

$$\Leftrightarrow \quad (s|_u{}^\frown r) \in \tau(\mathsf{S} \| \mathsf{T}); \qquad \text{[this case and (6.8.3)]}$$

(case $s \in \tau\mathsf{T} \setminus \tau\mathsf{S}$)

Similarly;

(case $s \in \tau\mathsf{S} \cap \tau\mathsf{T}$)

$$\begin{array}{lll} \Leftrightarrow & (s|_u{}^\frown r) \in \tau\mathsf{S} \wedge (s|_u{}^\frown r) \in \tau\mathsf{T} & [u \not\rightsquigarrow v : \mathsf{S};\ u \not\rightsquigarrow v : \mathsf{T}] \\ \Leftrightarrow & (s|_u{}^\frown r) \in \tau(\mathsf{S} \| \mathsf{T}). & \text{[Definition 6.3]} \end{array}$$ □

We illustrate the laws by reconsidering the encryption process of Example 6.3.

**Example 6.4** Using process $\mathsf{W}_t$ from Definition 6.7 we rewrite $\mathsf{S}$ (Example 6.3) once again:

$$\begin{array}{lll} \mathsf{S} & \triangleq & in?x \rightarrow \mathsf{W}_0 ; \mathtt{Wait}\ \ t ; (chan!f(x) \rightarrow \mathtt{Stop}\ _{\alpha\mathsf{S}}) \sqcap \mathsf{T} \\ & & \| \\ & & \mathsf{W}_\delta ; \mathtt{Wait}\ \ t ; \mathsf{T}. \end{array}$$

So we prove $in \not\rightsquigarrow chan : \mathsf{S}$ using laws, noting that

$$(chan!f(x) \rightarrow \mathtt{Stop}\ _{\alpha\mathsf{S}}) \sqcap \mathsf{T} = \mathsf{T}$$

as $f : M \rightarrow C$ and $\mathsf{T}$ (see Example 6.3) chooses an element of $C$ at random and

$$\begin{array}{lll} & in \not\rightsquigarrow chan : \mathtt{Stop}\ _{\alpha\mathsf{S}} & \text{[Lemma 6.2, part 3]} \\ \Rightarrow & in \not\rightsquigarrow chan : (chan!c \rightarrow \mathtt{Stop}\ _{\alpha\mathsf{S}}) & \text{[Law 6.2]} \\ \Rightarrow & in \not\rightsquigarrow chan : \mathsf{T} & \text{[Law 6.8]} \\ \Rightarrow & in \not\rightsquigarrow chan : (\mathtt{Wait}\ \ t ; \mathsf{T}) & \text{[Law 6.1]} \\ \Rightarrow & in \not\rightsquigarrow chan : (\mathsf{W}_0 ; \mathtt{Wait}\ \ t ; \mathsf{T}) & \text{[Laws 6.1 and 6.8]} \\ & \wedge & \\ & in \not\rightsquigarrow chan : (\mathsf{W}_\delta ; \mathtt{Wait}\ \ t ; \mathsf{T}) & \\ \Rightarrow & in \not\rightsquigarrow chan : \mathsf{S}. & \text{[Law 6.3]} \end{array}$$

Hence $in \not\rightsquigarrow chan : \mathsf{S}$ and the eavesdropper cannot tell whether a real or a bogus message has been sent, even looking at the time at which messages are sent. △

# Chapter 7

# Related Work

## 7.1 Introduction

This chapter contains a survey of work related to that presented in the preceding chapters. It is split into four sections describing: access control and databases (Section 7.2); information flow policies (Section 7.3); non–interference—including unwinding theorems—(Section 7.4); and refinement of security policies (Section 7.5).

We show how our work relates to the policies and results of others working in security and give lemmas relating their definitions to ours (where necessary), as well as examples in CSP.

## 7.2 Access Control and Databases

The technique of *access control* to enforce the security requirements of a system is widespread and was one of the first formally modelled forms of security. Bell and LaPadula (see [7, 8, 9, 10]) produced a mathematical model of the control of access by subjects (users of a system) to objects (system entities such as files, printers, memory, other users, etc.).

They model a system as a state–machine (it is an abstraction of the operation of the MULTICS) and it is considered secure if the system satisfies a collection of properties that are designed to guarantee that no reading or writing of data occurs unless allowed.

The theorem which Bell and LaPadula use to show security is a form of induction: if the initial state is secure (no disallowed reading or writing) and each state transition is secure then the system is secure; that theorem is known as the 'Basic Security Theorem'.

Because of the proof of security and the formal model, the Bell and LaPadula is widely used, but it has been criticized (see [66, 67]).

McLean shows that the constraints of the Bell and LaPadula model did not prevent a state transition which allowed every subject to read or write any object (that transition simply downgrades all objects to the lowest classification and allows every subject read and write access). That problem is countered (see [6]) by the additional restriction of *tranquillity*. A system is tranquil if the classification of each object may not change.

The addition of the tranquillity property does not solve all the criticisms of the model (see [67]) and suffers from the problem of not capturing the security policy originally specified.

Access control is intended to prevent any information flowing from high classification to low classification (i.e. to maintain confidentiality in a manner reflecting the operation of paper–based security—see [59]); it does that by specifying the access permitted to objects. Unfortunately access control does not capture communication along covert channels (see [58]). Although the access–control method can be modified to capture integrity (a system has integrity if there is no unauthorized alteration of data)—see [11]—availability (or denial of service—see [59]) is still a problem. As shown in Chapter 3 confidentiality, integrity and availability can all be considered as information–flow properties and specification (and development) of a system satisfying an information–flow policy captures covert communication too. In the case study (Chapter 5) the security policy is given formally (as a non–interference assertion) and, in the implementation, the restrictions 'no read–up' and 'no write–down' are used to enforce the policy.

When building a secure database system the access–control technique is commonly altered to allow polyinstantiation (see [62]). In a multi–level database (a database in which each row has a unique identifying value—the key—and the data stored in that row has an associated classification) data at various classifications is stored and may be searched by the users of the database. Each user has a clearance and, in the familiar way, information from high classification must not flow to users with a lower clearance. In order to enforce that restriction data in each row is polyinstantiated—multiple copies are stored for each classification for each key. The data a user sees for a particular key may be different from that seen by another; in that way communication is prevented from high to low.

Example 7.1 illustrates polyinstantiation in a simple system and its use in enforcing non–interference.

**Example 7.1** System $\mathsf{V}$ is a simple variable with two users $h$ (whose interface consists of the channels $h.rd$ and $h.wr$ on which $h$ may read and write the variable) and $l$ (using channels $l.rd$ and $l.wr$). The

security policy is 'no communication from $h$ to $l$' and captured formally by the non–interference assertion $h \not\rightsquigarrow l : \mathsf{V}$.

We suppose that the variable stores a single binary value and is initialized to 0. The variable represents a single row of a database.

$$
\begin{aligned}
\mathsf{V} &\triangleq \mathsf{V}_0^0 \\
\mathsf{V}_{lv}^{hv} &\triangleq (h.wr?hv' \rightarrow \mathsf{V}_{lv}^{hv'} \mid h.rd!hv \rightarrow \mathsf{V}_{lv}^{hv} \\
&\quad \mid l.wr?lv' \rightarrow \mathsf{V}_{lv'}^{lv'} \mid l.rd!lv \rightarrow \mathsf{V}_{lv}^{hv}).
\end{aligned}
$$

The system is non–interfering by Laws 4.1 and 4.14. To see why notice that the value read by $l$ is $lv$ and that the only way $lv$ changes is by $l$ writing it. $h$ cannot change $lv$ and cannot communicate with $l$.

When $l$ writes a value both $hv$ and $lv$ are changed and so both the values read by $h$ and $l$ will be identical; they remain identical until $h$ writes a different value. △

Whilst polyinstantiation is a useful technique, for example the SeaView project makes extensive use of it (see [62]), there is disagreement about whether it is an essential property of multi–level databases (see [52, 61, 71]). We believe that the correct way to *specify* the security requirements of any system (including databases) is by assertions about information flow. Thus the particular techniques used in the implementation need not be considered at the specification stage.

## 7.3 Information Flow

Information–flow policies concern the movement of information within a system without referring to specific techniques to prevent access to information. We examine three approaches to information flow: separability, inference functions, and deducibility.

Other approaches are given in Section 7.3.4 and the most common information–flow policy (non–interference) is treated in Section 7.4.

### 7.3.1 Separability

In [58] Lampson proposes the *confinement problem*: the problem of preventing a process (or user) from communicating with another in an unauthorized manner. Rushby, in [83], proposes a *separation kernel*, a security kernel that enforces separation of each of the subjects that interact with it. Once separated the subjects are confined in the sense of [58]. The term 'secure isolation' is also used to refer to the way in which subjects are confined.

Rushby requires that the output seen by a user depends only upon that user's input. He proves that a system (or machine) is separable by showing that a machine, local to each user, can be constructed that behaves no differently from the actual, shared implementation. In his M.Sc. thesis Burnham (see [17]) reworks Rushby's definition in terms of local refinement in CSP (local refinement is defined in [44]) and in [30] Burnham's definitions are slightly altered to remove the use of interleaving. Definitions 7.1 and 7.2 define separability through the use of local refinement.

**Definition 7.1** (*Burnham*) *A* user $\mathsf{U}_i$ *of system* $\mathsf{S}$ *is a CSP process. The alphabets of users* $\mathsf{U}_i$ *(for* $i$ *in some indexing set* $I$*) partition* $\alpha\mathsf{S}$*.* $\triangle$

**Definition 7.2** (*Burnham*) *A system* $\mathsf{S}$ *is* separable *if*

$$\mathsf{A} \triangleq \underset{i \in I}{\|}\ i : (\mathsf{U}_i \| \mathsf{S})$$

*is locally refined by*

$$\mathsf{B} \triangleq (\underset{i \in I}{\|}\ i : \mathsf{U}_i) \| I : \mathsf{S}$$

(*with the windows* $\{i : \alpha\mathsf{U}_i \mid i \in I\}$). $\triangle$

System $\mathsf{A}$ represents the behaviour of $\mathsf{S}$ with each user having exclusive access to an instance of $\mathsf{S}$ (each copy of $\mathsf{S}$ represents the local—or private—machine in Rushby's definition); $\mathsf{B}$ is the shared machine. The users' events are labelled with their index to distinguish events performed by different users.

$\mathsf{B}$ is a local refinement of $\mathsf{A}$, which means that the users are unable to distinguish between the shared machine and a (local) machine to which they have exclusive access.

Separability means that no user should be able to communicate with any other user; i.e. that no user interferes with another. We can capture that by the definition of pairwise non–interference.

**Definition 7.3** *If system* $\mathsf{S}$ *has users* $u_i$ (*in the sense of Definition* 3.1) *with indexing set* $i \in I$ *and*

$$\forall\, i, j \in I \bullet i \neq j \Rightarrow u_i \not\rightsquigarrow u_j : \mathsf{S},$$

*then the users of* $\mathsf{S}$ *are* pairwise non–interfering. $\triangle$

Pairwise non–interference is stronger than (Burnham's) separability.

**Lemma 7.1** *If* $\mathsf{S}$ *has pairwise non–interfering users* $u_i \triangleq \alpha\mathsf{U}_i$ $(i \in I)$ *then* $\mathsf{S}$ *is separable* (*Definition* 7.2)*.*

**Proof** Consider $s \in \tau\mathsf{B}$, we show that $s \in \tau\mathsf{A}$ and hence $\mathsf{B}$ (locally) refines $\mathsf{A}$ in traces. By Law 4.4 we obtain $(I : u_i) \not\rightsquigarrow (I : u_j) : (I : \mathsf{S})$.

$$
\begin{array}{ll}
 & s \in \tau\mathsf{B} \\
\Leftrightarrow & s \in \tau(\mathop{\|}\limits_{i \in I} i : \mathsf{U}_i) \wedge s \in \tau(I : \mathsf{S}) \qquad [\text{L1:72}] \\
\Leftrightarrow & \forall\, i \in I \bullet s\!\upharpoonright\!(i : u_i) \in \tau(i : \mathsf{U}_i) \wedge s \in \tau(I : \mathsf{S}) \qquad [\text{L1:72}] \\
\Rightarrow & \forall\, i \in I \bullet s\!\upharpoonright\!(i : u_i) \in \tau(i : \mathsf{U}_i) \wedge s|_{\bigcup_{j \neq i}} I : u_j \in \tau(I : \mathsf{S}) \\
 & \qquad [\text{Law 4.4}] \\
\Leftrightarrow & \forall\, i \in I \bullet s\!\upharpoonright\!(i : u_i) \in \tau(i : \mathsf{U}_i) \wedge s\!\upharpoonright\!(I : u_i) \in \tau(I : \mathsf{S}) \\
 & \qquad [u_i \text{ partition } \alpha\mathsf{S}] \\
\Leftrightarrow & \forall\, i \in I \bullet s\!\upharpoonright\!(i : u_i) \in \tau(i : \mathsf{U}_i) \wedge s\!\upharpoonright\!(i : \alpha\mathsf{S}) \in \tau(I : \mathsf{S}) \\
 & \qquad [i \in I;\ u_i \subseteq \alpha\mathsf{S}] \\
\Leftrightarrow & \forall\, i \in I \bullet s\!\upharpoonright\!(i : \alpha\mathsf{S}) \in \tau(i : \mathsf{U}_i \| i : \mathsf{S}) \qquad [\text{L1:72}] \\
\Leftrightarrow & \forall\, i \in I \bullet s\!\upharpoonright\!(i : \alpha\mathsf{S}) \in \tau i : (\mathsf{U}_i \| \mathsf{S}) \qquad [\text{L1:72}] \\
\Leftrightarrow & s \in \tau\mathsf{A}. \qquad [\text{L1:72}] \quad \square
\end{array}
$$

In [48] Jacob defines separability by saying that a system $\mathsf{S}$ is separable if it can be decomposed in to systems with disjoint alphabets that acting in parallel are *equal* to $\mathsf{S}$.

**Definition 7.4** (*Jacob*) *If* $\mathsf{S} = \mathop{\|}\limits_{i \in I} \mathsf{B}_i$ *for some* $\mathsf{B}_i$ *whose alphabets partition* $\alpha\mathsf{S}$ *then* $\mathsf{S}$ *is* separable*.* △

That definition is very strong: it says that no environment can distinguish between $\mathsf{S}$ and $\mathop{\|}\limits_{i \in I} \mathsf{B}_i$. That is, no possible user can distinguish those systems, not just the actual users. So, Jacob's definition is stronger than pairwise non–interference (and stronger than Burnham's definition of separability).

**Lemma 7.2** *If* $\mathsf{S}$ *is separable* (*Definition* 7.4) *then* $\mathsf{S}$ *is pairwise non–interfering for users* $u_i = \alpha\mathsf{B}_i$ $(i \in I)$*.*

**Proof** Consider some unequal $i$ and $j$, we require $u_i \not\rightsquigarrow u_j : \mathsf{S}$. By Lemma 4.22 we have $u_i \not\rightsquigarrow \emptyset : \mathsf{B}_i$, $\emptyset \not\rightsquigarrow u_j : \mathsf{B}_j$ and for all other $k$

(not equal to $i$ or $j$) $\emptyset \not\rightsquigarrow \emptyset : \mathsf{B}_k$. Apply Law 4.7 repeatedly to obtain $u_i \not\rightsquigarrow u_j : \mathsf{S}$. □

### 7.3.2 Inference Functions

In [44] (and [45, 46]) Jacob defines a function which determines the information a user of a shared system can deduce about the behaviour of another user. The deduction is made by observing interactions at a specified interface, and by comparing those interactions against the knowledge of the system's capabilities (the system's traces are used in [44]).

He begins by defining the projection of a system (a CSP process) that a user observes at some interface (a subset of the system's alphabet).

**Definition 7.5** (*Jacob*) *If system* $\mathsf{S}$ *has user* $u \subseteq \alpha\mathsf{S}$ *then*

$$\mathsf{S}@u \triangleq \{t \upharpoonright u \mid t \in \tau\mathsf{S}\}. \qquad \triangle$$

The inferences a user can make about the activity of a system are given by the inference function: the set of traces of the system consistent with an observation.

**Definition 7.6** (*Jacob*) *If system* $\mathsf{S}$ *has user* $u \subseteq \alpha\mathsf{S}$ *then the* inference function *of* $\mathsf{S}$ *for user* $u$ *observing interaction* $l \in \mathsf{S}@u$ *is*

$$\mathit{infer}_{\mathsf{S}}\ u\ l \triangleq \{t \mid t \in \tau\mathsf{S} \wedge t \upharpoonright u = l\}. \qquad \triangle$$

The principle differences between Jacob's approach and ours are that we do not assume that a user has full knowledge of the system's capabilities and that we have not attempted to define a measure of security.

The definition of an inference function is extended (in [44]) to provide the definition of a security specification. Such a specification defines which inferences are consistent with a given observation.

**Definition 7.7** (*Jacob*) *A* security specification *over* $A$ *is a function* $f : \mathbb{P}\ A \rightarrow A^* \rightarrow \mathbb{P}\ A^*$*, such that for all* $B, C \in \mathbf{dom}\ f$*,* $l \in \mathbf{dom}\ (f\ B)$

- ⋄ $\mathbf{dom}\ (f\ B) \subseteq B^*$ *is prefix–closed; and*
- ⋄ $\forall\, t \in (f\ B\ l) \bullet (t \upharpoonright B = l) \wedge (t \upharpoonright C \in \mathbf{dom}\ (f\ C))$. △

In that definition a security specification defines the valid inferences a user may make about the behaviour (the traces) of a system. Those inferences can be tested against the actual inferences given by *infer* to determine whether or not a system is secure.

Jacob uses the inference function to give a security ordering on systems: a system $\mathsf{S}$ is at least as secure as $\mathsf{T}$ if no more can be inferred of $\mathsf{S}$ than of $\mathsf{T}$, by any user. The ordering is used to give specifications of the degree of security required in a system.

The orderings of Definition 4.1 and 4.2 do not measure security, they show when one system has the same security properties as another. As Section A.2.3 shows, those orderings can be successfully applied to produce methods of refinement for secure systems (see also Section 7.5).

Jacob uses *infer* to define non–interference, see Section 7.4.2.

### 7.3.3 Deducibility

A security policy closely related to Jacob's approach (Section 7.3.2) is deducibility. Deducibility is defined (implicitly) in [94].

Sutherland argues as follows. $f_1$ and $f_2$ are 'information functions' from the set of possible worlds of a machine. The set of possible worlds corresponds to the set of 'execution sequences of an abstract machine,' or its set of traces. The information functions correspond to projections of the system onto users' interfaces. Sutherland says:

> Given a set of possible worlds $W$ and two functions $f_1$ and $f_2$ with domain $W$, we say that **information flows from** $f_1$ **to** $f_2$ if and only if there exists some possible world $w$ and some element $z$ in the range of $f_2$ such that $z$ is achieved by $f_2$ in some possible world but in every possible world $w'$ such that $f_1(w') = f_1(w)$, $f_1(w')$ is not equal to $z$. [94, page 176]

In CSP, with the information functions considered as projections onto interfaces, that definition becomes:

**Definition 7.8** *If* $\mathsf{S}$ *has users* $u$ *and* $v$ *(subsets of* $\alpha\mathsf{S}$*) then* information flows from $u$ to $v$ *iff there exists* $t \in \tau\mathsf{S}$, $l \in \mathsf{S}@v$ *and* $t' \in \tau\mathsf{S}$ *such that* $t' \upharpoonright v = l$ *and for all* $s \in \tau\mathsf{S}$ $s \upharpoonright u = t \upharpoonright u \Rightarrow s \upharpoonright v \neq l$. △

Like the inference function of Jacob, Sutherland's definition relies on the users knowing the set of possible worlds (traces) of a system.
Sutherland shows that if no information flows between users $u$ and $v$ in a system then $u$ is non–interfering with $v$. He also gives an example

that shows that the converse is not true: deducibility security is strictly stronger than non–interference (cf. Lemma 7.4 and Example 7.2).

