Insertions, omissions, errata and corrigenda
for
On Shared Systems
by
Jeremy Jacob

I would like to thank Prof. Brian Warboys and Dr. Jeffrey Sanders for pointing out the need for an extra paragraph, some minor insertions and omissions, and some errata.

Insert the following paragraph at the end of section 1.4 (p 8):

> The work reported here, on both the topics of refinement and security, is pitched at the semantic, or model, level. As such it is not of immediate use to industrial practitioners who would wish to apply its results. In addition a syntactic, or algebraic, level of this work is needed. Its form would be a collection of laws that record program text manipulations which produce new program texts that are (local, independent or secure) refinements of the original text. We do not address such algebras in this thesis.

At the end of definition 20 (p 33) add:

> *Note that lw t is not, in general, a total function; we will need to check when we come to apply it that we do so within its domain.*

Immediately following example 8 (p 51) add the comment:

> Examples 7 and 8 show that local and independent refinement are different relations.

In section 4.3.2 ff change all occurrences of "enforces restriction of information flow" to "restricts information flow".
In appendix B add the following definition immediately following that of recursion (p106):

> The after operator:

P/t	$t \in \tau P$	αP	$\{(s,r) \mid (t \hat{} s, r) \in \phi P\}$	$\{d \mid t \hat{} d \in \delta P\}$

On Shared Systems

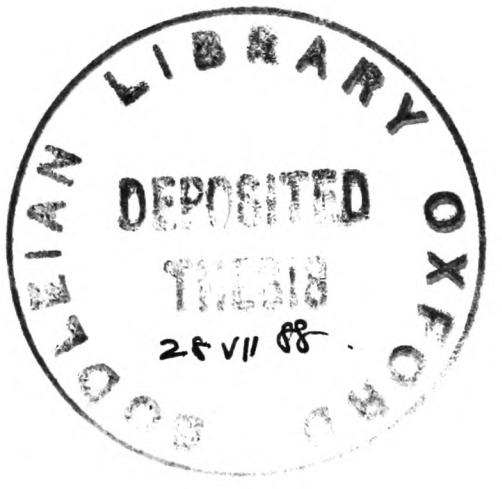
appendix B omit the clause (p 106):

where

$$strip_c(c.e) \triangleq e$$

age	Lines	Erratum	Corrigendum
10	14	$b\,\mathbf{r}^{-1}\,a \iff a\,\mathbf{r}\,a$	$b\,\mathbf{r}^{-1}\,a \iff a\,\mathbf{r}\,b$
15	8	in in	in
16	20	$s\,F\,(r \cup \{a\}) \vee (s\,\hat{}\,\langle a\rangle)\,F\,\{\}$	$t\,F\,(r \cup \{a\}) \vee (t\,\hat{}\,\langle a\rangle)\,F\,\{\}$
16	22	to engage	the capability to engage
16	23	to refuse	the capability to refuse
18	16	for for	for
19	19	$\mathbf{fi}_e$	**fi**
23	22	$b1$	$a2$
30	20	$\iff$ [lemmas 2–4]	$\implies$ [definition 18]
34	15	δD	δP
39	2	order updates	order in which updates
45	9	is trivial	is a trivial
47	16	function that	partial function that
49	3	u **reg** u	u **reg** v
51	24	lemma 34 and example 6	example 8
61	30	As a user	A user
63	10	‖‖	‖
63	11	$\{\{a\},\{b\}\}$.	$\{\{a\},\{b\}\}$, where $a \neq b$.
72	26	‖‖	‖
74	16–18	*all* that... done so.	the most information an implementation can allow the clerk to gain is the order in which customers have used the service.
74	26	‖‖	‖
91	7	way a of	way of

On Shared Systems



Jeremy Jacob

D.Phil. Thesis, 1987

Abstract

On Shared Systems

A thesis by
Jeremy Jacob
of
The Queens' College, Oxford
submitted for
the degree of Doctor of Philosophy
in
Michaelmas Term, 1987

Most computing systems are shared between users of various kinds. This thesis treats such systems as mathematical objects, and investigates two of their properties: refinement and security. The first is the analysis of the conditions under which one shared system can be replaced by another, the second the determination of a measure of the information flow through a shared system.