### 7.3.4 Other Information–Flow Approaches

#### McLean's Flow Model

One of the results of [94] is that if information flows from $f_1$ to $f_2$ then it also flows from $f_2$ to $f_1$; that is one of the reasons that deducibility is such a strong definition.

In [68] McLean proposes a definition of lack of information flow based around Sutherland's ideas that is not symmetric. His definition is given in terms of the probability of particular assignments of the values of all system objects. He defines the *Flow Model*:

**Definition 7.9** (*McLean*) *If system* $\mathsf{S}$ *has a high objects space* $H$ *and a low objects space* $L$ *then* $\mathsf{S}$ *is* secure *iff*

$$p(L_t \mid H_s \wedge L_s) = p(L_t \mid L_s).$$

*Where* $H_s$ *and* $L_s$ *are the sequences of values of high and low level objects in every state that precedes* $t$ *and* $p$ *is a probability function.* △

The definition says that the value of low–level objects is independent of the value of high–level objects; the state of low–level objects is determined only by the previous states of those objects.

McLean uses Definition 7.9 to analyse non–interference and restrictiveness (see Section 7.4.3) and to argue that the Bell and LaPadula model is not sufficient to guarantee non–interference. Whilst the Flow Model may be useful in analysing the definitions of different security policies it does not seem to be easily applicable to the construction of secure systems.

#### O'Halloran's Information–Flow Category

In [77] an expression of Jacob's inference function is given in a categorical framework. The information–flow category has objects representing pieces of information and arrows representing the relationship between pieces of information; an arrow from knowledge $k$ to knowledge $l$ means that knowing $k$ is consistent with knowing $l$ (cf. the function $infer_{\mathsf{S}}$ which defines the set of observations—the knowledge—consistent with an observation).

An equivocation proposition is defined as an endofunctor, over the information–flow category, for which a natural transformation to the

category exists; an equivocation proposition is a specification of consistent knowledge and is an abstraction of Definition 7.6.

O'Halloran investigates the product in the category and gives a concrete instantiation of the category: the traces of a CSP process. There the product corresponds to parallel composition. He is able to prove (Theorem 3 in [77]) that the product of systems satisfies the conjunction of their equivocation propositions. In the concrete category this means that the parallel composition of systems satisfies the conjunction of their equivocation propositions; cf. Law B.4.

See Section 7.4.4 where O'Halloran's definition of *non–inference* is related to non–interference.

### Foley's Universal Theory of Information flow

Foley, in [25], captures information flow by saying that information flows from one user to another if the first user can restrict the behaviour of the second.

He defines a user as a CSP process whose alphabet is a subset of the alphabet of the system with which the user interacts; users are restricted so that they always agree to engage in events which they are allowed to *observe* and can choose whether to engage in events they *control*.

**Definition 7.10** *A user* $\mathsf{U}$ *of system* $\mathsf{S}$ *is* valid *if* $\alpha\mathsf{U} \subseteq \alpha\mathsf{S}$, $\tau\mathsf{U} \subseteq \mathsf{S}@\alpha\mathsf{U}$ *and* $ValidUser(\mathsf{S}, \mathsf{U})$ (*ValidUser is defined syntactically in* [25]). $\triangle$

Information flows from interface $B$ to interface $C$ (subsets of the alphabet of the system) if there is a user with alphabet $B$ that can restrict the behaviour at interface $C$.

**Definition 7.11** *If system* $\mathsf{S}$ *has interfaces* $B \subseteq \alpha\mathsf{S}$ *and* $C \subseteq \alpha\mathsf{S}$ *then information flows from* $B$ *to* $C$ *iff*

$$\exists\, t \in \mathsf{S}@C\; \exists\, \mathsf{U} \bullet (\alpha\mathsf{U} = B \wedge \mathsf{U} \text{ is valid}) \Rightarrow t \notin (\mathsf{S}\|\mathsf{U})@C. \qquad \triangle$$

In [25] various examples are proved secure (i.e. no information flows between specified users) using the traces semantics of CSP.

## 7.4 Non–interference

The original expression of non–interference given in [28] is examined in Section 7.4.1, along with the associated conditional non–interference

assertions. A number of authors have expressed non–interference using CSP, their definitions are related to ours in Section 7.4.2.

McCullough's definition of generalized non–interference (see the paper [63]) lead to the security property called restrictiveness, and others: they are considered in Section 7.4.3.

Non–interference is widely used and other non–interference definitions and unwinding theorems are discussed in Section 7.4.4.

### 7.4.1 Goguen and Meseguer's Non–interference

Goguen and Meseguer's papers on non–interference, [28, 29], were one of the first expressions of an information–flow policy. They motivate the definition of non–interference by saying:

> one group of users, using a certain set of commands, is **noninterfering** with another group if what the first group does with those commands has no effect on what the second group of users can see. [28, page 11]

The way in which the phrases 'what the first group does' and 'what the second group of users can see' is of prime importance in giving a non–interference policy. Rushby, in [84], interprets the first phrase by using the familiar purge function to remove the first user's commands; then the presence of absence of operations (with any input) must not affect the (second group of) users' view of the system. (The purge is more complex in Goguen and Meseguer's original definition because of their definition of non–interference between groups of users with certain commands.)

Both Goguen and Meseguer and Rushby define non–interference in terms of a deterministic finite state–machine model and the view of the system is given by the set of outputs a user can generate from any given state.

Goguen and Meseguer's treatment is given here; Rushby's is very similar.

**Definition 7.12** (*Goguen and Meseguer*) *A* state machine *$M$ consists of*

- *A set $U$ of users; a set $S$ of states; a set $C$ of state commands; a set $Out$ of outputs;*
- *A function $out : S \times U \to Out$ (the* output function*) which gives the output a user sees in a state;*

- ⋄ *A function* $do : S \times U \times C \rightarrow S$ (*the* state transition function) *which shows how the state is updated when a given user performs a command.*

*The state the machine is in after performing a sequence of commands* $t \in (U \times C)^*$ *is denoted* $[\![t]\!]$ *and the possible output a user* $u$ *may see after performing the sequence* $t$ *is* $[\![t]\!]_u \triangleq out([\![t]\!], u)$. △

They define non–interference ([28, page 16]) for groups of users and sets of commands, we simplify that definition to a pair of users and any command.

One user does not interfere with another if the output seen by the second user is unaffected by the first user's operations.

**Definition 7.13** (*Goguen and Meseguer*) *If machine* $M$ *has users* $u$ *and* $v$ *then* $u$ *is* non–interfering *with* $v$ (*written* $u :| \, v$) *iff*

$$\forall\, t \in (U \times C)^* \bullet [\![t]\!]_v = [\![t|_u]\!]_v . \qquad \triangle$$

The purge function in Definition 7.13 removes commands from $\{u\} \times C$ in the manner that the purge of Definition 3.8 operates.

It is easy to see the connection between Definition 7.13 and the definitions in Section 3.3: the output after trace $t$ for user $u$ is replaced by the equivalence class $[t]^{\mathsf{S}}_u$. We are able to generate that class because we have not distinguished between input and output (both are bound–up in the definition of an event) and because we do not, unlike Goguen and Meseguer, assume that all traces are possible.

In [29] they extend the definition by giving three weakened forms of non–interference, which they call conditional non–interference assertions (one was defined in [28] but is refined in [29]). They are easily expressed in our notation.

**Definition 7.14** (*Goguen and Meseguer*) *If machine* $M$ *has users* $u$ *and* $v$ *and* $P$ *is a predicate on* $(U \times C)^*$ *then* $u$ *is* non–interfering *with* $v$ *under condition* $P$ (*written* $u :| \, v$ **if** $P$) *iff*

$$\forall\, t \in (U \times C)^* \bullet [\![t]\!]_v = [\![p(t)]\!]_v$$

*where* $p : (U \times C)^* \rightarrow (U \times C)^*$ *is the strict function such that*

$$p(t \frown \langle (ur, co) \rangle) = (p(t) \mathrm{\ C\ } (ur = u) \wedge P(t) \mathrm{\ B\ } p(t) \frown \langle (ur, co) \rangle).$$

*In our notation that is written* $\forall\, t \in (U \times C)^* \bullet t \approx^M_v p(t)$. △

The predicate $P$ governs how commands performed by $u$ are purged from a trace. Goguen and Meseguer show that **if** can be used to define discretionary access control by removing $u$'s events if a command (causing $u$'s events to be hidden) has been performed (see [28, Example 7] which defines a form of discretionary access control).

**Definition 7.15** (*Goguen and Meseguer*) *If machine $M$ has users $u$ and $v$, and $Q$ is a predicate on $(U \times C)^*$ then $u$ is* non–interfering *with $v$ unless $Q$ (written $u$ :| $v$* **unless** *$Q$) iff*

$$\forall\, t \in (U \times C)^* \bullet [\![t]\!]_v = [\![t']\!]_v$$

*where $t = t_0{}^\frown t_1$ and $t' = t_0{}^\frown(t_1|_u)$ where $t_0$ is the longest initial sub–sequence of $t$ such that $Q(t_0)$.*

*In our notation that is: $\forall\, t \in (U \times C)^* \bullet (t \approx_v^M t')$.* △

The **unless** assertion is used in [29] to define a channel between two users by allowing $u$ to communicate with $v$ using some subset of $u$'s events. $Q$ is instantiated for some $A \subseteq u$ with meaning 'the current operation is in $A$.' Then $u$ :| $v$ **unless** $Q$ means that 'no input from $u$ after his last operation in $A$ can affect $v$, so that $v$ can see effects of $u$ commands only after $u$ executes an operation in $A$' (see [29, Example 3]).

**Definition 7.16** (*Goguen and Meseguer*) *If machine $M$ has users $u$ and $v$ and $P \subseteq (U \times C)^*$ then $u$ is* non–interfering *with $v$ given $P$ (written $u$ :| $v$* **given** *$P$) iff*

$$\forall\, t \in P \bullet [\![t]\!]_v = [\![p(t)]\!]_v.$$

*In our notation: $\forall\, t \in P \bullet (t \approx_v^M t|_u)$.* △

They do not state the purpose of, or investigate, **given**.

Goguen and Meseguer give an Unwinding Theorem in [29] which shows that an assertion of non–interference (they used non–interference to express MLS—as we did in Chapter 5) can be proved for the individual transitions (commands), from which non–interference can be deduced by induction on sequences of commands. We gave such theorem (Theorem 4.1) as have other authors (see Section 7.4.4).

### 7.4.2 CSP Formulations of Non–interference

A number of authors have defined non–interference using the CSP notation. In this section we examine their results.

**Non–interference expressed using Initials**

In [86] (and the earlier draft [85]) non–interference is defined using the traces model of CSP (the ready–set model is also used, but we consider the traces definition here). In [1] the presentation of non–interference in Ryan's papers is improved slightly and is given in Definition 7.17.

**Definition 7.17** (*Allen*) *If system* $\mathsf{S}$ *has users* $A$ *and* $B$ ($A$ *and* $B$ *partition* $\alpha\mathsf{S}$) *then* $A$ *is* non–interfering *with* $B$ *through* $\mathsf{S}$ (*which is written* $A \rightarrow \|_{\mathsf{S}} B$) *iff*

$$\forall\, t \in \tau\mathsf{S} \bullet t \upharpoonright B \in \tau\mathsf{S} \wedge \iota(\mathsf{S}/t) \cap B = \iota(\mathsf{S}/t \upharpoonright B) \cap B. \qquad \triangle$$

Lemma 7.3 shows that $A \rightarrow \|_{\mathsf{S}} B$ is equivalent to $A \not\rightsquigarrow B : \mathsf{S}$ when only considering traces (we use the traces definition of $\approx_B^{\mathsf{S}}$ from Definition 3.4).

**Lemma 7.3** *If* $\mathsf{S}$ *has users* $A$ *and* $B$ *that partition* $\alpha\mathsf{S}$ *then, in traces,*

$$A \not\rightsquigarrow B : \mathsf{S} \Leftrightarrow A \rightarrow \|_{\mathsf{S}} B.$$

**Proof** It is clear from Definition 3.4 that $A \not\rightsquigarrow B : \mathsf{S} \Rightarrow A \rightarrow \|_{\mathsf{S}} B$. We show the converse, that $\forall\, r \in B^* \bullet (t ^\frown r \in \tau\mathsf{S} \Leftrightarrow t|_A {}^\frown r \in \tau\mathsf{S})$ for any $t \in \tau\mathsf{S}$, by induction on $r$.

**(base case)** $t$ is a trace of $\mathsf{S}$ and, as the users partition the alphabet, $t \upharpoonright B = t|_A$ is also a trace. Hence the case $r = \langle\rangle$ holds.

**(inductive step)** The hypothesis is that for all $r \in B^*$ such that $\#r \leq n$

$$t ^\frown r \in \tau\mathsf{S} \Leftrightarrow t|_A {}^\frown r \in \tau\mathsf{S}.$$

If $b \in B$ then

$$\begin{array}{lll}
 & (t ^\frown r ^\frown \langle b \rangle) \in \tau\mathsf{S} & \\
\Leftrightarrow & b \in \iota(\mathsf{S}/t ^\frown r) \cap B & [\text{Definition 2.15}] \\
\Leftrightarrow & b \in \iota(\mathsf{S}/t ^\frown r \upharpoonright B) \cap B & [\text{Definition 7.17}] \\
\Leftrightarrow & b \in \iota(\mathsf{S}/t \upharpoonright B ^\frown r) \cap B & [r \in B^*] \\
\Leftrightarrow & b \in \iota(\mathsf{S}/t|_A {}^\frown r) \cap B & [t \upharpoonright B = t|_A] \\
\Leftrightarrow & (t|_A {}^\frown r ^\frown \langle b \rangle) \in \tau\mathsf{S}. & [\text{Definition 2.15}] \quad \square
\end{array}$$

**Non–interference in terms of inference functions**

In [51] Jacob gives a definition of non–interference in terms of inference functions. His definition says that a user $u$ is non–interfering with user

$v$ if for any trace $v$ can infer that $u$ has either performed some sequence of events, or none at all.

**Definition 7.18** (*Jacob*) *If system* $\mathsf{S}$ *has users* $u$ *and* $v$ (*disjoint subsets of* $\alpha\mathsf{S}$) *then* $u$ *is* non–interfering *with* $v$ *iff*

$$\forall\, l \in \mathsf{S}@v \bullet \langle\rangle \in (\mathit{infer}_{\mathsf{S}}\ v\ l)@u. \qquad \triangle$$

His definition of non–interference is weaker than ours, consider Lemma 7.4 and Example 7.2.

**Lemma 7.4** *If system* $\mathsf{S}$ *has users* $u$ *and* $v$ *and* $u \not\rightsquigarrow v : \mathsf{S}$ *then* $u$ *is non–interfering with* $v$ (*Definition* 7.18).

**Proof** Take any $l \in \mathsf{S}@v$ then there exists $t \in \tau\mathsf{S}$ such that $t \upharpoonright v = l$ (Definition 7.5) and since $u \not\rightsquigarrow v : \mathsf{S}$ we have $t|_u \in \tau\mathsf{S}$. $t|_u \upharpoonright v = l$ (since $u$ and $v$ are disjoint) and $t|_u \upharpoonright u = \langle\rangle$, therefore $\langle\rangle \in (\mathit{infer}_{\mathsf{S}}\ v\ l)@u$ and $u$ is non–interfering with $v$ by Definition 7.18. □

**Example 7.2** The system

$$\mathsf{S} \triangleq \mu\,\mathsf{X} \bullet (ve \rightarrow \mathsf{X}) \| ue \rightarrow \mathrm{Stop}\ _{\{ue,ve\}},$$

with users $u \triangleq \{ue\}$ and $v \triangleq \{ve\}$, is non–interfering by Definition 7.18, but it is not true that $u \not\rightsquigarrow v : \mathsf{S}$. To see that consider the trace $\langle ue\rangle$, it is not $v$–equivalent to $\langle ue\rangle|_u = \langle\rangle$, as the system stops after engaging in $ue$. △

**Foley's Strong Noninterference**

In [26] Foley defines *strong noninterference* in CSP:

**Definition 7.19** (*Foley*) *If system* $\mathsf{S}$ *has user* $v$ (*a subset of* $\alpha\mathsf{S}$) *then* $v$ *cannot be* strongly interfered *with iff* $\mathsf{S}@v \subseteq \tau\mathsf{S}$. △

He uses that definition to give a small collection of laws for strong noninterference: he shows that prefixing a strongly noninterfering process by an event from $u$ does not introduce interference (cf. Law 4.1); that strong noninterference is preserved by parallel composition (cf. Law 4.6); gives a condition for strong noninterference to be preserved under choice composition (cf. Law 4.11) and that if $\mathsf{F}(\mathsf{X})$ is strongly noninterfering then so is the recursive process $\mu\,\mathsf{X} \bullet \mathsf{F}(\mathsf{X})$ (cf. Law 4.12).

Definition 7.19 says that for any trace of $\mathsf{S}$ the trace restricted to the interface of $v$ is a trace of $\mathsf{S}$; in a two–user system (with the other user named $u$; $u$ and $v$ partition $\mathsf{S}$'s alphabet) that is equivalent to the trace with $u$'s events purged being in $\tau\mathsf{S}$.

In traces, our definition of non–interference is stronger than Foley's.

**Lemma 7.5** *If* $\mathsf{S}$ *has users* $u$ *and* $v$ *that partition* $\alpha\mathsf{S}$ *and* $u \not\rightsquigarrow v : \mathsf{S}$ *then* $v$ *cannot be strongly interfered with* (*Definition* 7.19).

**Proof**

$$\begin{aligned} u \not\rightsquigarrow v : \mathsf{S} \;&\Leftrightarrow\; \forall\, t \in \tau\mathsf{S} \bullet t \approx^{\mathsf{S}}_{v} t|_{u} && \text{[definition 3.9]}\\ &\Rightarrow\; \forall\, t \in \tau\mathsf{S} \bullet t|_{u} \in \tau\mathsf{S} && \text{[Definition 3.4]}\\ &\Leftrightarrow\; \mathsf{S}@v \subseteq \tau\mathsf{S}. && \text{[$u$ and $v$ partition the alphabet]} \end{aligned}$$ □

With more than two users the relationship exhibited by Lemma 7.5 does not hold; Example 7.3 shows why.

**Example 7.3** The system $\mathsf{S}$ with users $u \triangleq \{ue\}$ and $v \triangleq \{ve\}$:

$$\mathsf{S} \;\triangleq\; (ue \rightarrow e \rightarrow ve \rightarrow \mathtt{Stop}_{\;\{e,ue,ve\}}) [\!] (e \rightarrow ve \rightarrow \mathtt{Stop}_{\;\{e,ue,ve\}}).$$

That system satisfies the policy $u \not\rightsquigarrow v : \mathsf{S}$ (by Lemma 4.22, Law 4.1 and Law 4.2), but is not strongly noninterfering: $\langle ve \rangle$ is in $\mathsf{S}@v$ but not in the traces of $\mathsf{S}$. This is because strong noninterference (Definition 7.19) does not specify the user which is interfering with $v$, any subset of $(\alpha\mathsf{S} \setminus v)$ is a candidate. △

### 7.4.3 Restrictiveness and Correctability

McCullough, in [64], was concerned that any security policy should be composable and defined a composable security policy which he calls restrictiveness.

Restrictiveness is a close relation of non–interference; it is one of many security properties defined in terms of the traces of a system.