Under the heading of refinement we show what it means for one shared system to be a suitable replacement for another, both in an environment of co-operating users and in an environment of independent users. Both refinement relations are investigated, and a large example is given to demonstrate the relation for cooperating users.

We show how to represent the security of a shared system as an 'inference function', and define several security properties in terms of such functions. A partial order is defined on systems, with the meaning 'at least as secure as'. We generalise inference functions to produce 'security specifications' which can be used to capture the desired degree of security in any shared system. We define what it means for a shared system to meet a security specification and indicate how implementations may be derived from their specifications in some cases.

A summary of related work is given.

Acknowledgements

Many people deserve thanks for the help they have given me throughout the last four years. Not least are Professor Tony Hoare and Jifeng He, who have supervised this work. Among my other colleagues that deserve a mention are: Ali Abdallah, Matthew Arcus, Andy Brightwell (who introduced me to the large example of chapter 3), John Bruton (who informed me about the PRG), Jim Davies, Paul Fertig (who posed the problem of specifying security properties), Paul Gardiner, Yigal Hoffner (who introduced me to computing), Geraint Jones (who posed the problem of the buffer in Appendix C), Ben Potter, Jeff Sanders, Julie Sheppard, Emma Sowton, Alastair Tocher, Phil Wadler and Jim Woodcock, my fellow D.Phil. students and PRG staff. But most thanks is due to my wife, Thea Jacob, for unending moral support and encouragement.

I would like to thank the Science and Engineering Research Council and International Computers Limited for financial support over the past four years.

Contents

Chapter 1

Introduction

1.1 Shared systems and their users

This thesis investigates systems designed to operate in an environment that consists of several users. We will call such systems *shared*. Examples of shared systems range from simple buffers to large operating systems. This work is intended to be of use to specifiers and designers of such systems. We investigate two aspects in particular. The first is the determination of when the users cannot detect the difference between two systems: that is, when one system *refines* another. The second is the determination of what the users can infer about each other's behaviour with the system: that is, describing the system's *security* properties. We introduce this pair of topics in more detail below.

As those who build shared systems have little control over the users of the systems which they build we will make only a minimal number of assumptions about the users. We will assume that each user has an interface to the system that is completely separate from any other user's interface (two otherwise identical interfaces may be distinguished by tagging them with unique names). We will also assume that all communications between users go through the system under consideration. These two assumptions are summed up by saying that the users are *isolated* from each other.

We will record these assumptions, and express the properties of shared systems in the theory known as "Communicating Sequential Processes" (CSP) [Ho85]. This, together with some other areas of mathematics we will use, is introduced in chapter 2.

1.2 Refinement

A system S *refines* another, R say, if every interaction an environment can make with S is an interaction which it could have made with R. Refinement is an important concept. It subsumes the concept of *upward compatibility* (that is, one machine is a suitable replacement for another in the sense that all programs which ran successfully on the original machine also run successfully on the replacement). Refinement is recommended to software engineers as the mathematical tool to help them turn an abstract system design into one sufficiently concrete to be implemented directly. Various theories exist in which the notion of refinement of one system by another can be expressed formally (see for example [Jo80] and [Ho85], we describe the latter in chapter 2). However all these theories describe a refinement relation which assumes that the environment interacting with the systems is a single indivisible entity.

Consider a system that provides, say, a sine calculation service for two clients. If one of the clients wants the sine of a number it sends the number to the service, which later returns the sine. A sequential implementation refuses to accept a request from one client while it is serving the other. Under the usual definitions of refinement such an implementation could not be replaced by a distributed one that provides two servers, one dedicated to each client. This is because the environment as a whole may observe that two calculations are in progress on the distributed system, which is impossible on the sequential system. However, neither user can individually detect the substitution: requests for sines are satisfied in the normal way. The lesson we can draw from this example is that notions of refinement for a single user are not appropriate for systems shared between several users.