McCullough only considers non–interference assertions between two users named *high* and *low*. The usual military restriction of no communication from *high* to *low* applies and is captured by $high \not\rightsquigarrow low$.

McCullough introduces a property which he calls *generalized non–interference*. It is a version of non–interference for non–deterministic systems given purely in terms of its traces; if after any trace the possible extensions of that trace restricted to *low*'s outputs is the same as after the trace followed by an input by *high* then *high* does not interfere with *low*. In traces:

**Definition 7.20** (*McCullough*) *If system* $\mathsf{S}$ *has users high and low* (*sets of events*) *that partition* $\alpha\mathsf{S}$ *then high is* generalized non–interfering *with low iff*

$$\forall\, t \in \tau\mathsf{S}\,\forall\, h \in high \bullet (\mathsf{S}/t^\frown\langle h\rangle)@low = (\mathsf{S}/t)@low. \qquad \triangle$$

The difference between McCullough's definition and ours is that we stop events occurring after a trace $t$ apart from those from the interface of *low*; McCullough allows those events to occur and just considers the projection of the system onto *low*.

Our definition is stronger than McCullough's:

**Lemma 7.6** *If system* $\mathsf{S}$ *has users high and low that partition* $\alpha\mathsf{S}$ *and* $high \not\rightsquigarrow low : \mathsf{S}$ *then high is generalized non–interfering with low.*

**Proof**

$$\begin{array}{lll} & s \in (\mathsf{S}/t^\frown\langle h\rangle)@low & \\ \Leftrightarrow & \exists\, r \in \alpha\mathsf{S}^* \bullet (t^\frown\langle h\rangle^\frown r) \in \tau\mathsf{S} \wedge (r \upharpoonright low = s) & \text{[Definition 7.5]} \\ \Leftrightarrow & (t^\frown\langle h\rangle^\frown r)|_{high} \in \tau\mathsf{S} & [high \not\rightsquigarrow low : \mathsf{S} \text{ and Definition 3.4}] \\ \Leftrightarrow & t|_{high}{}^\frown s \in \tau\mathsf{S} & [r \upharpoonright low = s;\ h \in high] \\ \Leftrightarrow & t^\frown s \in \tau\mathsf{S} & [high \not\rightsquigarrow low : \mathsf{S} \text{ and } s \in low^*] \\ \Rightarrow & s \in (\mathsf{S}/t)@low. & \text{[Definition 7.5]} \end{array}$$

Similarly for the reverse direction. □

Restrictiveness is called *hook–up security* in [63] and is defined:

**Definition 7.21** (*McCullough*) *If system* $\mathsf{S}$ *has users high and low that partition* $\alpha\mathsf{S}$ *then* $\mathsf{S}$ *is* restrictive *iff* $\forall\, s \in \tau\mathsf{S}\,\forall\, s' \in \alpha\mathsf{S}^*\,\exists\, t \in \tau\mathsf{S}$ *such that*

$$\begin{array}{ll} & (s \upharpoonright low = s' \upharpoonright low) \wedge (s = s_0{}^\frown s_1) \wedge (s' = s_0{}^\frown s_1') \\ \Rightarrow & (s \upharpoonright low = t \upharpoonright low) \wedge (s' \upharpoonright high = t \upharpoonright high) \wedge (t = s_0{}^\frown t') \end{array}$$

*for some* $s_0, s_1, s_1'$ *and* $t'$ *in* $\alpha\mathsf{S}^*$. △

The definition in [63] also includes a restriction on when the first output to *high* can occur, we have ignored it as input and output are not considered different in CSP.

When a system is restrictive any change in the high–level events does not affect the low–level events in any trace: there are traces with the same sequence of low–level events for any sequence of high–level events.

McCullough's definition is weaker than ours; the proof of Lemma 7.7 is similar to the proof of Lemma 7.6.

**Lemma 7.7** *If system* $\mathsf{S}$ *has users high and low that partition* $\alpha\mathsf{S}$ *and* $high \not\rightsquigarrow low : \mathsf{S}$ *then* $\mathsf{S}$ *is restrictive.* □

Other variants on generalized non–interference and restrictiveness have been proposed by a number of authors, all of them rely on inserting and deleting events from traces. A summary of these possible perturbations of traces is given in [70]. Two such properties are forward correctability in [53] and ND in [35].

All the definitions based on perturbations have in common the distinction between input and output. The relative placing of inputs and outputs becomes very important in those definitions and much of the technical complexity comes from restrictions on the order of input and output. We have avoided such issues by considering synchronous systems with input and output considered as part of each event. If a distinction between input and output is desired we could partition each set of events into 'inputs' and 'outputs' without disturbing our definition of security; we would, however, have to be careful not to then define security policies with specific references to inputs and outputs as if they were different interfaces to the system.

## 7.4.4 Other Non–interference Results

### Non–interference in a Labelled Transition System

In [54] Johnson and Thayer define non–interference by way of the testing semantics of processes of [36]. They assign each action of a labelled transition system (LTS) a security level and define the set of actions at or below each level. Using those sets they define a *probe*. A probe is a LTS for some security level which represents the possible actions and states of users at or below that security level. Definition 7.22 summarizes those definitions.

**Definition 7.22** (*Johnson and Thayer*) *If* $\langle P, Act, \rightarrow\rangle$ *is an LTS and* $\mathcal{L}$ *is the partially ordered* $(\preceq)$ *set of* security levels *then there is a function* $l : Act \rightarrow \mathcal{L}$ *which assigns levels to actions.*

*The set* $LAct_l$ *is the set of actions whose security level is less than or equal to* $l$ *and* $LAction_l \triangleq LAct_l \cup \{i\}$ ($i$ *is the internal action*). *The sets* $HAct_l$ *and* $HAction_l$ *are defined dually.*

*A* probe *at level* $l$ *is an LTS* $\mathcal{F} \triangleq \langle F, LAction_l, \rightarrow\rangle$. △

Then they define when two processes are equivalent at a security level $l$. To do this they use probes to test whether the processes are different;

if probes at level $l$ cannot distinguish them then they are equivalent. That is analogous to our algebraic definition of equivalence (Definition 3.7).

**Definition 7.23** (*Johnson and Thayer*) *The processes $p$ and $q$ are $l$–*equivalent (*written $p \approx_l q$*) *iff $p$ and $q$ are indistinguishable by probes at level $\preceq l$.* △

They say that a system is secure when low levels cannot distinguish between processes before or after high level actions have occurred, and when processes that are equivalent evolve equivalently. These are exactly the proof obligations of Theorem 4.1, the unwinding theorem for non–interference, expressed in the notation of algebraic processes.

**Definition 7.24** (*Johnson and Thayer*) *An LTS $\langle P, Act, \rightarrow\rangle$ is* secure *at level $l$ iff*

$$\forall\, h \in HAct_l \bullet (p \xrightarrow{h} p') \Rightarrow (p \approx_l p')$$
$$\forall\, a \in LAct_l \bullet (p \approx_l q) \wedge (p \xrightarrow{h} p') \wedge (q \xrightarrow{h} q') \Rightarrow (p' \approx_l q').$$
△

They take that as the definition of security and show that the conditions imply a form of non–interference. (The symbol $\stackrel{s}{\Rightarrow}\!\bullet$ indicates the occurrence of the sequence of actions $s$ performed in order.)

**Theorem 7.1** (*Johnson and Thayer*) *If $\langle P, Act, \rightarrow\rangle$ is a secure LTS and $p \stackrel{s}{\Rightarrow}\!\bullet q$, then there exists $q'$ such that $p \stackrel{s'}{\Rightarrow}\!\bullet q'$, where $s' = HLPurge(s)$ and $q \approx_l q'$ (HLPurge removes high–level events from a sequence of actions).* □

### Extending the Purge Function

In [55] Johnson and Thayer extend the Goguen and Meseguer definition of non–interference by allowing purge functions other than the standard one (which removes all elements from some set from a trace). They do that by defining a *tolerable set*, a set of input events whose visibility (effect on other users of the system) is tolerated (an input event is invisible if it does not affect the other users' view of the system).

Then a purge function is a function which maps each sequence of input events onto some element of the tolerable set, i.e. the tolerable set defines the purge function. Johnson and Thayer show how the conditional non–interference assertions of [29] fit into this framework and give small examples of their use.

Tolerable sets are not as general as the definition of security given in Appendix B as restrictions are placed upon them in [55]. The paper does show, however, by way of examples that non–interference is applicable to security policies more subtle than total lack of communication by simple alterations to the definition of purge (cf. Goguen and Meseguer's conditional non–interference assertions in Section 7.4.1).

### Non–inference in CSP

O'Halloran gives a definition of a property closely related to non–interference in [77] which he calls *non–inference*. It is weaker than our definition of non–interference (the proof is a trivial consequence of Definition 3.9 and Definition 3.4 which shows that $t|_u \in \tau\mathsf{S}$); the definitions are not equivalent: Example 7.4 shows why.

**Definition 7.25** (*O'Halloran*) *If system* $\mathsf{S}$ *has users* $u$ *and* $v$ (*disjoint subsets of* $\alpha\mathsf{S}$) *then* $v$ *is* non–inferring *of* $u$ *iff*

$$\forall\, t \in \tau\mathsf{S} \bullet (t|_u \in \tau\mathsf{S}) \vee (t \upharpoonright v = \langle\rangle). \qquad \triangle$$

**Example 7.4** Consider system $\mathsf{S}$ of Example 7.2, which we showed is not non–interfering. $v$ is non–inferring of $u$ (Definition 7.25) since any trace $t$ is of the form $\langle ve\rangle^n$ or $\langle ve\rangle^n{}^\frown\langle ue\rangle$ (any $n \geq 0$) and hence $t|_u \in \tau\mathsf{S}$. △

### Unwinding Theorems

As well as Goguen and Meseguer's Unwinding Theorem, Rushby showed in [84] that non–interference could be unwound with two simple conditions. Theorem 7.2 is a statement of Rushby's theorem.

**Theorem 7.2** (*Rushby*) *If machine* $M$ *has users* $u$ *and* $v$ (*in* $U$) *and* $\forall\, s, t \in (U \times C)^*$

$$\forall\, c \in C \bullet [\![t^\frown\langle(u, c)\rangle]\!]_v = [\![t]\!]_v \qquad (5)$$

$$\forall\, op \in (U \times C) \bullet [\![s]\!]_w = [\![t]\!]_w \Rightarrow [\![s^\frown\langle op\rangle]\!]_w = [\![t^\frown\langle op\rangle]\!]_w. \qquad (6)$$

*then* $u$ *is non–interfering with* $v$ (*in the sense of Definition* 7.13). □

The two obligations in that theorem are very similar to those of our unwinding theorem (Theorem 4.1), the difference is in the definition of the view a user has of the system: Rushby uses the outputs a user can generate, we use the equivalence class of a trace under $\approx_v^{\mathsf{S}}$.

Note that the obligations in Theorem 7.2 are used as the equations defining the security property in [54] (cf. Definition 7.24).

## 7.5 Refinement of Security Policies

The main work on refinement of security properties is contained in Jacob's thesis [44] and subsequent papers [47]. In [49] Jacob considers *integrity* and its relationship with CSP refinement, the result of that paper is given below.

We work here from the definition given in [44]. Recall from Section 7.3.2 that Jacob defines the set of inferences a user can make about the systems behaviour by the function $infer_{\mathsf{S}}$ (see Definition 7.6). He then defines when one system is at least as secure as another. System $\mathsf{R}$ is at least as secure as system $\mathsf{S}$ through a set of windows $\mathcal{A}$ (set of subsets of $\alpha\mathsf{S}$) if the observations (i.e. the inferences) each member of $\mathcal{A}$ (a user) can make about about $\mathsf{S}$ are included in the observations that user can make about $\mathsf{R}$.

**Definition 7.26** (*Jacob*) *If* $\mathcal{A} \subseteq \mathbb{P}\ A$ *and systems* $\mathsf{R}$ *and* $\mathsf{S}$ *have alphabet* $A$ *then* $\mathsf{R}$ *is* at least as secure *as* $\mathsf{S}$ *through* $\mathcal{A}$ *iff*

$$\forall B \in \mathcal{A} \forall l \in \mathsf{R}@B \bullet infer_{\mathsf{S}}\ B\ l \subseteq infer_{\mathsf{R}}\ B\ l.$$

*That is written* $\mathsf{R} \succeq^{\mathcal{A}} \mathsf{S}$. △

(That definition is a contraction of Definitions 30 and 31 in [44].)

He then extends the definition to an ordering on security specifications (see Definition 7.7) by noting that $infer_{\mathsf{S}}$ is a security specification and:

**Definition 7.27** (*Jacob*) *If* $f$ *and* $g$ *are security specifications over* $A$ *then* $f$ *is* at least as secure *as* $g$ *iff for all* $B \in \mathbf{dom}\ f \cap \mathbf{dom}\ g$ *and* $l \in \mathbf{dom}\ (f\ B)$*:*

$$\mathbf{dom}\ f \supseteq \mathbf{dom}\ g\,; \quad \mathbf{dom}\ (f\ B) \subseteq \mathbf{dom}\ (g\ B) \ \ \textit{and} \ \ (f\ B\ l) \supseteq (g\ B\ l).$$

*That is written* $f \succeq g$. △

He shows, in [44, Lemma 47], that security specifications form a complete lattice and defines satisfaction of a specification by:

**Definition 7.28** (*Jacob*) *If* $f$ *is a security specification over* $\alpha\mathsf{S}$ *then* $\mathsf{S}$ satisfies $f$ (*written* $\mathsf{S} \succeq f$) *iff* $infer_{\mathsf{S}} \succeq f$. △

The main difference between Jacob's definitions and ours is that Jacob claims that $\succeq$ defines an order of the form 'is more secure than.' We have defined a security ordering which preserves a particular property, and have not tried to make it a measure of security.

In [49] (and also [43]) he focuses his attention on integrity and defines the expected behaviour of a system at a user's interface by way of an *expectation*. An expectation is a specification of the expected behaviour of a system at an interface; i.e. a process with the expected behaviour.

He then defines the exceptions of a process to be the observations a user can make that are unexpected. (We give the definition here using traces; in [49] Jacob is more general: he does not specify the type of observation that is being made.)

**Definition 7.29** (*Jacob*) *If system* $\mathsf{S}$ *has expectation* $\mathsf{E}$ *then the* exceptions *of* $\mathsf{S}$ *are*

$$\mathit{exception}\ \mathsf{E}\ \mathsf{S} \triangleq \{t \in \tau\mathsf{S} \mid t \upharpoonright \alpha\mathsf{E} \in \tau\mathsf{E}\}$$

*whenever* $\alpha\mathsf{E} \subseteq \alpha\mathsf{S}$. $\triangle$

He uses Definition 7.29 to define when one processes has no less integrity than another. That is expressed here without the use of Z (Z is used in the original paper—[49]—as well as CSP).

**Definition 7.30** (*Jacob*) *If processes* $\mathsf{S}$ *and* $\mathsf{R}$ *have expectation* $\mathsf{E}$ *then* $\mathsf{S}$ *has* no less integrity *than* $\mathsf{R}$ *iff*

$$\mathit{exception}\ \mathsf{E}\ \mathsf{S} \subseteq \mathit{exception}\ \mathsf{E}\ \mathsf{R}.$$

*That is written* $\mathsf{R} \mapsto \mathsf{S}$. $\triangle$

He then proves that if $\mathsf{R}$ is refined by $\mathsf{S}$ ($\mathsf{R} \sqsubseteq \mathsf{S}$) then $\mathsf{S}$ has no less integrity than $\mathsf{R}$ ($\mathsf{R} \mapsto \mathsf{S}$).

That result may seem slightly at odds without the characterisation of integrity from Section 3.2.1, where integrity is shown to be an information–flow policy, and Section 4.2.2, where an example shows that security properties are not preserved by refinement. There is not a conflict, Jacob has characterized integrity separately from other security properties and proved a result for that particular definition.

In [69] McLean defines non–interference and then refinement by saying that $S^*$ is a refinement of $S$ iff $S^* \vdash S$. He defines $\vdash$ in [65], it means that any trace assertion that can be derived from $S$ can be

derived from $S^*$. Derivation is given a definition in terms of axioms and rules of inference and proved sound and complete with respect to a trace semantics (see [65, pages 615–626] for details).

Hence, if non–interference can be proved of $S$ it can be proved of a refinement $S^*$ of $S$. This is not, as McLean points out, the usual notion of functional refinement and does not conflict with the example in Section 4.2.2.

He does say, however, that

> it is best to prove noninterference on the most abstract-level possible so that our noninterference proof can survive program changes that leave functional behaviour intact. [69, page 45]

He then uses the definition of refinement based on derivations to prove that a program meets a specification and hence is non–interfering.

The major difference between the secure replacement relation here and McLean's refinement relation is that his relation is not restricted to non–interference or security at all. In fact in the argument above any property of $S$ can replace non–interference. McLean's definition of refinement is very general and rather than preserving specific properties can be thought of as preserving *proofs*—if assertion $A$ is derivable from $S$ and $S^*$ is a refinement of $S$ then assertion $A$ is derivable from $S^*$.

# Chapter 8

# Conclusion

## 8.1 Review

The principal contribution of this thesis is the clear expression of equivalence of traces to capture a user's perceived state, and that definition's application to the expression of security policies. We have used CSP as a framework in which to investigate security and have shown how to capture the notion of security (in particular non–interference) using the models and algebra of CSP.

The three possible security breaches mentioned in [59] and commonly called (lack of) confidentiality, integrity and availability (denial of service) are often used to describe the security requirements of a system. Additionally, the idea of a system being in equivalent states, from the viewpoint of a user, is common place in literature about secure systems (see Chapter 7 for examples). We have taken the latter approach, defining equivalence in a natural manner (Definition 3.7), and used the equivalent states to define non–interference (Definition 3.9).

We defined a preordering of systems—the secure replacement ordering in Definition 4.1—which can be used in the stepwise development of secure systems.

Having expressed the definition of equivalence in CSP the extensive algebra is used in Section 4.3 to give a collection of rules for the development of systems satisfying non–interference. We extended the standard notion of non–interference to one which captures the idea of transactions not interfering with other users. Non–interference turned out to be a special case of transaction non–interference. Section 4.3.3 focuses on transaction non–interference and characterizes that property by a collection of laws.

The collection of laws given as part of Chapter 4 are intended to

be of practical use in the construction and verification of secure systems. In order to determine their utility the case study in Chapter 5 demonstrates how those laws can be used in developing a multi–user multi–level secure system. The system $\mathsf{S}$ in Chapter 5 (Definition 5.31) has the common military multi–level security policy expressed using non–interference, developed using the laws in Section 4.3.3 and enforced by the restrictions on reading and writing common to many secure systems ('no read–up' and 'no write–down') and the design of the interface processes $\mathsf{I}_i$ (Definition 5.26) which ensures that there is no covert communication through the system (within the confines of the model: untimed CSP).

At the end of the case study (Section 5.5) the possible flaws in the occam implementation are discussed, with particular reference to communication, that are outside the security policy. In Chapter 6 the definitions in Chapter 3 are extended to show how equivalence including time can be encoded. A small collection of laws is given for non–interference within that framework.

## 8.2 Future Directions

Foley shows in [24] that non–interference is applicable to a number of policies and Goguen and Meseguer used conditional non–interference—see Section 7.4.1—to express other security policies. The definition of non–interference given here could be used to explore the sorts of security policy that can be expressed using non–interference and the laws used to develop systems satisfying those properties.