In chapter 3 we investigate refinement when there are multiple users of a system and develop appropriate refinement relations for shared systems. We present two refinement relations. One is for systems whose users are co-operating on a task, and where the 'answer' is represented by the state of the shared system when all the users have terminated. An example is an operating system which allows communication between users by shared store; we shall spend some time analysing this example. The second relation is for systems where the final state of the system is not important. The sine server is an example of this case.

1.3 Security

Security, unlike refinement, has no meaning in the context of single-user systems. In the first part of chapter 4 we show how to measure the maximum information flow through, and hence the security of, a shared system.

The measurement of security is in terms of a function which gives, for each system, the set of inferences a user can make about its fellows' interactions with the system from its own interaction. We assume that every user has complete knowledge of the system's potential interactions with its users (perhaps the user has read the code of the system). It is the completeness of this knowledge which gives maximal information flow. Then, given an observation by the user we can give the set of system behaviours consistent with the observation. It is this set which represents the most knowledge that the user can have of its fellows. One of its members must have occurred, but the user cannot be sure which; the user knows that no other of the system's behaviours can have occurred.

Consider a simple word guessing game. One player (the chooser) picks a word from an agreed and known dictionary and presents it to a second player (the guesser), but with some letters blanked out. The guesser's objective is to guess the word. The guesser's observations are strings of letters and asterisks (representing blanked out letters). His inferences are the subsets of the dictionary that match the string. Suppose the string presented is 'poe*'; the chosen word is restricted to the subset of the dictionary $\{\text{poem}, \text{poet}\}$ and the guesser is caught on the horns of a dilemma. If the string presented is 'gru*' then there is only one possibility: 'grub'. The structure of the dictionary (more four letter words beginning with 'poe' than with 'gru') means that it is harder for the guesser to win when the chooser has picked 'poet' than when he has picked 'grub'.

We can sometimes compare the security of two systems and declare one less secure than the other. Consider the simple word game again. We can change the dictionary to one that did not include 'poet' but was otherwise identical. Now being presented with 'poe*' gives the chooser enough information to infer 'poem', but all other observations give the identical information. More information can flow from the chooser to the guesser with the second dictionary than the first: that is, the second dictionary is less secure than the first. It is this possibility of comparison which allows us to talk about measuring the degree of security of a system.

From measurement it is only a small step to specification. In the final part of chapter 4 we generalise the functions that represent the security of a system to allow the writing of specifications without going into all the detail that giving a complete system description requires. The ordering on the inference functions extends to specifications allowing us to decide whether an implementation satisfies a specification in the usual way.

1.4 Summary of the structure of this thesis

In chapter 2 we present the mathematical tools that we use in the rest of the thesis. The chapter comprises relations, pre- and partial orders and communicating sequential processes. In chapter 3 we develop the theory of safe replacement of one shared system by another. This chapter includes a large example. Chapter 4 describes the measurement and specification of the security of systems. We show how to capture several security properties in our style. In chapter 5 we review other approaches to the work we have done. Finally we sum up and report our conclusions in chapter 6.

Chapter 2

Background Theory

Our investigation of shared systems is to be a branch of applied mathematics rather than alchemy, as recommended in [Ho84]. Rather than develop all of the theory from scratch, we base it on well understood areas of mathematics. These are the theory of *relations*, the theory of *pre-* and *partial orders*, and a theory of interacting machines: *Communicating Sequential Processes* (CSP) [Ho85].

2.1 Relations

We are interested in relations for two distinct purposes: to model sequential programs, and as orders between programs. We assume familiarity with the basic concepts of the theory of relations; [Ta41] provides a complete definition. In this subsection we introduce our notation and names for the concepts and explain the relationship with sequential programs. We borrow from the methods for proving correctness of sequential programs to carry out a large proof in chapter 3.