The work in Chapter 6 needs extension: as a motivating factor we have investigated non–interference in timed systems and the laws should be extended to cover recursion. A significant case study of the utility of the results in Chapter 6 would, no doubt, lead to an enhancement of those results. Those results need to be extended to timed models that are more complex than timed traces and laws verified for those models. In Chapter 4 much use was made of the algebra of CSP; a similar algebra of Timed CSP could be applied to the proof of laws for security in that model.

Completeness of the laws in Section 4.3.3 has not been shown. Once the laws of timed non–interference have been extended to a semantic model more detailed than timed traces (and to include recursion) it would be worthwhile to verify the completeness of those laws.

None of the work in this thesis has concerned the possibility of communication (interference) through the analysis of the infinite traces of a system. One example of infinite traces comes in the analysis of the

probabilistic properties of systems. Such security breaches through analysis of probabilities are a concern in high–security systems (see, for example, [33, 34]). Recent work on Probabilistic CSP (PCSP), in [90], could be used to give definitions of equivalence and non–interference which capture indistinguishability of infinite traces. Such an analysis would be able to differentiate two systems which, in CSP, both appear to be Run on the alphabet $\{ue, ve\}$ but which have different asymptotic frequencies of the events $ue$ and $ve$. Such a system would appear non–interfering ($\{ue\} \not\rightsquigarrow \{ve\}$) in CSP (see Lemma 4.22, part 3), but could, in practice, be considered interfering if asymptotic frequencies of events were analysed.

We have defined two orders on processes which can be used when performing refinements and which preserve non–interference. Appendix A highlights some further properties of those definitions and gives proof rules which show when a downwards simulation is a secure replacement. That work can be extended and it would be useful to apply secure replacement to strengthen the rules given in [75] to produce a similar collection for the development of secure systems.

Another major concern of the owners of secure systems is the process by which those systems were developed, and a common requirement is the use of proof–checking tools in that process (see [91, Chapter 4]). The collection of laws given here (and particularly those for non–interference in Section 4.3.3) could be coded into a proof system and used to validate a case study of a secure system.

In our approach to security no meaning is attached to the events of a system: all events are considered equal. In a secure system particular events (or operations) might be more critical than others. An approach to security would be to define an ordering on events, which expresses the sensitivity of particular events (an event to downgrade all files to unclassified might be considered more sensitive than one which displays the time), and use that order to define a 'more secure than' ordering on systems. The properties of the 'more secure than' order should be investigated as well as its relationship with conventional 'more deterministic than' orderings.

Another way of viewing the process of refinement of a secure system is to consider a non–interfering system and systems which are identical to it from the viewpoint of a particular user. If $u \not\rightsquigarrow v : \mathsf{S}$ then there is a collection of processes $\mathsf{S}_i$ such that

$$(\mathsf{S}@u = \mathsf{S}_i@u) \wedge (\mathsf{S}@v \sqsubseteq \mathsf{S}_i@v)$$

(recall Definition 7.5). The $\mathsf{S}_i$ are the processes that appear identical to $u$, but are refinements of $\mathsf{S}$ from $v$'s viewpoint. The condition

under which $u \not\rightsquigarrow v : \mathsf{S}_i$ should be investigated to highlight the way in which $v$'s possible interactions can be altered without introducing interference. Using that refinement ordering a suitable definition of the 'smallest' non–interfering process can be made. Such a 'smallest' process would be the most deterministic from $v$'s viewpoint. The roles of $u$ and $v$ in that analysis could also be reversed to discover the changes that can be made to $u$'s interactions.

# Bibliography


**1.** P. G. Allen. A comparison of non–interference and non–deducibility using CSP. In *Proceedings of the 1991 IEEE Computer Security Workshop*, pages 43–54. IEEE Computer Society Press, 1991.

**2.** W. R. Ashby. *An Introduction to Cybernetics*. Methuen, London, 1976.

**3.** R. J. R. Back and R. Kurki–Suonio. Decentralization of process nets with centralized control. In *2nd ACM SIGACT–SIGOPS Symposium on Principles of Distributed Computing*, pages 131–142, August 1983.

**4.** G. Barrett, M. Goldsmith, G. Jones, and A. Kay. The meaning and implementation of `PRI ALT` in occam. In C. Askew, editor, *Proceedings of the 9th* occam *User Group Technical Meeting*, pages 37–46. IOS, 19–21 September 1988.

**5.** B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, 1983.

**6.** D. E. Bell. Concerning "modeling" of computer security. In *Proceedings of the 1988 IEEE Symposium on Security and Privacy*, pages 8–13. IEEE Computer Society Press, 1988.

**7.** D. E. Bell and L. J. LaPadula. Secure computer systems: A mathematical model. Technical Report ESD–TR–73–278–II. The MITRE Corporation, May 1973.

**8.** ——— Secure computer systems: Mathematical foundations. Technical Report ESD–TR–73–278–I. The MITRE Corporation, March 1973.

**9.** ——— Secure computer systems: A refinement of the mathematical model. Technical Report ESD–TR–73–278–III. The MITRE Corporation, April 1974.

**10.** ——— Secure computer system: Unified exposition and MULTICS implementation. Technical Report ESD–TR–75–306. The MITRE Corporation, March 1976.

**11.** K. J. Biba. Integrity considerations for secure computer systems. Technical Report MTR–3153 (ESD–TR–76–372). The MITRE Corporation, April 1976.

**12.** D. F. C. Brewer and M. J. Nash. The chinese wall security policy. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 206–258. IEEE Computer Society Press, 1989.

**13.** S. D. Brookes. *A Model for Communicating Sequential Processes.* D.Phil. Thesis, Oxford University, 1983.

**14.** S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(7):560–599, 1984.

**15.** S. D. Brookes and A. W. Roscoe. An improved failures model for communicating sequential processes. In *Proceedings NSF–SERC Seminar on Concurrency*, pages 281–305. Springer–Verlag LNCS 197, 1985.

**16.** R. Burger. *Computer Viruses: A high–tech disease.* Abacus, 1988.

**17.** R. Burnham. *The Specification of Security in Distributed Computing Systems.* M.Sc. Thesis, Oxford University, 1987.

**18.** H. R. Chivers and P. Y. A. Ryan, editors. *CESG Workshop on the Refinement of Security Properties.* CESG, GCHQ, March 1990. RESTRICTED; no pagination.

**19.** D. D. Clark and D. R. Wilson. A comparison of commercial and military computer security policies. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, pages 184–194. IEEE Computer Society Press, 1987.

**20.** J. Davies. *Specification and Proof in Real–time Systems.* D.Phil. Thesis, Oxford University, 1991.

**21.** J. Davies and S. A. Schneider. An introduction to timed CSP. Technical Report PRG–75. Oxford University Computing Laboratory, Programming Research Group, 1989.

**22.** D. E. R. Denning. *Cryptography and Data Security.* Addison–Wesley, 1983.

**23.** T. Fine, J. T. Haigh, and R. C. O'Brien. A general non–interference unwinding theorem. *Cipher*, pages 38–40. IEEE Computer Society Press, April 1989.

**24.** S. N. Foley. Aggregation and separation as non–interference properties. To appear in *Journal of Computer Security*.

**25.** ——— A universal theory of information flow. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, pages 116–122. IEEE Computer Society Press, 1987.

**26.** ——— A theory of strong noninterference, Private Communication, December 1991.

**27.** M. Gasser. *Building a Secure Computer System.* Van Nostrand Reinhold, 1988.

**28.** J. A. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society Press, 1982.

**29.** ——— Unwinding and inference control. In *Proceedings of the 1984 IEEE Symposium on Security and Privacy*, pages 75–86. IEEE Computer Society Press, 1984.

**30.** J. Graham–Cumming. *Formal Methods for Secure Systems*. M.Sc. to D.Phil. Transfer Dissertation, Oxford University, September 1990.

**31.** J. Graham–Cumming and J. W. Sanders. Secure refinement. In [18].

**32.** ——— On the refinement of non–interference. In *Proceedings of the 1991 IEEE Computer Security Workshop*, pages 35–42. IEEE Computer Society Press, 1991.

**33.** J. W. Gray III. Probabilistic interference. In *Proceedings of the 1990 IEEE Symposium on Security and Privacy*, pages 170–179. IEEE Computer Society Press, 1990.

**34.** ——— Toward a mathematical foundation for information flow security. In *Proceedings of the 1991 IEEE Symposium on Security and Privacy*, pages 21–34. IEEE Computer Society Press, 1991.

**35.** J. D. Guttman and M. E. Nadel. What needs securing? In *Proceedings of the Computer Security Foundations Workhop*, pages 34–57. The MITRE Corporation, 1988.

**36.** M. Hennessy. *Algebraic Theory of Processes.* The MIT Press, 1988.

**37.** C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

**38.** ——— A model for communicating sequential processes. In R. M. McKeag and A. M. McNaughton, editors, *On the Construction of Programs*, pages 229–248. Cambridge University Press, 1980.

**39.** ——— *Communicating Sequential Processes.* Prentice–Hall International, 1985.

**40.** C. A. R. Hoare, I. J. Hayes, J. He, C. C. Morgan, J. W. Sanders, I. H. Sorenson, J. M. Spivey, and B. A. Sufrin. Data refinement refined. Draft Paper. Limited Distribution outside Oxford University Computing Laboratory, May 1985.

**41.** INMOS Limited. occam *Programming Manual*, 1984. Prentice–Hall International.

**42.** Information Technology Security Evaluation Criteria. Technical Report CD–71–91–502–EN–C. Commission of the European Communities, June 1991.

**43.** J. L. Jacob. Basic theorems about security. To appear in *Journal of Computer Security.*

**44.** ——— *On Shared Systems.* D.Phil. Thesis, Oxford University, 1987.

**45.** ——— A security framework. In *Proceedings of the Computer Security Foundations Workhop*, pages 98–111. The MITRE Corporation, 1988.

**46.** ——— Security specifications. In *Proceedings of the 1988 IEEE Symposium on Security and Privacy*, pages 14–23. IEEE Computer Society Press, 1988.

**47.** ——— On the derivation of secure components. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 242–247. IEEE Computer Society Press, 1989.

**48.** ——— Separability and the detection of hidden channels. *Information Processing Letters*, 34(1):27–29, February 1990.

**49.** ——— The basic integrity theorem. In *Proceedings of the 1991 IEEE Computer Security Workshop*, pages 89–97. IEEE Computer Society Press, 1991.

**50.** ——— A uniform presentation of confidentiality properties. *IEEE Transactions on Software Engineering*, 17(11):1186–1194, November 1991.

**51.** ——— What is data security? Draft Paper. Limited Distribution outside Oxford University Computing Laboratory, 1991.

**52.** S. Jajodia. Panel discussion on the polyinstantiation problem: A position paper. In *Proceedings of the 1991 IEEE Computer Security Workshop,* page 235. IEEE Computer Society Press, 1991.

**53.** D. M. Johnson and F. J. Thayer. Security and the composition of machines. In *Proceedings of the Computer Security Foundations Workhop*, pages 72–89. The MITRE Corporation, 1988.

**54.** ——— Security properties consistent with the testing semantics for communicating processes. In *Proceedings of the 1989 IEEE Computer Security Workshop*, pages 9–21. IEEE Computer Society Press, 1989.

**55.** ——— Stating security requirements with tolerable sets. *ACM Transactions on Computer Systems*, 6(3):284–295, August 1989.

**56.** G. Jones. *Programming in* occam. Prentice–Hall International, 1987.

**57.** R. B. Jones and R. Stokes. Methods for specification–to–model correspondence. Technical Report DS/FMU/DSM/037, Issue 2.1. ICL Defence Systems, February 1990. Issued to coincide with [18].

**58.** B. W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, October 1973.

**59.** C. E. Landwehr. A survey of formal methods for computer security. Technical Report 8489. Naval Research Laboratory, Computer Science and Systems Branch, Information Technology Division, Washington D.C., 1981.

**60.** S. B. Lipner. A comment on the confinement problem. *ACM Operating Systems Review*, 9(5):192–196, 1975.

**61.** T. F. Lunt. Polyinstantiation: An inevitable part of a multilevel world. In *Proceedings of the 1991 IEEE Computer Security Workshop*, pages 236–238. IEEE Computer Society Press, 1991.

**62.** T. F. Lunt and D. Hsieh. The SeaView secure data base system: A progress report. In *Proceedings of the 1990 European Symposium on Research in Computer Security*, pages 3–13. AFCET, 1990.

**63.** D. McCullough. Specifications for multi–level security and a hook–up property. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, pages 161–166. IEEE Computer Society Press, 1987.

**64.** ——— Noninterference and the composability of security properties. In *Proceedings of the 1988 IEEE Symposium on Security and Privacy*, pages 177–186. IEEE Computer Society Press, 1988.

**65.** J. McLean. A formal method for the abstract specification of software. *Journal of the ACM*, 31(3):600–627, July 1984.

**66.** ——— A comment on the 'basic security theorem' of Bell and LaPadula. *Information Processing Letters*, 20:67–70, 1985.

**67.** ——— Reasoning about security models. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, pages 123–131. IEEE Computer Society Press, 1987.

**68.** ——— Security models and information flow. In *Proceedings of the 1990 IEEE Symposium on Security and Privacy*, pages 180–187. IEEE Computer Society Press, 1990.

**69.** ——— Proving noninterference and functional correctness using traces. *Journal of Computer Security*, (1):37–57, 1992.

**70.** J. McLean and C. Meadows. Composable security properties. *Cipher*, pages 27–37. IEEE Computer Society Press, Fall 1989.

**71.** C. Meadows. Panel discussion on the polyinstantiation problem: An introduction. In *Proceedings of the 1991 IEEE Computer Security Workshop,* page 234. IEEE Computer Society Press, 1991.

**72.** J. K. Millen. Covert channel capacity. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, pages 60–66. IEEE Computer Society Press, 1987.

**73.** C. C. Morgan. Private Communication, 1985.

**74.** ——— Of wp and CSP. In W. H. J. Feijn, A. J. M. van Gasteren, D. Gries, and J. Misra, editors, *Beauty is Our Business: A Birthday Salute to Edsger W. Dijkstra*, pages 319–326. Springer–Verlag, 1990.

**75.** ——— *Programming from Specifications.* Prentice–Hall International, 1990.

**76.** National Computer Security Center, NCSC, Fort Meade, Md. *Department of Defense Trusted Computer System Evaluation Criteria*, December 1985.

**77.** C. M. O'Halloran. A calculus of information flow. In *Proceedings of the 1990 European Symposium on Research in Computer Security*, pages 147–159. AFCET, 1990.

**78.** E.-R. Olderog and C. A. R. Hoare. Specification–oriented semantics for communicating processes. *Acta Informatica*, 23:9–66, 1986.

**79.** G. M. Reed. *A Uniform Mathematical Theory for Real–time Distributed Computing.* D.Phil. Thesis, Oxford University, 1988.

**80.** G. M. Reed and A. W. Roscoe. Metric spaces as models for real–time concurrency. In *Proceedings of the* $3^{rd}$ *Workshop on the Mathematical Foundation of Programming Language Semantics*, pages 331–343. LNCS 298, Springer–Verlag, 1987.

**81.** ——— A timed model for communicating sequential processes. *Theoretical Computer Science*, 58:249–261, 1988. (Also, Proceedings ICALP'86, LNCS 226, pages 314–323, Springer–Verlag, 1986).

**82.** A. W. Roscoe. *A Mathematical Theory of Communicating Sequential Processes.* D.Phil. Thesis, Oxford University, 1982.

**83.** J. Rushby. 'Proof of separability:' a verification technique for a class of security kernels. Technical report. SRI International, 1982.

**84.** ——— The SRI security model. Technical report. SRI International, 1985.

**85.** P. Y. A. Ryan. Some thoughts on unwinding. Technical report. GCHQ, CESG, 1989.

**86.** ——— A CSP formulation of non–interference. In *Proceedings of the 1990 IEEE Computer Security Workshop.* IEEE Computer Society Press, 1990. Only a title page in the proceedings; paper appeared as [87].

**87.** ——— A CSP formulation of non–interference. *Cipher*, pages 19–27. IEEE Computer Society Press, Winter 1991.

**88.** J. W. Sanders. Refinement for Z. Lecture Notes, Oxford University, 1988.

**89.** S. A. Schneider. *Correctness and Communication in Real–time Systems.* D.Phil. Thesis, Oxford University, 1989.

**90.** K. Seidel. *Probabilistic Communicating Processes.* D.Phil. Thesis, Oxford University, 1992.

**91.** C. T. Sennett, editor. *High–integrity Software.* Pitman, 1989.

**92.** J. M. Spivey. *The Z Notation.* Prentice–Hall International, 1989.

**93.** J. E. Stoy. *Denotational Semantics: the Scott–Strachey approach to Programming Language Theory.* The MIT Press, 1977.

**94.** D. Sutherland. A model of information. In *Proceedings of the 9th National Computer Security Conference*, pages 175–183. U. S. National Computer Security Center and U. S. National Bureau of Standards, 1986.

**95.** A. S. Tanenbaum. *Operating Systems: Design and Implementation.* Prentice–Hall International, 1987.

**96.** M. Thomas. Should we trust computers? BCS/Unisys Annual Lecture, British Informatics Society Ltd., July 4$^{\text{th}}$ 1988.

**97.** D. Welsh. *Codes and Cryptography.* Oxford Science Publications, 1988.

**98.** J. C. Wray. A methodology for the detection of timing channels. *Cipher*, pages 35–42. IEEE Computer Society Press, Winter 1991.

# Appendix A

# Another Model of Computation

## A.1 Introduction

The body of work in this thesis has concerned the application of CSP to the development of secure systems. In [32] we considered the development of systems expressed using state machines (or abstract data–types). This appendix shows how to translate the results given in CSP to systems using state–based specification and development.

State–machine definitions are common in the specification of secure systems (see, for example, [10, 28, 72, 84]) and the relationship between refinement and preservation of security properties (Section 4.2.2) is important when developing a system from such a specification.

This appendix considers the definition of non–interference in an abstract data–type by constructing the equivalence relation $\approx_v^{\mathsf{S}}$. We recall the definition of the complete, state–based refinement technique called downwards simulation and use it in this appendix. We show how secure replacement (Section 4.2.3) is related to downwards simulation (see Section A.2.2 below). Other refinement methods are applicable to the methods of this appendix, we focus on downwards simulation to give a definite example.

We call the conjunction of refinement (in this case downwards simulation) and secure replacement *secure refinement*: refinement which preserves the functional and security properties of a system. Example 4.2 expressed as an abstract data–type (ADT) using Z ([92]) is used to illustrate secure refinement. The relationship between the CSP and ADT expressions of a system is given by a 'natural' transformation called, following local tradition, the *garage map* (Section A.2.1).

## A.2 Abstract Data–types

Action Systems (see [3]) provide a way of giving a specific behaviour of an abstract data–type, but for our purposes the following definitions suffice. The relationship between CSP and Action Systems is given in [74]; the relationship between CSP and ADTs, which we use, is given below.

In [40] (and earlier work, see, for example, [37]) an abstract data–type is defined for use in describing the events (operations) and states of a system. An ADT has three parts: a set of states; a set of initial states and a set of events.

An event is a predicate (or relation) on states and input and output; the event $e(s, \alpha?, \beta!, s')$ denotes an event that starting in state $s$ with input $\alpha?$ gives output $\beta!$ and moves to state $s'$. In keeping with the nature of events in CSP we consider $e(s, \alpha?, \beta!, s')$ as atomic.

**Definition A.1** *An* abstract data–type $\mathsf{S}$ *is a triple* ($\mathsf{SS}$, $\mathsf{SS}_0$, $\mathsf{SE}$) *where:*

- ◇ $\mathsf{SS}$ *is the set of states of* $\mathsf{S}$*;*
- ◇ $\mathsf{SS}_0 \subseteq \mathsf{SS}$ *is the set of initial states of* $\mathsf{S}$*;*
- ◇ $\mathsf{SE}$ *is the set of events* (*often called* operations) *of* $\mathsf{S}$. △

From the viewpoint of a user events such as $e(s, \alpha?, \beta!, s')$ have the state information concealed to become $e(\alpha?, \beta!)$.

The behaviour of an ADT is defined recursively as follows: initially one of the states from $\mathsf{SS}_0$ is chosen non–deterministically; then the environment engages in any $e \in \mathsf{SE}$ with associated input and output and moves into one of the possible $s'$ non–deterministically (cf. Definition A.6). The system continues from state $s'$ in that fashion until there are no $e$s available: the system deadlocks in that case.

An event is available when its precondition is true; the precondition holds for some state $s$ and input $\alpha?$ if there is some output $\beta!$ and state $s'$ for which $e(s, \alpha?, \beta!, s')$ holds.

**Definition A.2** *If* $e \in \mathsf{SE}$ *then the* precondition *of* $e$ *is*

$$\mathbf{pre}\ e(s, \alpha?) \triangleq \exists\, \beta! \,\exists\, s' \in \mathsf{SS} \bullet e(s, \alpha?, \beta!, s'). \qquad \triangle$$

The set of internal traces of an ADT is the set of possible sequences of events with the state information exposed.

**Definition A.3** *If* $\mathsf{S}$ *is the abstract data–type* ($\mathsf{SS}$, $\mathsf{SS}_0$, $\mathsf{SE}$) *then the set of* internal traces $\rho\mathsf{S}$ *is defined by*

$$\begin{array}{rcl} \langle\rangle & \in & \rho\mathsf{S}; \\ t \in \rho\mathsf{S} & \Rightarrow & t^\frown\langle e(s,\alpha?,\beta!,s')\rangle \in \rho\mathsf{S}, \end{array}$$

*for all* $e(s,\alpha?,\beta!,s')$ *such that* **pre** $e(s,\alpha?)$ *holds and* $t$ *can leave the ADT in state* $s$*.* △

The set of external traces (or just the set of *traces*) is defined by removing the state information from the external traces; the strict and distributive function *conceal*, where *conceal* $\langle e(s,\alpha?,\beta!,s')\rangle \triangleq \langle e(\alpha?,\beta!)\rangle$, does just that.

**Definition A.4** *If* $\mathsf{S}$ *is the abstract data–type* ($\mathsf{SS}$, $\mathsf{SS}_0$, $\mathsf{SE}$) *then the set of external* traces *of* $\mathsf{S}$ *is* $\tau\mathsf{S} \triangleq \{\mathit{conceal}\ t \mid t \in \rho\mathsf{S}\}$*.* △

After engaging in a trace the ADT may be in one of a number of states, one of the reachable states.

**Definition A.5** *If* $\mathsf{S}$ *is the abstract data–type* ($\mathsf{SS}$, $\mathsf{SS}_0$, $\mathsf{SE}$) *and* $t \in \tau\mathsf{S}$ *then the set of* states reachable *from* (*non–empty*) $t$ *is*

$$[\![t]\!]^{\mathsf{S}} \triangleq \{s' \mid \mathit{conceal}(t'^\frown\langle e(s,\alpha?,\beta!,s')\rangle) = t\}.$$

*The states reachable from* $\langle\rangle$ *are the initial states* $\mathsf{SS}_0$*.* △

Note that a *user* of an ADT is simply a subset of the set of events, hence $v$ is a user of ADT $\mathsf{S}$ iff $v \subseteq \mathsf{SE}$.

### A.2.1 The Garage Map

The garage map of an abstract data–type was proposed by [73] and relates an ADT to a CSP process with the same behaviour. Its importance here is that it may be used in converting our results from a CSP framework to an ADT framework, and so saves us doing that explicitly.

The garage map of an ADT is the process which initially selects one of the ADT's initial states non–deterministically and then in any state $s$ gives the environment the choice between all the events of the ADT whose precondition is true. The environment has the choice of input and event, but not the state; an internal choice is made over which output and state is chosen.

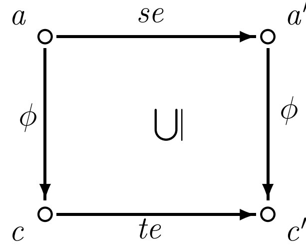


Figure 5: Downwards Simulation.

**Definition A.6** *If* $\mathsf{S}$ *is the abstract data–type* $(\mathsf{SS}, \mathsf{SS}_0, \mathsf{SE})$ *then the* garage map *of* $\mathsf{S}$ *is the process*

$$\mathsf{S} \triangleq \sqcap_{s \in \mathsf{SS}_0} \mathsf{S}_s;$$

$$\mathsf{S}_s \triangleq \Box_{\mathbf{pre}\ e(s,\alpha?)} \sqcap_{e(s,\alpha?,\beta!,s')} e.(\alpha?, \beta!) \to \mathsf{S}_{s'}.$$

△

## A.2.2 Refinement of an ADT

As noted above the behaviour of an ADT is given in terms of its traces. Refinement of an abstract data–type is the subset relationship on those traces. Hence ADT $\mathsf{A}$ is refined by $\mathsf{C}$ iff $\tau\mathsf{C} \subseteq \tau\mathsf{A}$, cf. the definition of refinement in CSP given in Section 2.4.

In [40] (and elsewhere—see, for example, [88]) proof rules are given to guarantee that an ADT refines another by considering properties of their events. The relationship between them is expressed in terms of a relation between their states and rules are given to ensure refinement.

The technique of downwards simulation (sometimes called semi–simulation) is expressed in terms of three rules concerning the events of an ADT.

The rules given in Definition A.7 form the proof obligations of a downwards simulation (they are taken from [40], but see also [88]).

**Definition A.7** *If* $\mathsf{S}$ *and* $\mathsf{T}$ *are abstract data–types then* $\phi : \mathsf{SS} \leftrightarrow \mathsf{TS}$ *is a* downwards simulation *for* $\mathsf{S}$ *and* $\mathsf{T}$ *iff*

$$c \in \mathsf{TS}_0 \Rightarrow \exists\, a \in \mathsf{SS}_0 \bullet a\ \phi\ c\,; \tag{7}$$

$$(a\ \phi\ c \wedge \mathbf{pre}\ se(a, \alpha?)) \Rightarrow \mathbf{pre}\ te(c, \alpha?)\,; \tag{8}$$

$$(a\ \phi\ c \wedge \mathbf{pre}\ se(a, \alpha?) \wedge te(c, \alpha?, \beta!, c')) \Rightarrow \exists\, a' \in \mathsf{SS} \bullet (se(a, \alpha?, \beta!, a') \wedge a'\ \phi\ c'). \tag{9}$$

*for each* $se \in \mathsf{SE}$ *with corresponding* $te \in \mathsf{TE}$, $a, a' \in \mathsf{SS}$, *and* $c, c' \in \mathsf{TS}$.
$\triangle$

Obligation (1) in Definition A.7 means that for any initial state of $\mathsf{T}$ there must be a corresponding initial state of $\mathsf{S}$. Obligation (2) means that if *se*'s precondition is true in some state there must be a corresponding state of $\mathsf{T}$ for which *te*'s precondition holds. Together with Obligation (3) (1) and (2) guarantee that Figure 5 sub–commutes. Theorem 2.1 in [40], that downwards simulation is a sufficient condition for refinement, enables us to write $\mathsf{S} \sqsubseteq \mathsf{T}$ when there is a downwards simulation $\phi$ from $\mathsf{S}$ to $\mathsf{T}$. That is, when there is a downwards simulation from $\mathsf{S}$ to $\mathsf{T}$ the set of traces of $\mathsf{T}$ is a subset of the set of traces of $\mathsf{S}$. Hence the rules in Definition A.7 relating to individual events give a result concerning sequences of those events. The utility of Definition A.7 follows from that result as it enables a system to be reasoned about at the level of events and a result deduced about all such sequences of events.

### A.2.3 Secure Refinement

We consider secure replacement and its relationship to downwards simulation (see Section A.2.2 and Definition A.7).

If $\mathsf{C}$ is a downwards simulation of $\mathsf{A}$ then $\tau\mathsf{C} \subseteq \tau\mathsf{A}$ (see Theorem 2.1 in [40]). Trace inclusion is used in the following lemmas. Lemma 4.5 shows that, when considering a downwards simulation, proof of semi–secure replacement is enough to guarantee secure replacement.

We give sufficient conditions for a downwards simulation (see Section A.2.2) to be a secure replacement by instantiating the function $\theta_v$ of Theorem 4.2 and by examining the commuting diagram given in Figure 7.

Consider two abstract data–types $\mathsf{A}$ and $\mathsf{C}$ with $\phi$ a downwards simulation from $\mathsf{A}$ to $\mathsf{C}$. (We have been working in CSP so the 'states' of those processes are defined as equivalence classes of traces. They are the global states given as the equivalence classes of $\approx^{\mathsf{A}}$ and $\approx^{\mathsf{C}}$: see Section 3.2.2. Here we define $\mathsf{A}$ and $\mathsf{C}$ as abstract data–types (see Section A.2); the relationship between abstract data–types and CSP is given in Section A.2.1).

We define a relation between the $v$–equivalence classes of $\mathsf{A}$ and the $v$–equivalence classes of $\mathsf{C}$, which we denote ${}^{\mathsf{A}}\Theta_v^{\mathsf{C}}$. It is defined in terms of the abstraction $\phi$, which relates the states of those two processes, and projection relations such as $\pi_v^{\mathsf{A}}$ which relates a state of $\mathsf{A}$ to the $v$–equivalence classes consistent with that state.

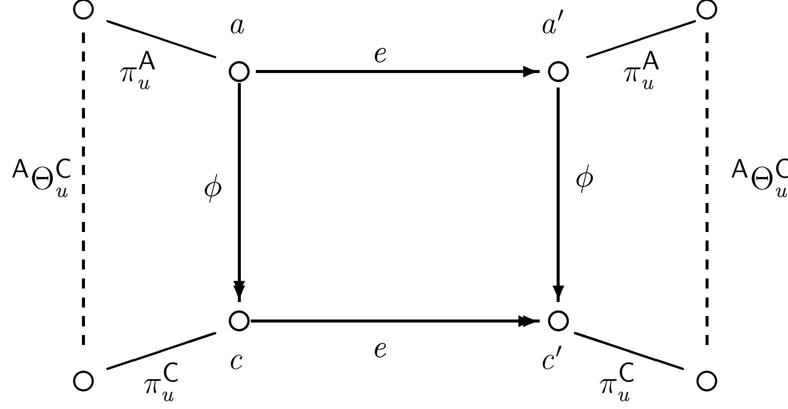


Figure 6: Downwards Simulation with Θ.

**Definition A.8** *If an abstract data–type* A *has user* $v$ *then* $\pi_v^{\mathsf{A}} : \mathsf{AS} \leftrightarrow \mathcal{C}_v^{\mathsf{A}}$ *is defined by*

$$\forall\, a \in \mathsf{AS}\, \forall\, t \in \tau\mathsf{A} \bullet a \in [\![t]\!]^{\mathsf{A}} \Rightarrow a\ \pi_v^{\mathsf{A}}\ [t]_v^{\mathsf{A}}. \qquad \triangle$$

**Definition A.9** *If* $\phi$ *is a downwards simulation from abstract data–type* A *to* C*, with user* $v$*, then* ${}^{\mathsf{A}}\Theta_v^{\mathsf{C}}$ *is defined by*

$$\pi_v^{\mathsf{A}}\ ;\ {}^{\mathsf{A}}\Theta_v^{\mathsf{C}} \triangleq \phi\ ;\ \pi_v^{\mathsf{C}}. \qquad \triangle$$

Recall Theorem 4.2 which shows that secure replacement is guaranteed when $\theta_v$ can be shown to be a particular function. If ${}^{\mathsf{A}}\Theta_v^{\mathsf{C}}$ is a function then we find (see Lemma A.1) that ${}^{\mathsf{A}}\Theta_v^{\mathsf{C}}$ maps the equivalence class $[t]_v^{\mathsf{A}}$ to $[t]_v^{\mathsf{C}}$.

**Lemma A.1** *If* ${}^{\mathsf{A}}\Theta_v^{\mathsf{C}}$ *is a function and* $\phi$ *is a downwards simulation from* A *to* C *then* $\forall\, t \in \tau\mathsf{C} \bullet\ {}^{\mathsf{A}}\Theta_v^{\mathsf{C}}\ ([t]_v^{\mathsf{A}}) = [t]_v^{\mathsf{C}}$.

**Proof** We prove that, for any $t$ in $\tau\mathsf{C}$, ${}^{\mathsf{A}}\Theta_v^{\mathsf{C}}\ ([t]_v^{\mathsf{A}}) = [t]_v^{\mathsf{C}}$, by induction on the length of the trace $t$.

**(base case)** $t = \langle\rangle$ hence we have $a\ \pi_v^{\mathsf{A}}\ [\langle\rangle]_v^{\mathsf{A}}$ for $a \in \mathsf{AS}_0$. By condition (7) for any $a \in \mathsf{AS}_0$ there exists $c \in \mathsf{CS}_0$ such that $a\ \phi\ c$ and Definition A.8 gives $c\ \pi_v^{\mathsf{C}}\ [\langle\rangle]_v^{\mathsf{C}}$. So as ${}^{\mathsf{A}}\Theta_v^{\mathsf{C}}$ is a function we obtain ${}^{\mathsf{A}}\Theta_v^{\mathsf{C}}\ ([\langle\rangle]_v^{\mathsf{A}}) = [\langle\rangle]_v^{\mathsf{A}}$.

**(inductive step)** Suppose that for some $n$ and for any $t \in \tau\mathsf{C}$ such that $\#t \leq n\ {}^{\mathsf{A}}\Theta_v^{\mathsf{C}}\ ([t]_v^{\mathsf{A}}) = [t]_v^{\mathsf{C}}$. Hence there exist $a \in \mathsf{AS}$ and $c \in \mathsf{CS}$ such that $a\ \phi\ c$, $a\ \pi_v^{\mathsf{A}}\ [t]_v^{\mathsf{A}}$ and $c\ \pi_v^{\mathsf{C}}\ [t]_v^{\mathsf{C}}$.

Consider the trace $t^\frown\langle e\rangle$ in $\tau\mathsf{C}$. Hence by downwards simulation conditions (8) and (9) there exist $a' \in [\![t^\frown\langle e\rangle]\!]^{\mathsf{A}}$ and $c' \in [\![t^\frown\langle e\rangle]\!]^{\mathsf{C}}$ such that $a'\ \phi\ c'$. Definition A.8 shows that $a'\ \pi_v^{\mathsf{A}}\ [t^\frown\langle e\rangle]_v^{\mathsf{A}}$ and $c'\ \pi_v^{\mathsf{C}}\ [t^\frown\langle e\rangle]_v^{\mathsf{C}}$. Hence ${}^{\mathsf{A}}\Theta_v^{\mathsf{C}}\ ([t^\frown\langle e\rangle]_v^{\mathsf{A}}) = [t^\frown\langle e\rangle]_v^{\mathsf{C}}$. □

Hence if ${}^{\mathsf{A}}\Theta_v^{\mathsf{C}}$ is a function then $\mathsf{C}$ is a secure replacement for $\mathsf{A}$ under a downwards simulation. Thus an additional proof obligation to ensure both preservation of functional and security properties is '${}^{\mathsf{A}}\Theta_v^{\mathsf{C}}$ is a function'.

Theorem A.1 brings together the definition of ${}^{\mathsf{A}}\Theta_v^{\mathsf{C}}$, Theorem 4.2 and Lemma A.1 to give a result useful in performing a secure refinement.

**Theorem A.1** *If ${}^{\mathsf{A}}\Theta_v^{\mathsf{C}}$ is a function and $\phi$ is a downwards simulation from $\mathsf{A}$ to $\mathsf{C}$ then $\mathsf{A} \preceq_v \mathsf{C}$.*

**Proof** Follows from Lemma A.1 and Theorem 4.2. □

Further, somewhat simpler, conditions are obtained by considering each step of the proof of a downwards simulation. In the following it will be convenient to refer to Figure 7 where a diagram depicts a downwards simulation by $\phi$.

Theorem A.2 gives sufficient conditions for a downwards simulation to be a secure replacement. The conditions are given in terms of an abstract data–type and its events. Condition (10) says that for two states to be $v$–equivalent all events must have the same precondition on those states and condition (11) says that, additionally, the same input, output and resulting $v$–equivalent states must be possible.

**Theorem A.2** *Assume $\phi$ is a downwards simulation from $\mathsf{A}$ to $\mathsf{C}$ and if for all traces $s$ and $t$ in $\tau\mathsf{A}$ which are $v$–equivalent ($s \approx_v^{\mathsf{A}} t$), with states $s_0 \in [\![s]\!]^{\mathsf{A}}$ and $s_1 \in [\![t]\!]^{\mathsf{A}}$ and corresponding concrete states $t_0, t_1 \in \mathsf{CS}$ ($s_0\ \phi\ t_0$ and $s_1\ \phi\ t_1$) such that*

$$\mathbf{pre}\ e(t_0, \alpha?) \iff \mathbf{pre}\ e(t_1, \alpha?) \tag{10}$$

$$e(t_0, \alpha?, \beta!, t_0') \iff e(t_1, \alpha?, \beta!, t_1'), \tag{11}$$

*then $\mathsf{A} \sqsubseteq_v \mathsf{C}$ and hence $\mathsf{A} \preceq_v \mathsf{C}$.*

**Proof** We prove that $(\approx_v^{\mathsf{A}} \diamond \tau\mathsf{C}) \subseteq \approx_v^{\mathsf{C}}$. Suppose $s$ and $t$ are traces of $\mathsf{C}$ and $s \approx_v^{\mathsf{A}} t$. Then we require that $s \approx_v^{\mathsf{C}} t$; that is for all non–empty $r \in u^*$

$$s^\frown r \in \tau\mathsf{C} \Leftrightarrow t^\frown r \in \tau\mathsf{C}$$

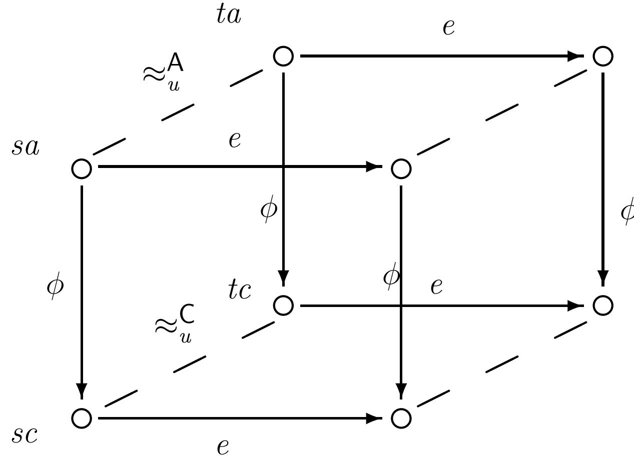


Figure 7: Secure Downwards Simulation.

(see Definition 3.4). We prove that by induction on $r$ (we start with empty $r$).

**(base case)** As $s$ and $t$ are both traces of $\mathsf{C}$ this case holds.

**(inductive step)** Suppose that for all $r \in u^*$ such that $\#r \leq n$ (for some $n$) $s^\frown r \in \tau\mathsf{C} \Leftrightarrow t^\frown r \in \tau\mathsf{C}$ and consider $r' = r^\frown\langle e(\alpha?, \beta!)\rangle$ where $e \in u$.

If $s^\frown r \notin \tau\mathsf{C}$ then $s^\frown r' \notin \tau\mathsf{C}$ and this case is trivially true.