If A and B are any two sets the set of all relations from A to B is written $A \leftrightarrow B$. If $a \in A$ is related to $b \in B$ by relation $\mathbf{r} \in A \leftrightarrow B$ we write $a\,\mathbf{r}\,b$. The *domain* of $\mathbf{r}$ is the largest subset of A each of whose elements is related to some element of B; we write it $\mathrm{dom}\,\mathbf{r}$. The *range* of $\mathbf{r}$ is defined symmetrically, as the largest subset of B each of whose elements are related to some member of A[1]; we write it $\mathrm{ran}\,\mathbf{r}$. If $C \subseteq A$,

[1] The range is often called the *co-domain*.

and $\mathbf{r} \in A \leftrightarrow B$, the *image* of C under $\mathbf{r}$ is defined

$$\mathbf{r}(\!|C|\!) \;\widehat{=}\; \{b : B \mid \exists c : C \cdot c\,\mathbf{r}\,b\}$$

The range of $\mathbf{r}$ is the image of the domain (and all supersets of the domain). A relation $\mathbf{r} \in A \leftrightarrow B$ can be identified with the set

$$\{(a,b) : A \times B \mid a\,\mathbf{r}\,b\}$$

This gives meaning to terms such as "$\mathbf{r} \subseteq \mathbf{p}$" and "$\mathbf{r} \cap \mathbf{p}$".

If $\mathbf{f}$ is a relation with the property

$$\forall a : \mathrm{dom}\,\mathbf{f} \cdot \#(f(\!|\{a\}|\!)) = 1$$

then $\mathbf{f}$ is a *function*[2]. We will write "$\mathbf{f}\,a$" for the unique value in $\mathrm{ran}\,\mathbf{f}$ to which a is related[3] and occasionally "$\mathbf{f}_a$". The set of functions from A to B is written $A \longrightarrow B$.

The *inverse* of a relation $\mathbf{r} \in A \leftrightarrow B$ is the relation $\mathbf{r}^{-1} \in B \leftrightarrow A$ defined by

$$b\,\mathbf{r}^{-1}\,a \iff a\,\mathbf{r}\,a$$

A non-deterministic sequential program can be modelled as a relation from states to states (this is more fully discussed in [HoHe85]). That is: if *STATE* is the set of states of a sequential machine, then a program is modelled as a member of $STATE \leftrightarrow STATE$. If a program $\mathbf{p}$ is started with the machine in a state $s \in \mathrm{dom}\,\mathbf{p}$, then the program will halt in one of the states $\mathbf{p}(\!|\{s\}|\!)$. We can conclude nothing if $\mathbf{p}$ is started in a state outside its domain; this is usually thought of as non-termination, but may be far worse: if the program terminates it may be in any state.

The relation that models the program **skip** is the identity relation I, and the relation that models **abort** is the empty relation $\emptyset$. These are defined:

$$\begin{aligned} a\,I\,b &\iff a = b \\ a\,\emptyset\,b &\iff \mathbf{false} \end{aligned}$$

Sequential composition of programs is modelled by sequential composition of relations.

---

[2] We will not usually write functions in bold symbols.

[3] This is often written "$\mathbf{f}(a)$" instead.

Definition 1 *Let* $\mathbf{r} : A \leftrightarrow B$ *and* $\mathbf{s} : B \leftrightarrow C$. *The* composition *of* $\mathbf{r}$ *and* $\mathbf{s}$ *is defined by:*

$$a(\mathbf{r};\mathbf{s})c \;\widehat{=}\; \exists b : B \cdot a\,\mathbf{r}\,b \wedge b\,\mathbf{s}\,c$$

◇

Composition is associative, has identity I and zero $\emptyset$[4]. To show that a program represented by $(\mathbf{p};\mathbf{q})$ and started with a state drawn from the set C terminates in a state in the set D we need only to show that $C \subseteq \mathrm{dom}\,\mathbf{p}$, $\mathbf{p}(\!|C|\!) \subseteq \mathrm{dom}\,\mathbf{q}$ and $\mathbf{q}(\!|\mathbf{p}(\!|C|\!)|\!) \subseteq D$.