If $s^\frown r \in \tau\mathsf{C}$ then we must prove that $s^\frown r' \in \tau\mathsf{C} \Leftrightarrow t^\frown r' \in \tau\mathsf{C}$. Consider states $sc \in [\![s^\frown r]\!]^{\mathsf{C}}$ and $tc \in [\![t^\frown r]\!]^{\mathsf{C}}$ with corresponding $sa \in [\![s^\frown r]\!]^{\mathsf{A}}$ and $ta \in [\![t^\frown r]\!]^{\mathsf{A}}$. Downwards simulation shows that $sa\ \phi\ sc$ and $ta\ \phi\ tc$ and $s \approx_v^{\mathsf{A}} t$ and conditions (10) and (11) then show that $s^\frown r^\frown\langle e(\alpha?, \beta!)\rangle \in \tau\mathsf{C} \Leftrightarrow t^\frown r^\frown\langle e(\alpha?, \beta!)\rangle \in \tau\mathsf{C}$.

Hence $\mathsf{A} \sqsubseteq_v \mathsf{C}$ and Lemma 4.5 gives $\mathsf{A} \preceq_v \mathsf{C}$. □

## A.3 Example of a Secure Refinement

We use the Z notation to specify a simple system which we show to be non–interfering by translating to CSP using the garage map (Definition A.6) and using the laws of Section 4.3.3.

Example 4.2 is reworked in Z and related to the garage map. We see that the definition of the system given in Example 4.2 is the garage map of the Z specification given here. Hence the guarantee of non–interference given by Lemma 4.2 shows that the Z specification has the same non–interference property.

**Example A.1** The refinement method downwards simulation (see Section A.2.2) is used to transform an abstract system with two events to a corresponding concrete system. The concrete system has the same data structure as the abstract, but non–determinism in the abstract system is resolved in the concrete and that introduces a security breach.

The abstract system stores binary numbers:

$$B == \{0, 1\}.$$

One number is stored for each of its two users $u$ and $v$; $uSt$ is the state component for $u$ and $vSt$ for $v$.

$$\begin{array}{l} \hline \textit{Abstract} \\ \hline uSt, vSt : B \\ \hline \end{array}$$

Initially both state components are set to 0.

$$\begin{array}{l} \hline \textit{Init} \\ \hline \Delta Abstract \\ \hline uSt' = vSt' = 0 \\ \hline \end{array}$$

Each user has an event: $u$ has $uUpdate$ and $v$ has $vUpdate$. They each use their event to update their part of the state.

$uUpdate$ accepts a number (from $u$) as input, updates $uSt$ with that number and the returns (to $u$) the previous value of $uSt$; $vSt$ is unchanged.

$$\begin{array}{l} \hline \textit{uUpdate} \\ \hline \Delta Abstract \\ uIn? : B \\ uOut! : B \times B \\ \hline uSt' = uIn? \land \mathbf{fst}\ uOut! = uSt \\ vSt' = vSt \\ \hline \end{array}$$

$v$ has a similar event $vUpdate$ which updates $vSt$ in the same way that $uUpdate$ alters $uSt$; $uSt$ is unchanged by $vUpdate$.

$$\begin{array}{l} \hline \textit{vUpdate} \\ \hline \Delta Abstract \\ vIn? : B \\ vOut! : B \times B \\ \hline uSt' = uSt \\ vSt' = vIn? \land \mathbf{fst}\ vOut! = vSt \\ \hline \end{array}$$

Using the garage map (Definition A.6) the abstract system is translated to the CSP process $\mathsf{A}$ (which is the process $\mathsf{A}$ of Example 4.2).

$$\begin{array}{lcl}
\mathsf{A} & = & \mathsf{A}_0^0; \\
\mathsf{A}_{vSt}^{uSt} & = & (\mathop{[\!]}_{uSt' \in B} \mathop{\sqcap}_{b \in B} uUpdate.(uSt', uSt, b) \rightarrow \mathsf{A}_{vSt}^{uSt'}) \\
& & [\!] \\
& & (\mathop{[\!]}_{vSt' \in B} \mathop{\sqcap}_{b \in B} vUpdate.(vSt', vSt, b) \rightarrow \mathsf{A}_{vSt'}^{uSt}).
\end{array}$$

Both the security policies $u \not\rightsquigarrow v : \mathsf{A}$ and $v \not\rightsquigarrow u : \mathsf{A}$ hold as either of the events can occur at any time and neither of them is affected by the occurrence of the other event. Lemma 4.2 proves those non–interference properties of $\mathsf{A}$.

There is non–determinism in both $uUpdate$ and $vUpdate$ as the second component of both $uOut!$ and $vOut!$ has not been specified. The concrete system does not contain that non–determinism as the second component of those outputs is specified. That negates the security policy by allowing each user to see the other user's state: the second component of each output is the other user's part of the state and so the whole state is output at each event.

The concrete and abstract systems have the same data structure and same initial state.

$$Concrete == Abstract.$$

$$InitC == Init.$$

$uUpdateC$ is the concrete version of $uUpdate$ and it is identical to $uUpdate$ except that the second component of $uOut!$ is now the value of $vSt$.

$$\begin{array}{|l}
\hline
uUpdateC \\
\Delta Concrete \\
uIn? : B \\
uOut! : B \times B \\
\hline
uSt' = uIn? \wedge uOut! = (uSt, vSt) \\
vSt' = vSt \\
\hline
\end{array}$$

A similar change has been made to $vUpdate$ to obtain $vUpdateC$; the second component of $vOut!$ is the value of $uSt$.

$$
\begin{array}{|l}
\hline
vUpdateC \\
\Delta Concrete \\
vIn? : B \\
vOut! : B \times B \\
\hline
uSt' = uSt \\
vSt' = vIn? \wedge vOut! = (vSt, uSt) \\
\hline
\end{array}
$$

The concrete system is a refinement of the abstract.

**Lemma A.2** *Concrete is a refinement of Abstract.*

**Proof** We verify conditions (7), (8) and (9).

- ◇ Condition (7) since *Abstract* and *Concrete* have the same initial states.
- ◇ Condition (8) is trivial as the preconditions of all the schemas are true.
- ◇ Condition (9) is $te \Rightarrow se$ as the preconditions are true. Consider the case when $te = uUpdateC$ and $se = uUpdate$. In the concrete example the value $uOut!$ is equal to $vSt$ and in the abstract it is any value at all. There is no other difference in those two schemas and so the implication holds. □

In the concrete system $u$ and $v$ can communicate and hence interfere: when one user changes his part of the state the other user can see that change by performing his update event.

The concrete system is $\mathsf{C}$ of Example 4.2 and the same conclusion of lack of non–interference is reached. Hence downwards simulation does not preserve security properties either (as would be expected).

The concrete system in Example A.1 is interfering; Example A.2 illustrates a refinement of *Abstract* that is non–interfering.

That example uses the techniques described in Section A.2.3 to prove that the refinement is non–interfering and describes how such a secure refinement of *Abstract* to *Concrete* fails.

**Example A.2** Consider the system *Secure* which has the same data structure as *Abstract* and the same initial state (described by *InitS*).

$$Secure == Abstract.$$

$$InitS == Init.$$

*uUpdateS* is the version of *uUpdate* in *Secure* and it is identical to *uUpdate* except that the second component of $uOut!$ is now always 0.

$$
\begin{array}{|l}
\text{__ } uUpdateS \text{ ________________________________________} \\
\Delta Secure \\
uIn? : B \\
uOut! : B \times B \\
\hline
uSt' = uIn? \land uOut! = (uSt, 0) \\
vSt' = vSt \\
\hline
\end{array}
$$

A similar change has been made to *vUpdate* to obtain *vUpdateS*; the second component of $vOut!$ is now 0.

$$
\begin{array}{|l}
\text{__ } vUpdateS \text{ ________________________________________} \\
\Delta Secure \\
vIn? : B \\
vOut! : B \times B \\
\hline
uSt' = uSt \\
vSt' = vIn? \land vOut! = (vSt, 0) \\
\hline
\end{array}
$$

The changes to the schema for the users' events has not introduced any security breach. All that has been done is to change an arbitrary value (the second part of each output is unspecified in *Abstract*) to the fixed value 0 (which is not dependent upon any part of the state). Hence $u$ does not interfere with $v$ in *Secure* and vice versa.

*Secure* is a refinement of *Abstract* (with the identity downwards simulation); the proof of that fact follows the form of the proof of Lemma A.2.

In Section A.2.3 the function $\Theta$ mapping equivalence classes of an abstract system to a concrete system is used to give a sufficient condition for a refinement to preserve the equivalence relation on traces (the traces of an ADT are described in Definition A.4).

In *Abstract* the equivalence classes from $u$'s viewpoint are characterized by $u$'s component of the state (since $u$'s event *uUpdate* on affects that part of the state). All traces in which the last *uUpdate* in each trace has the input are equivalent.

The function *lastu* maps a trace to the last value of $u$'s state input, or 0 if none has been input.

$$
\begin{array}{lcl}
lastu\ \langle\rangle & \triangleq & 0; \\
lastu\ t & \triangleq & (in?\ \mathrm{C}\ \mathbf{last}\ t = uUpdate.(in?, out!)\ \mathrm{B}\ lastu\ (\mathbf{init}\ t)).
\end{array}
$$

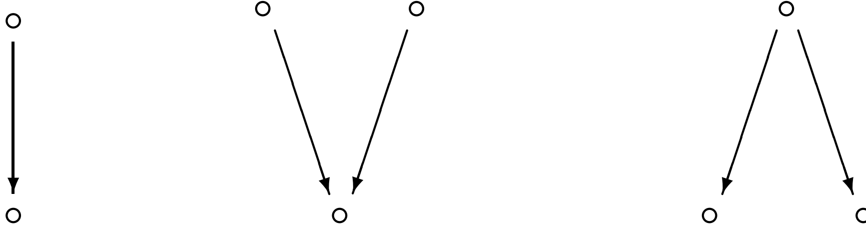

Figure 8: Mappings between equivalence classes.

The $u$–equivalence class of trace $t$ is the set of traces with the same value of *lastu*. (We abbreviate *Secure* to $\mathsf{S}$ and *Abstract* to $\mathsf{A}$.)

$$[t]^{\mathsf{S}}_u \triangleq \{s \mid s \in \tau\mathsf{S} \wedge \mathit{lastu}\ s = \mathit{lastu}\ t\}.$$

The $u$–equivalence classes of traces of *Secure* are the same; the classes of *Abstract* and *Secure* are in one–one correspondence and hence *Secure* is non–interfering by Theorem A.1. That case is illustrated by the left–hand diagram in Figure 8. The middle diagram illustrates the case when the equivalence classes get larger, but there is still a functional relationship between them and hence the refinement is secure.

The right–hand diagram of Figure 8 illustrates a refinement in which the equivalence classes of the concrete system are not related to the abstract classes by a function. An example of such a refinement is given by the systems *Abstract* and *Concrete* in Example A.1. The $u$–equivalence classes of *Concrete* are described by *lastu* and *lastv*:

$$\begin{aligned} \mathit{lastv}\ \langle\rangle &\triangleq 0; \\ \mathit{lastv}\ t &\triangleq (\mathit{in}?\ \lhd\ \mathbf{last}\ t = \mathit{vUpdate}.(\mathit{in}?, \mathit{out}!)\ \rhd\ \mathit{lastv}\ \mathbf{init}\ t). \end{aligned}$$

$$[t]^{\mathsf{C}}_u \triangleq \{s \mid s \in \tau\mathsf{C} \wedge \mathit{lastu}\ s = \mathit{lastu}\ t \wedge \mathit{lastv}\ s = \mathit{lastv}\ t\}.$$

There is no function mapping the $u$–classes of *Abstract* to *Concrete* as described by Theorem A.1. The equivalence class $[\langle\rangle]^{\mathsf{A}}_u$ contains both $[\langle\rangle]^{\mathsf{C}}_u$ and $[\langle \mathit{VUpdateC}(1,(0,0))\rangle]^{\mathsf{C}}_u$, but those two classes are not equal; hence ${}^{\mathsf{A}}\Theta^{\mathsf{C}}_u$ is not a function. $\triangle$

# Appendix B

# General Security Policies

## B.1 Introduction

In Chapter 3 the equivalence relation $\approx_v^{\mathsf{S}}$ (Definition 3.7) is used to define the information–flow property non–interference (Definition 3.9). That relation can be used to give other security specifications, and this appendix shows how.

Many computer security specifications are based around the idea of equivalent (or indistinguishable) states (see, for example, [28, 35, 63]) and so we use $\approx_v^{\mathsf{S}}$ as the basis of a notation for giving security specifications.

Many of the laws of non–interference can be extended to laws of more general security specifications, other new laws are derived from considering security specifications in general; Section B.3 gives laws of general security specifications.

## B.2 A Notation for Policy Specification

A security specification is a relation, $\mathcal{P}$, on sequences of events drawn from some alphabet and is defined for a user $v$. $s\,\mathcal{P}\,t$ means that from $v$'s viewpoint the traces $s$ and $t$ must be indistinguishable.

We will assume that such a relation is symmetric, since the order in which traces are examined does not affect whether they appear identical to a particular user. This idea of specifying security at an interface is largely due to [44, 77] and generalisation of properties such as non–interference is discussed in, for example, [23, 55].

**Definition B.1** *A* security policy (*or specification*) *for user $v$ is a symmetric relation, $\mathcal{P}$, on traces over some alphabet $A$ ($v \subseteq A$).* $\triangle$

Once we have the security specification we must consider whether a particular system satisfies it. Using the definition of $v$–equivalence the definition is simple: a system satisfies the security policy if all traces which the policy states should be indistinguishable are indistinguishable in the system under consideration.

**Definition B.2** *A system* $\mathsf{S}$ *satisfies the security specification* $\mathcal{P}$ *(on* $\alpha\mathsf{S}$*), for user* $v$*, when* $\mathcal{P} \subseteq \approx_v^{\mathsf{S}}$*. That is written* $\mathsf{S}\ \mathbf{sec}_v\ \mathcal{P}$*.* △

An example of a security specification is non–interference; Lemma B.1 shows how non–interference can be specified using Definition B.1 and how it is related to Definition B.2.

**Lemma B.1** *If system* $\mathsf{S}$ *has disjoint users* $u$ *and* $v$ *then* $u \not\rightsquigarrow v : \mathsf{S}$ *is equivalent to* $\mathsf{S}\ \mathbf{sec}_v\ \mathcal{NI}$*, where* $\mathcal{NI}$ *is defined by:*

$$\forall\, t \in \tau\mathsf{S} \bullet (t\ \mathcal{NI}\ t|_u) \wedge (t|_u\ \mathcal{NI}\ t).$$

**Proof** $u \not\rightsquigarrow v : \mathsf{S}$

$\Leftrightarrow \forall\, t \in \tau\mathsf{S} \bullet (t \approx_v^{\mathsf{S}} t|_u)$ [Definition 3.9]

$\Leftrightarrow \forall\, t \in \tau\mathsf{S} \bullet (t \approx_v^{\mathsf{S}} t|_u) \wedge (t|_u \approx_v^{\mathsf{S}} t)$ [$\approx_v^{\mathsf{S}}$ symmetric]

$\Leftrightarrow \mathcal{NI} \subseteq \approx_v^{\mathsf{S}}$

$\Leftrightarrow \mathsf{S}\ \mathbf{sec}_v\ \mathcal{NI}$. [Definition B.2] □

We define an order on security policies by considering the set of policies on a particular alphabet and define when one policy 'distinguishes more' traces than another.

**Definition B.3** *If* $\mathcal{P}$ *and* $\mathcal{Q}$ *are policies on* $A$*, we say* $\mathcal{P}$ distinguishes no more traces *than* $\mathcal{Q}$ *when* $\mathcal{P} \subseteq \mathcal{Q}$ *and write* $\mathcal{P} \preceq_A \mathcal{Q}$*.* △

Using the order $\preceq_A$ we see that a system satisfies a security policy if the system does not distinguish more traces than the policy allows. $\approx_v^{\mathsf{S}}$ is itself a security policy and we can think of our satisfaction relation, $\mathbf{sec}_v$, in terms of the order $\preceq_A$.

**Lemma B.2** *In system* $\mathsf{S}$*, with security policy* $\mathcal{P}$ *on* $\alpha\mathsf{S}$ *and user* $v$*,* $\approx_v^{\mathsf{S}}$ *is a security policy on* $\alpha\mathsf{S}$*, and* $\mathsf{S}\ \mathbf{sec}_v\ \mathcal{P}$ *iff* $\mathcal{P} \preceq_{\alpha\mathsf{S}} \approx_v^{\mathsf{S}}$*.*

**Proof** $\approx_v^{\mathsf{S}}$ is an equivalence relation (Lemma 3.2) and is therefore symmetric. It is a policy on $\alpha\mathsf{S}$ since it is defined on the traces of $\mathsf{S}$.

$$\mathsf{S}\ \mathbf{sec}_v\ \mathcal{P} \;\Leftrightarrow\; \mathcal{P} \subseteq \approx_v^{\mathsf{S}} \Leftrightarrow \mathcal{P} \preceq_{\alpha\mathsf{S}} \approx_v^{\mathsf{S}}.$$ [Definitions B.2 and B.3] □

As the order $\preceq_A$ is the subset ordering on security policies $A$ forms a complete lattice under $\preceq_A$. We define suprema and infima in the usual way and identify the top and bottom elements.

**Lemma B.3** *$A$ forms a complete lattice under $\preceq_A$ with top $\top_A = A^* \times A^*$ and bottom $\bot_A = \emptyset$. The supremum and infimum of chains of policies correspond to union and intersection.*

**Proof** $A$ forms a lattice under $\preceq_A$ since $\preceq_A$ is just $\subseteq$ on policies. $\top_A$ is the top policy since it is the largest relation on $A^*$ under $\subseteq$ and it is symmetric; $\bot_A$ is the smallest symmetric relation on $A^*$.

The supremum and infimum of chains of policies are policies since they are relations on $A^*$ and symmetry is preserved by union and intersection. □

In Section 4.2.3 we use the symbol $\preceq$ to refer to an ordering on systems; we write $\mathsf{S} \preceq_v \mathsf{T}$ when $\mathsf{T}$ is a secure replacement of $\mathsf{S}$. That ordering corresponds to the ordering $\preceq_A$ on security policies, as the equivalence relations $\approx_v^{\mathsf{S}}$ and $\approx_v^{\mathsf{T}}$ are policies on $\alpha\mathsf{S}$ (Lemma B.2).

Lemma B.4 shows the relationship between secure replacement and satisfaction of security policies: if system $\mathsf{T}$ is a secure replacement of $\mathsf{S}$ and $\mathsf{S}$ satisfies $\mathcal{P}$ then $\mathsf{T}$ satisfies $\mathcal{P}$.

**Lemma B.4** *If $\mathsf{S}\,\mathbf{sec}_v\,\mathcal{P}$ and $\mathsf{S} \preceq_v \mathsf{T}$ then $\mathsf{T}\,\mathbf{sec}_v\,\mathcal{P}$.*

**Proof**

$$
\begin{array}{lll}
 & (\mathsf{S}\,\mathbf{sec}_v\,\mathcal{P}) \wedge (\mathsf{S} \preceq_v \mathsf{T}) & \\
\Leftrightarrow & (\mathcal{P} \subseteq \approx_v^{\mathsf{S}}) \wedge (\approx_v^{\mathsf{S}} \subseteq \approx_v^{\mathsf{T}}) & \text{[Definitions B.2 and 4.1]} \\
\Rightarrow & \mathcal{P} \subseteq \approx_v^{\mathsf{T}} & [\subseteq \text{ transitive}] \\
\Leftrightarrow & \mathsf{T}\,\mathbf{sec}_v\,\mathcal{P}. & \text{[Definition B.2]}
\end{array}
$$

□

In Appendix A secure replacement is related to downwards simulation and Theorems A.1 and A.2 give conditions under which a downwards simulation is a secure replacement. In that appendix non–interference is used as the security policy and an example is worked through (see Section A.3).

Lemma B.4 shows that secure replacement guarantees that if a system satisfies a security policy (in general) then a secure replacement of it satisfies the same security policy. Hence Theorems A.1 and A.2 apply to general security policies, not just non–interference.

The following section gives laws for general security policies.

# B.3 Laws of General Policies

In Section 4.3.3 a collection of laws for non–interference is given. Here we give a similar collection which shows how systems may be constructed using the algebra of CSP, and how the constructs of CSP relate to security policies.

Lemma B.5 highlights some of the basic properties of the relation $\mathbf{sec}_u$ . The proofs of those properties are simple consequences of Definitions B.1, B.2, B.3 and Lemmas 3.3 and B.3.

**Lemma B.5** *For any policies $\mathcal{P}$, $\mathcal{Q}$ on $A$, system $\mathsf{S}$ and users $u, v \subseteq A$:*

- $\mathsf{S}\ \mathbf{sec}_u\ \bot_A$ ; $\mathsf{S}\ \mathbf{sec}_\emptyset\ \mathcal{P}$;
- $\mathtt{Run}\ _A\ \mathbf{sec}_u\ \mathcal{P}$;$\mathtt{Chaos}\ _A\ \mathbf{sec}_u\ \mathcal{P}$;$\mathtt{Stop}\ _A\ \mathbf{sec}_u\ \mathcal{P}$;
- $\mathsf{S}\ \mathbf{sec}_u\ \mathcal{P} \wedge \mathsf{S}\ \mathbf{sec}_u\ \mathcal{Q} \Rightarrow \mathsf{S}\ \mathbf{sec}_u\ (\mathcal{P} \cap \mathcal{Q}) \wedge \mathsf{S}\ \mathbf{sec}_u\ (\mathcal{P} \cup \mathcal{Q})$;
- $\mathsf{S}\ \mathbf{sec}_u\ \mathcal{P} \wedge \mathcal{Q} \preceq_A \mathcal{P} \Rightarrow \mathsf{S}\ \mathbf{sec}_u\ \mathcal{Q}$;
- $\mathsf{S}\ \mathbf{sec}_u\ \mathcal{P} \wedge v \subseteq u \Rightarrow \mathsf{S}\ \mathbf{sec}_v\ \mathcal{P}$.

□

### Laws concerning Prefixing

The prefix version of a security policy given in Definition B.4 is used to give the policy that a system defined using prefixing satisfies.

**Definition B.4** *If $\mathcal{P}_b$ are policies on alphabet $A$ for $b \in B$ ($B \subseteq A$) then*

$$\begin{aligned}\mathcal{P}^B \ \triangleq\ & \{(\langle b\rangle^\frown s, \langle b\rangle^\frown t) \mid b \in B \wedge (s,t) \in \mathcal{P}_b\} \\ & \cup \\ & \{(\langle\rangle, \langle\rangle)\}.\end{aligned}$$

△

**Law B.1** (*choice*)

$$\frac{(2.1.1)\ \ \forall\, b \in B \bullet \mathsf{S}_b\ \mathbf{sec}_u\ \mathcal{P}_b}{(b : B \rightarrow \mathsf{S}_b)\ \mathbf{sec}_u\ \mathcal{P}^B}$$

**Proof** Suppose that $\alpha\mathsf{S} = A$ and consider $(s,t) \in \mathcal{P}^B$. If $s = t = \langle\rangle$ then clearly $(s,t) \in \approx_u^{(b:B\rightarrow \mathsf{S}_b)}$ ($u$–equivalence is reflexive: Lemma 3.2).

Hence suppose $s$ and $t$ are not empty, so $s = \langle b \rangle ^\frown s'$ and $t = \langle b \rangle ^\frown t'$ where $b \in B$ and $(s', t') \in \mathcal{P}_b$ (see Definition B.4):

$$\begin{array}{rcll} (b : B \rightarrow \mathsf{S})/s & = & \mathsf{S}/s' & [\text{L2:53; L3:53}] \\ & \equiv_u^A & \mathsf{S}/t' & [(s', t') \in \mathcal{P}_b;\ \mathsf{S}_b\ \mathbf{sec}_u\ \mathcal{P}_b] \\ & = & (b : B \rightarrow \mathsf{S})/t. & [\text{L2:53; L3:53}] \end{array}$$

Hence $s \approx_u^{(b:B \rightarrow \mathsf{S}_b)} t$ and $\mathcal{P}^B \subseteq \approx_u^{(b:B \rightarrow \mathsf{S}_b)}$. By Definition B.2 we have $(b : B \rightarrow \mathsf{S}_b)\ \mathbf{sec}_u\ \mathcal{P}^B$. □

**Laws concerning Change of Alphabet**

Changing the alphabet of a policy simply requires the change of alphabet function to be mapped across all traces in the policy.

**Definition B.5** *If $\mathcal{P}$ is a policy on alphabet $A$ and $f : A \rightarrow B$ then*

$$f(\mathcal{P}) \triangleq \{(f^*(s), f^*(t)) \mid s, t \in A^* \wedge (s, t) \in \mathcal{P}\}. \qquad \triangle$$

Changing the alphabet of a process changes the alphabet of the security policy and the interfaces of users; Law B.2 shows how.

**Law B.2** (*change alphabet*)

$$\frac{(2.2.1)\ \mathsf{S}\ \mathbf{sec}_u\ \mathcal{P}}{f(\mathsf{S})\ \mathbf{sec}_{f(u)}\ f(\mathcal{P})}$$

**Proof** The proof follows from Lemma 4.14 in a similar manner to the proof of Law 4.4. □

Hiding part of the alphabet of a policy is achieved by hiding events in the traces used to define a policy.

**Definition B.6** *If $\mathcal{P}$ is a policy on alphabet $A$ and $C \subseteq A$ then*

$$\mathcal{P} \setminus C \triangleq \{(s|_C, t|_C) \mid (s, t) \in \mathcal{P}\}. \qquad \triangle$$

A policy $\mathcal{P}$ is $C$–paired if for all pairs of traces $(s, t)$ such that $(s|_C, t|_C)$ is in $(\mathcal{P} \setminus C)$ then $(s, t) \in \mathcal{P}$.

That strengthens the definition of $(\mathcal{P} \setminus C)$ by requiring that not only are all $(s|_C, t|_C) \in (\mathcal{P} \setminus C)$ for $(s, t) \in \mathcal{P}$, but that the opposite is also true: all such $(s, t)$ must be in $\mathcal{P}$.

**Definition B.7** *Policy* $\mathcal{P}$ *on alphabet* $A$ *is* $C$–paired *on* $\mathsf{S}$ *if*

$$\forall\, s, t \in \tau\mathsf{S} \bullet (s|_C, t|_C) \in (\mathcal{P} \setminus C) \Rightarrow (s, t) \in \mathcal{P}. \qquad \triangle$$

Alphabet hiding can be applied to a system, hiding events from the alphabet of a user and changing the security policy as appropriate.

**Law B.3** (*hiding*)

$$\frac{\begin{array}{ll} (2.3.1) & \mathsf{S}\ \mathbf{sec}_u\ \mathcal{P} \\ (2.3.2) & C \subseteq u \\ (2.3.3) & \mathcal{P}\ \textit{is}\ C\textit{–paired} \end{array}}{(\mathsf{S} \setminus C)\ \mathbf{sec}_{(u \setminus C)}\ (\mathcal{P} \setminus C)}$$

**Proof** From Lemma 4.15 in a similar manner to Law 4.5. □

**Laws concerning Composition of Systems**

The policy $\mathcal{P} \mid \mathcal{Q}$ is used in the parallel combination of systems. When $\mathcal{P}$ and $\mathcal{Q}$ are both defined on $A$ it is simply $\mathcal{P} \cap \mathcal{Q}$.

**Definition B.8** *If* $\mathcal{P}$ *and* $\mathcal{Q}$ *are policies on* $A$ *and* $B$*, respectively, then*

$$(\mathcal{P} \mid \mathcal{Q}) \triangleq \{(s, t) \mid (s \upharpoonright A, t \upharpoonright A) \in \mathcal{P} \wedge (s \upharpoonright B, t \upharpoonright B) \in \mathcal{Q}\} \cap (A \cup B)^*. \quad \triangle$$

Law B.4 shows that the parallel combination of two processes satisfies the combination of their security policies given by Definition B.8. Lemma B.6 shows the relationship between $\approx^{\mathsf{S}}_u$, $\approx^{\mathsf{T}}_u$ and $\approx^{(\mathsf{S}\|\mathsf{T})}{}_u$; cf. [L1:90].

**Lemma B.6** *If systems* $\mathsf{S}$ *and* $\mathsf{T}$ *have user* $u$ *then for* $s, t \in \alpha(\mathsf{S}\|\mathsf{T})^*$ *if* $(s \upharpoonright \alpha\mathsf{S} \approx^{\mathsf{S}}_u t \upharpoonright \alpha\mathsf{S}) \wedge (s \upharpoonright \alpha\mathsf{T} \approx^{\mathsf{T}}_u t \upharpoonright \alpha\mathsf{T})$ *then* $s \approx^{\mathsf{S}\|\mathsf{T}}_u t$.

**Proof** Suppose that $\alpha\mathsf{S} = A$, $\alpha\mathsf{T} = B$ and $s, t \in \tau(\mathsf{S}\|\mathsf{T})$ such that $(s \upharpoonright A \approx^{\mathsf{S}}_u t \upharpoonright A) \wedge (s \upharpoonright B \approx^{\mathsf{T}}_u t \upharpoonright B)$ (the case $s, t \notin \tau(\mathsf{S}\|\mathsf{T})$ is trivial—both $(\mathsf{S}\|\mathsf{T})/s$ and $(\mathsf{S}\|\mathsf{T})/t$ are undefined).

$$\begin{array}{lll} & ((\mathsf{S}\|\mathsf{T})/s)\|\overline{u}_{(A \cup B)} & \\ = & (\mathsf{S}/s \upharpoonright A)\|(\mathsf{T}/s \upharpoonright B)\|\overline{u}_{(A \cup B)} & \text{[L2:72]} \\ = & (\mathsf{S}/s \upharpoonright A)\|(\mathsf{T}/s \upharpoonright B)\|\overline{u}_{(A \cup B)}\|\overline{u}_A\|\overline{u}_B & \text{[Lemma 4.17]} \\ = & (\mathsf{S}/s \upharpoonright A\|\overline{u}_A)\|(\mathsf{T}/s \upharpoonright B\|\overline{u}_B)\|\overline{u}_{(A \cup B)} & \text{[L1:70; L2:70]} \\ = & (\mathsf{S}/t \upharpoonright A\|\overline{u}_A)\|(\mathsf{T}/t \upharpoonright B\|\overline{u}_B)\|\overline{u}_{(A \cup B)} & \text{[assumption]} \end{array}$$

$$\begin{aligned} &= (\mathsf{S}/t\restriction A)\|(\mathsf{T}/t\restriction B)\|\overline{u}_{(A\cup B)}\|\overline{u}_A\|\overline{u}_B && \text{[L1:70; L2:70]}\\ &= (\mathsf{S}/t\restriction A)\|(\mathsf{T}/t\restriction B)\|\overline{u}_{(A\cup B)} && \text{[Lemma 4.17]}\\ &= ((\mathsf{S}\|\mathsf{T})/t)\|\overline{u}_{(A\cup B)}. && \text{[L2:72]} \end{aligned}$$

Hence $(\mathsf{S}\|\mathsf{T})/s \equiv_u^{(A\cup B)} (\mathsf{S}\|\mathsf{T})/t$ and $s \approx_u^{\mathsf{S}\|\mathsf{T}} t$ by Definition 3.7. □

**Law B.4** (*parallel*)

$$\frac{\begin{array}{l}(2.4.1)\ \mathsf{S}\ \mathbf{sec}_u\ \mathcal{P}\\ (2.4.2)\ \mathsf{T}\ \mathbf{sec}_u\ \mathcal{Q}\end{array}}{(\mathsf{S}\|\mathsf{T})\ \mathbf{sec}_u\ (\mathcal{P}\mid\mathcal{Q})}$$

**Proof** $\mathsf{S}\ \mathbf{sec}_u\ \mathcal{P} \wedge \mathsf{T}\ \mathbf{sec}_u\ \mathcal{Q}$

$$\begin{aligned} &\Leftrightarrow \mathcal{P} \subseteq \approx_u^{\mathsf{S}} \wedge \mathcal{Q} \subseteq \approx_u^{\mathsf{T}} && \text{[Definition B.2]}\\ &\Rightarrow (\mathcal{P}\mid\mathcal{Q}) \subseteq (\approx_u^{\mathsf{S}}|\approx_u^{\mathsf{T}}) && \text{[Definition B.8]}\\ &\Rightarrow (\mathcal{P}\mid\mathcal{Q}) \subseteq \approx_u^{\mathsf{S}\|\mathsf{T}} && \text{[Lemma B.6]}\\ &\Leftrightarrow (\mathsf{S}\|\mathsf{T})\ \mathbf{sec}_u\ (\mathcal{P}\mid\mathcal{Q}). && \text{[Definition B.2]} \end{aligned}$$ □

With no further information about the processes S and T the security policy which their external choice composition satisfies is the union of policies they satisfy, restricted to their disjoint traces; and the intersection of the policies. The same applies for internal choice. The policy $\mathcal{P}\ {}^{\mathsf{S}}\!\uplus^{\mathsf{T}}\ \mathcal{Q}$ of Definition B.9 is the one required.

$\mathcal{P}\ {}^{\mathsf{S}}\!\uplus^{\mathsf{T}}\ \mathcal{Q}$ is defined for policy $\mathcal{P}$ on system S and policy $\mathcal{Q}$ on system T. It consists of the policy $\mathcal{P}$ with traces from T removed, $\mathcal{Q}$ with traces from S removed and the intersection of the policies which both systems agree on. ($\diamondminus$ and $\diamond$ are given in Definition 2.1.)

**Definition B.9** *If $\mathcal{P}$ is defined for system* S *and $\mathcal{Q}$ for system* T *then*

$$\mathcal{P}\ {}^{\mathsf{S}}\!\uplus^{\mathsf{T}}\ \mathcal{Q} \triangleq (\mathcal{P} \diamondminus \tau\mathsf{T}) \cup (\mathcal{Q} \diamondminus \tau\mathsf{S}) \cup ((\mathcal{P}\cap\mathcal{Q}) \diamond (\tau\mathsf{S}\cap\tau\mathsf{T})).$$ △

**Law B.5** (*external choice; internal choice*)

$$\frac{\begin{array}{l}(2.5.1)\ \mathsf{S}\ \mathbf{sec}_u\ \mathcal{P}\\ (2.5.2)\ \mathsf{T}\ \mathbf{sec}_u\ \mathcal{Q}\end{array}}{\begin{array}{l}(\mathsf{S}\,[\!]\,\mathsf{T})\ \mathbf{sec}_u\ (\mathcal{P}\ {}^{\mathsf{S}}\!\uplus^{\mathsf{T}}\ \mathcal{Q}) \diamond \{\langle\rangle\}\\ (\mathsf{S}\sqcap\mathsf{T})\ \mathbf{sec}_u\ (\mathcal{P}\ {}^{\mathsf{S}}\!\uplus^{\mathsf{T}}\ \mathcal{Q})\end{array}}$$

**Proof** Suppose that $\alpha\mathsf{S} = \alpha\mathsf{T} = A$ and take any $s$ and $t$ such that

$$s\,(\mathcal{P}\ {}^{\mathsf{S}}\!\uplus^{\mathsf{T}}\ \mathcal{Q}) \diamond \{\langle\rangle\}\ t.$$

Examining the case $s\,(\mathcal{P} \diamondsuit \tau\mathsf{T})\,t$ (the case $s\,(\mathcal{Q} \diamondsuit \tau\mathsf{S})\,t$ follows by symmetry) we find that $s \in \tau\mathsf{S}$ iff $t \in \tau\mathsf{S}$ (since $\mathsf{S}\,\mathbf{sec}_u\,\mathcal{P}$). When $s, t \notin \tau\mathsf{S}$ $(\mathsf{S}\|\mathsf{T})/s$ and $(\mathsf{S}\|\mathsf{T})/t$ are undefined; when $s, t \in \tau\mathsf{S}$:

$$\begin{array}{rcll} (\mathsf{S}\|\mathsf{T})/s & = & \mathsf{S}/s & \text{[L2:108]} \\ & \equiv_u^A & \mathsf{S}/t & [s\,\mathcal{P}\,t \text{ and } \mathsf{S}\,\mathbf{sec}_u\,\mathcal{P}] \\ & = & (\mathsf{S}\|\mathsf{T})/t. & \text{[L2:108]} \end{array}$$

When $s\,((\mathcal{P} \cap \mathcal{Q}) \diamond \tau\mathsf{S} \cap \tau\mathsf{T})\,t$,then $s \neq \langle\rangle$ and $t \neq \langle\rangle$ and $s, t \in \tau\mathsf{S} \cap \tau\mathsf{T}$.

$$\begin{array}{rcll} (\mathsf{S}\|\mathsf{T})/s & = & (\mathsf{S}/s) \sqcap (\mathsf{T}/s) & \text{[L2:108]} \\ & \equiv_u^A & (\mathsf{S}/t) \sqcap (\mathsf{T}/t) & \text{[Lemma 4.13, } \mathsf{S}\,\mathbf{sec}_u\,\mathcal{P} \text{ and } \mathsf{T}\,\mathbf{sec}_u\,\mathcal{Q}] \\ & = & (\mathsf{S}\|\mathsf{T})/t. & \text{[L2:108]} \end{array}$$

So $s \approx_u^{(\mathsf{S}\|\mathsf{T})} t$ and by Definitions B.2 and B.9 $(\mathsf{S}\|\mathsf{T})\,\mathbf{sec}_u\,(\mathcal{P}\ {}^{\mathsf{S}}\!\uplus^{\mathsf{T}}\ \mathcal{Q})$.

The proof for $\sqcap$ is similar, replacing [L2:108] by [L2:106] and changing $(\mathcal{P}\ {}^{\mathsf{S}}\!\uplus^{\mathsf{T}}\ \mathcal{Q}) \diamond \{\langle\rangle\}$ to $(\mathcal{P}\ {}^{\mathsf{S}}\!\uplus^{\mathsf{T}}\ \mathcal{Q})$. □

**Law concerning Recursive Systems**

The recursion law is similar to [L6:62]. If when a process (marker) $\mathsf{X}$ satisfies the policy $\mathcal{P}$ the process $\mathsf{F}(\mathsf{X})$ satisfies $\mathcal{P}$ then the fixed–point of $\mathsf{F}$ satisfies $\mathcal{P}$ (the fixed–point exists when $\mathsf{F}$ is guarded: see [39, pages 29,98–99]). Note that we do not need to verify the base condition as Lemma B.5 shows that Chaos satisfies any security policy.

**Law B.6** (*recursion*)

$$\frac{(2.6.1)\ \mathsf{X}\,\mathbf{sec}_u\,\mathcal{P} \Rightarrow \mathsf{F}(\mathsf{X})\,\mathbf{sec}_u\,\mathcal{P}}{\mu\,\mathsf{X} \bullet \mathsf{F}(\mathsf{X})\,\mathbf{sec}_u\,\mathcal{P}}$$

**Proof** The antecedent and Lemma B.5 guarantee that $\mathsf{F}^n(\mathrm{Chaos}_A)$ satisfies $\mathcal{P}$ for all $n$ by induction. Put $\mathsf{Z} = \bigsqcup_{n \geq 0} \mathsf{F}^n(\mathrm{Chaos}_A)$.

$$\begin{array}{rll} & \forall\, n \in \mathrm{N} \bullet \mathsf{F}^n(\mathrm{Chaos}_A)\,\mathbf{sec}_u\,\mathcal{P} & \\ \Leftrightarrow & \forall\, n \in \mathrm{N} \bullet \mathcal{P} \subseteq \approx_u^{\mathsf{F}^n(\mathrm{Chaos}_A)} & \text{[Definition B.2]} \\ \Rightarrow & \mathcal{P} \subseteq \bigcap_{n \geq 0} \approx_u^{\mathsf{F}^n(\mathrm{Chaos}_A)} & \text{[Lemma B.3]} \\ \Rightarrow & \mathcal{P} \subseteq \approx_u^{\mathsf{Z}} & \text{[Lemma 4.19]} \\ \Leftrightarrow & \mathcal{P} \subseteq \approx_u^{\mu\,\mathsf{X} \bullet \mathsf{F}(\mathsf{X})} & \text{[D17:132]} \\ \Leftrightarrow & \mu\,\mathsf{X} \bullet \mathsf{F}(\mathsf{X})\,\mathbf{sec}_u\,\mathcal{P}. & \text{[Definition B.2]} \end{array}$$

□

# Appendix C

# Glossary

References to definitions within this glossary are indicated by the use of italic type.

***–property** (See also *confinement*) One of the notions of the *Bell & LaPadula* model: the restrictions *no read–up* and *no write–down*; often used to mean no write–down alone.

**access class** A *level* in a *multi–level secure* system.

**access control** A protection mechanism which controls *subjects'* interaction with system *objects*, by way of an *access–control list*, *access matrix* or *capability list*. Most systems use both *discretionary* and *mandatory access control*. See [27, Chapter 6] and [95].

**access–control list** A row of an *access matrix*.

**access matrix** A matrix whose rows represent *subjects* and columns represent *objects*. Entries at a point represent the current access permission for that subject and object. Typical permissions are $r$ (read), $w$ (write), $a$ (append) or $x$ (execute).

**aggregation** Aggregation of data at a particular *level* may yield information that should be classified at a higher level. See [24].

**attribute polyinstantiation** A form of *polyinstantiation* in which multiple copies (for different *classifications*) of individual elements (attributes) of a database are maintained.

**authentication** The process of a *subject* convincing itself of the correct identity of another subject.

**availability** The property of prevention of unauthorized withholding of information (called denial of service in [59]).

**bandwidth** The number of bits per second (per unit time, per symbol) that can be transmitted across a *channel*.

**basic security theorem** The theorem of the *Bell & LaPadula* model which is used to show, by induction on sequences of operations, that a system is secure if each state satisfies the three properties: **–property*; *simple security* and *discretionary security*. See also *unwinding*.

**Bell & LaPadula Model** A model of an instance of *multi–level security* (in particular the security of MULTICS) which introduced a number of familiar terms: see **–property*, *simple security*, *discretionary security* and *tranquillity*. See [8, 7, 9, 10].

**Biba Integrity Model** A mathematical model of *integrity* described in [11] where *subjects* and *objects* are allocated an integrity class (an analogue of *level* in *multi–level security*). The model is analogous to *MLS* having two requirements: that subjects may not read from objects of a lower integrity class; and that subjects may not write to objects having a higher integrity class; cf. *no read–up* and *no write–down* for *confidentiality*.

**capability** An entry in a *capability list* giving the permission a *subject* has for an *object*.

**capability list** A column of an *access matrix*.

**channel** Any path along which information may flow. See also *legitimate channel* and *covert channel*. See [59].

**chinese wall** A security *policy* that is used in commerce to prevent conflicts of interest. Once a *subject* has information about a company he must not be able to gain information about other companies that would cause a conflict of interest. This is usually achieved by dividing the sources of information into conflict sets and only allowing information to flow from a single source in each set (see [12, 19, 24]).

**cipher text** A message after *encryption*. The cipher text is intended to be incomprehensible and provide no information about the *plain text* from which it came (except to an intended recipient).

**classification** The *level* of an *object* in a *multi–level secure* system.

**clearance** The *level* of a *subject* in a *multi–level secure* system.

**colour** An *access class* (used in [83]).

**composability** Defined by McCullough in [63, 64]. A property is composable if two computer systems exhibiting that property, when operating in parallel—in a sense defined by McCullough concerning the connection of inputs and outputs—still exhibit that property.

**conditional non–interference** Three forms of *non–interference* defined in [28, 29]. They are: **unless** (this form is used to allow communication between *subjects* on particular channels); **given** (the normal purge function is replaced by one supplied by the system designer to achieve restricted communication between subjects); and **if** (subjects can communicate if some condition is met; this form is used to specify *discretionary access control* using non–interference). See Section 7.4.1.

**confidentiality** The property of prevention of unauthorized disclosure of information (see [59, page 4]).

**confinement** A process (or program) is confined if it cannot communicate with any process other than one with which it is authorized to communicate (see [58]).

**cover story** A cover story is data provided to a user (usually of low *clearance*) of a database in order to hide data of a higher *classification.*

**covert channel** A *channel* outside the channels specified for use in a system (see [58, 60]).

**cryptography** Literally 'hidden writing', cryptography is the science of causing a message to be unintelligible to anyone other than intended recipients. See *encryption* and *decryption.* See [22, 97].

**decryption** The process of changing a *cipher text* to the original *plain text* using the appropriate algorithm and *key.*

**denial of service** See *availability.*

**discretionary access control** A form of *access control* in which the *access class* a *subject* has for an *object* may change during the operation of the system. See [27, pages 57 ff.].

**discretionary security** A restriction in the *Bell & LaPadula Model* on changes in *classification* of *objects* so that the current access cannot exceed maxima set in the *access* permission *matrix*.

**domain** The mode of operation of a system. Frequently an operating system's state is divided into a number of domains. Examples are privileged and unprivileged; in privileged mode the system has access to all *objects* and performs the low–level processing tasks required (including the enforcement of security) and in unprivileged mode (the mode usually used by application programs) access to objects (such as certain areas of memory) is restricted. See [95]

**encryption** The conversion of *plain text* to *cipher text* in order to hide the information in the plain text. To encrypt requires the relevant algorithm and *key*.

**entity integrity** A database has entity integrity if each row of that database is unique, i.e. for each *key* value there is at most one row.

**equivalent** A number of security models are based on the idea of equivalent states (see, for example, [28, 35, 63]). Two states are equivalent (at security *level* $l$) if no user at that level can distinguish them. The definition of equivalence is usually made in one of two ways: by examination of the structure of the state (of the *objects* visible at $l$); or by examination of the outputs a system generates at level $l$.

Another approach is based purely on examination of the interactions at interfaces (see Chapter 3).

**formal top–level specification** (FTLS) The specification of safety and security properties of a system given formally and against which all development is checked. See [76].

**generalized non–interference** This term is introduced in [64] (and see Section 7.4.3); it refers to a form of *non–interference* described by saying that user $h$ does not interfere with $l$ if for any trace $t$ of the system and operation $ho$ belonging to $h$ the future history (from $l$'s viewpoint) of both $t$ and $t^\frown\langle ho\rangle$ are identical.

**high–water mark** The high–water mark of a process is the supremum of the *levels* of all *objects* opened for reading or writing by that process; it never decreases. See [22, page 287].

**hook–up** See *composability* and *restrictiveness*.

**inference** Jacob, in [44, 47], describes an 'inference function' which gives the possible behaviours of a system based on a *subject's* view at some interface. That behaviour is inferred from knowledge of the construction of the system. See Section 7.3.2.

Inference control occurs in database systems, where secret information may be deduced from collecting authorized (often de–classified) data and it is necessary to prevent this type of leakage of classified information.

**information flow** Information flows in a system when one *subject* is able to detect changes caused by (or the activity of) another subject. See Section 7.3 for a summary of work concerning information flow.

**integrity** Prevention of unauthorized corruption of information (see *Biba Integrity Model*, *entity integrity* and [59]).

**kernel** A process used to control all aspects of system operation. All operations which affect *objects* are made through the kernel. It provides all the security features and so is *trusted*. See [27, Chapter 10].

**key** A token provided to an *encryption* or *decryption* algorithm to effect the change from *plain text* to *cipher text* or vice versa. The value of the key is generally secret and is used to protect the encrypted information.

Also a database term where it refers to the value in any row on which the data can be searched or sorted.

**label** A security *level* associated with a *subject* or an *object*.

**legitimate channel** A *channel* defined in the security policy.

Also, used by Lampson ([58]) to refer to a valid channel that is used to subvert a security *policy* by adding secret information to the legitimate messages; examples are of messages transmitted by modulating message length, word–frequency or text justification.

**level** (security level; sensitivity level) A security level is a value given to a *subject* or *object* which is used to define its security properties. These levels are usually arranged in an order or lattice structure and a security policy may be stated by reference to that order. An example is *multi–level security* where communication is restricted

from high levels to low levels. See *clearance* and *classification*. See [59].

**mandatory access control** *Access control* where the access a *subject* has to an *object* is defined before operation of the system and only changes within fixed limits. See **–property* and *simple security*. See [27, pages 61 ff.].

**multi–level security** (MLS) The (military) scheme of assigning a *level* to *subjects* and *objects* and asserting that no communication may occur from a high level to a lower level. See [59].

**non–deducibility** The term used to describe the security policy implied by Sutherland in [94]. One user cannot deduce information about another if he is unable to calculate the interaction at that user's interface from his knowledge of the system and his own interactions. See Section 7.3.3.

**non–interference** The term introduced in [28] to refer to a security policy that guarantees no communication from one user to another. This intransitive policy is based on the statement that no communication occurs if the presence or absence of operations (events) performed by a *subject* does not affect the state observed by another subject (i.e. the two states are *equivalent*). See Section 7.4.

**no read–up** A system satisfies the requirement no read–up if no user can read information with a higher *classification* than his own *clearance*.

**no write–down** A system satisfies the no write–down requirement if no user can write information with a lower *classification* than his own *clearance*.

**object** Any entity that forms part of a system. Examples are files, directories, programs, memory, printers.

**orange book** Slang term for the *TCSEC*.

**perfect non–interference** *Non–interference* was first presented with a state–machine model which did not encompass any consideration of time (or probability etc.) and thus a system that was non–interfering might still have undesired communication along routes not examined in the model (see *covert channel*). In [33] Gray introduces the term perfect non–interference to refer to a policy which prevents communication by (almost) any method.

The approach is to generalize the definition of non–interference given in [28] to models which encompass computation with time, probability etc. See also [45].

**plain text** A message before encryption. In this form it is possible that someone other than the intended recipient is able to read it.

**policy** The requirements for the security features of a system. This specification may take the form of an English or mathematical description. See, for example, [76].

**polyinstantiation** The technique of holding multiple copies of information (in database systems with different data for different security *levels*) to add security features. The two principle versions are *tuple polyinstantiation* and *attribute polyinstantiation*. Polyinstantiation is used in database systems to provide *multi–level security* (see *SeaView*). See Section 7.2.

**probabilistic non–interference** A form of *non–interference* which is based on a model which uses probability, to capture communication by probabilistic analysis (see [33, 34]).

**reference monitor** A single separated mechanism through which all accesses to *objects* are controlled (see also *kernel*). See [27, pages 35–37].

**referential integrity** A database has referential integrity if cross–references to *keys* exist, i.e. if a database row refers to another then that second row must exist.

**restrictiveness** A form of *generalized non–interference* which is *composable* under a composition defined by McCullough in [64]. See Section 7.4.3.

**SeaView** The name of a project to produce an A1 (see *TCSEC*) multi–level database. That work produced the term *polyinstantiation*. See [62].

**secure isolation** A term introduced in [83] to describe a policy in which two *subjects* are unable to communicate with one another.

**security** An imprecise term used to cover aspects of system design and use encompassing: *access control*, *viruses*, passwords, physical protection, etc. Most commonly used in formal methods to mean *information–flow* policies, access control, covert channel analysis and *integrity*.

**separability** Same as *secure isolation*

**simple security** One of the notions of the *Bell & LaPadula Model* where it refers to the restriction that low *classification* users cannot observe those with a higher classification.

**storage channel** A *channel* that results from observable changes in *objects*. These changes include existence, and the contents, of such objects. See [58] and [98].

**subject** A user of a system. That user may be human or another system. Subjects are used (with *objects*) to define the security *policy*.

**TCSEC** (Trusted Computer Security Evaluation Criteria) A set of criteria developed by the US Department of Defense (see [76]) to enable users to evaluate the security of a system and to give a clear definition of the security features of system for use by vendors. It lays out six important requirements of a secure system:

1. There must be an explicit and well–defined security policy enforced by the system;
2. Access–control labels must be associated with objects;
3. Individual subjects must be clearly identified;
4. Audit information must be selectively kept and protected so that actions affecting security can be traced to the responsible party;
5. The [...] system must contain hardware/software mechanisms that can be independently evaluated to provide sufficient assurance that the system enforces 1 to 4 above;
6. The trusted mechanisms that enforce these basic requirements must be continuously protected against tampering and/or unauthorized changes.

In addition seven classes of system are identified and may be used to state the security properties of a system (note that C2 includes C1 and B3 contains B2 and B2 contains B1):

**D** (Minimal Protection) The system does not satisfy any of the other classes.

**C1** (Discretionary Security Protection) The system controls access between named subjects and objects, authenticates the identity of subjects and there is no obvious way of by–passing security.

**C2** (Controlled Access Protection) This level includes a requirement about the addition of auditing mechanisms and their protection.

**B1** (Labelled Security Protection) The system enforces mandatory access control and has labels assigned to subjects and objects. It has the *–property and simple security and has been thoroughly tested for security flaws.

**B2** (Structured Protection) A formal model is given as well as covert channel analysis. The system has a simple security–oriented architecture.

**B3** (Security Domains) The security features are enforced by a kernel.

**A1** (Verified Design) There is a formal top–level specification, formal covert channel analysis and informal code correspondence.

**timing channel** A *channel* resulting from changes in system performance or by timing some observable event. Communication occurs by reference to a real–time clock. See [58, 60] and [98].

**tranquillity** The restriction, that the *classification* of an *object* cannot be changed, which was introduced after criticism of the *Bell & LaPadula Model*. See [6].

**trap door** A trap door is a process which can be activated to circumvent the security features of a system.

Also a term in cryptology which refers to a problem (or function) that is infeasible, but becomes feasible given some 'trap door' information. See [97, pages 178–179].

**trojan horse** A process that appears to be legitimate (and may perform a legitimate task) and performs some covert operation using the rights and access permissions of the *subject* that executed it. See [59].

**trusted** *Subjects* and processes are trusted if they are responsible for the enforcement of security. See [76].

**trusted computing base** (TCB) The combination of hardware and software in a secure system used by untrusted subjects to effect security (see *kernel* and *reference monitor*).

**tuple polyinstantiation** A form of *polyinstantiation* in which multiple copies of tuples (database rows with the same *key* value) are held for different *classifications*.

**unwinding** A method of proof of *non–interference* (the term is introduced in [29]) which reduces a proof from one involving all possible histories (or traces) of a system to assertions about the

individual operations (events or transitions). The *basic security theorem* is a form of unwinding.

**user** A *subject*.

**virus** A virus is a process introduced into a system covertly which is able to replicate and move from system to system. It may contain code which is intended to cause a security problem. The majority of viruses affect system *integrity* by corrupting data, or cause *denial of service*. Viruses are a problem present in both single and multiuser systems. See [16].

# Index of Symbols

## CSP and associated operators


Stop ........ 8
Run ........ 8
Chaos ........ 8
Wait ........ 80
$\rightarrow$ ........ 8
$\|$ ........ 8
${}_X\|_Y$ ........ 81
$\backslash$ ........ 8
$|||$ ........ 8
$\sqcap$ ........ 8
$//$ ........ 8
$\mu\,\mathsf{X}$ ........ 8
$\mu\,\underline{\mathsf{X}}$ ........ 9

**head** ........ 9
**tail** ........ 9
**init** ........ 9
**last** ........ 9
**begin** ........ 81
$\#$ ........ 9
$\frown$ ........ 9
$\upharpoonright$ ........ 9
$\downarrow$ ........ 9
**set** ........ 10
$+$ ........ 80
$\dot{-}$ ........ 80
$\tilde{}$ ........ 81
$\wr$ ........ 81

$\tau\mathsf{S}$ ........ 10
$\alpha\mathsf{S}$ ........ 10

$\iota\mathsf{S}$ .......... 10
$\phi\mathsf{S}$ .......... 10
$\delta\mathsf{S}$ .......... 11
$\Sigma$ .......... 80
$T\Sigma^*_\leq$ .......... 80
$\sigma\mathsf{S}$ .......... 81
$\sqsubseteq$ .......... 11

**Equivalence of Traces and Systems**

$\approx^{\mathsf{S}}$ .......... 16
$\approx^{\mathsf{S}}_v$ .......... 18
$\approx^{\mathsf{S}}_v$ (timed systems) .......... 84

$\equiv^A_v$ .......... 18
$\overline{v}_A$ .......... 18

**Orderings**

$\preceq_v$ .......... 29
$\underline{\mathrm{C}}_v$ .......... 29
$\preceq$ .......... 30
$\underline{\mathrm{C}}$ .......... 30
$\succeq^{\mathcal{A}}$ .......... 111
$\succeq$ .......... 111
$\preceq_A$ .......... 140

**Non–interference**

$|_u$ .......... 21
$\not\rightsquigarrow$ .......... 22
**com** .......... 38
$\not\rightsquigarrow$ (transaction non–interference) .......... 38
$\not\rightsquigarrow$ (timed non–interference) .......... 84

**Security Policies**

$\mathcal{P}$ .......... 139
$\mathsf{S}\ \mathbf{sec}_u\ \mathcal{P}$ .......... 140
$\mathcal{P}^B$ .......... 142
$\mathcal{P} \setminus C$ .......... 143
$f(\mathcal{P})$ .......... 143
$\mathcal{P} \mid \mathcal{Q}$ .......... 144

$\mathsf{S}_{\uplus}\mathsf{T}$ . . . . . 145

**Miscellaneous**

**dom** . . . . . 5
**ran** . . . . . 5
**fst** . . . . . 5
**snd** . . . . . 5
$\mathbf{id}_A$ . . . . . 6
$\diamond$ . . . . . 6
$\diamondminus$ . . . . . 6

$\top_A$ . . . . . 6
$\bot_A$ . . . . . 6
$\bigcup$ . . . . . 7
$\bigcap$ . . . . . 7

$\oplus$ . . . . . 8
$\mapsto$ . . . . . 7, 112

$[t]^{\mathsf{S}}$ . . . . . 7
$\mathcal{C}_v^{\mathsf{S}}$ . . . . . 31

$|$ . . . . . 45
**do** . . . . . 50

@ . . . . . 97
$:|$ . . . . . 102
$[\![]\!]$ . . . . . 102
$[\![]\!]_u$ . . . . . 102
$\approx_l$ . . . . . 109
$\overset{s}{\Rightarrow}\!\bullet$ . . . . . 109
$\rightarrow \|_{\mathsf{S}}$ . . . . . 104
$\vdash$ . . . . . 112

**pre** . . . . . 127
$[\![]\!]^{\mathsf{S}}$ . . . . . 128
$\rho\mathsf{S}$ . . . . . 128
$\pi_v^{\mathsf{A}}$ . . . . . 131
${}^{\mathsf{A}}\Theta_v^{\mathsf{C}}$ . . . . . 131