An operator related to composition is *exponentiation* or *iteration.*

Definition 2 *Let* $\mathbf{r} \in A \leftrightarrow A$. *The* n-fold iteration *of* $\mathbf{r}$ *is defined by:*

$$\begin{aligned} \mathbf{r}^0 &\;\widehat{=}\; I \\ \mathbf{r}^{n+1} &\;\widehat{=}\; \mathbf{r};\mathbf{r}^n \end{aligned}$$

◇

The n-fold iteration of $\mathbf{p}$ models the 'for' loop: **for** i **from** 1 **to** n **do** $\mathbf{p}$.

A more useful form of iteration is defined by

Definition 3 *Let* $\mathbf{r} \in A \leftrightarrow A$. *The* iterate *of* $\mathbf{r}$ *is defined:*

$$s\,\mathbf{r}^*\,t \;\widehat{=}\; \exists n : \mathcal{N} \cdot s\,\mathbf{r}^n\,t \wedge t \notin \mathrm{dom}\,\mathbf{r}$$

◇

The iterate of $\mathbf{p}$ models a while loop: **while** "state"$\in \mathrm{dom}\,\mathbf{p}$ **do** $\mathbf{p}$.

To prove that a program modelled by $\mathbf{p}^*$ terminates in a state in D when started with a state in C it is sufficient to find an invariant and a variant. An invariant is a set of states E with the properties $\mathbf{p}(\!|E|\!) \subseteq E$, $C \subseteq E$ and $(E - \mathrm{dom}\,\mathbf{p}) \subseteq D$. A variant is a function f from states to $\mathcal{N}$ with the properties $s \in E \Longrightarrow f\,s \geq 0$ and $s \in E \wedge s\,\mathbf{p}\,t \Longrightarrow (f\,s) > (f\,t)$. These are the proof techniques recommended in [Di76]. As stated above we apply this in chapter 3 to the proof of correctness of a program represented as a relation.

---

[4]This operator is often defined with its operands in the reverse order and written "$\circ$": $\mathbf{s} \circ \mathbf{r} \;\widehat{=}\; \mathbf{r};\mathbf{s}$. We prefer $\mathbf{r};\mathbf{s}$ as it emphasizes the connection with sequential programs.

2.2 Pre- and partial orders

Two important properties a relation may or may not have are *reflexivity* and *transitivity*. These terms are defined as follows:

Definition 4 *Let* $\mathbf{r} \in A \leftrightarrow A$; *then* $\mathbf{r}$ *is* reflexive *if:*

$$I \subseteq \mathbf{r}$$

and is transitive *if:*

$$\mathbf{r}; \mathbf{r} \subseteq \mathbf{r}$$

◇

The identity relation is both reflexive and transitive, the empty relation is only transitive. The iterate of a relation is not reflexive (as its range is disjoint from its domain), but is transitive.

Definition 5 *A relation which is both reflexive and transitive is a* pre-order. ◇

We will use $\rightharpoonup$ as a typical pre-order.

In chapter 4 we use pre-orders to proscribe information flows among users of shared systems. If $a \rightharpoonup b$ then a will be allowed to gain some knowledge of b; if $a \not\rightharpoonup b$ then a is forbidden from discovering anything about b. Making $\rightharpoonup$ a pre-order captures the properties that any user may have knowledge of itself (reflexivity) and if one user can discover something about a second, and the second about a third, then the first can discover something about the third (transitivity).

Another property relations may have is *symmetry*:

Definition 6 *Let* $\mathbf{r} \in A \leftrightarrow A$; *then* $\mathbf{r}$ *is* symmetric *if:*

$$\mathbf{r} = \mathbf{r}^{-1}$$

◇

Definition 7 *A symmetric pre-order is an* equivalence relation. ◇

The identity relation I is an equivalence relation. Any pre-order generates an equivalence relation: define

$$\sim \;\widehat{=}\; \rightarrowtail \cap \rightarrowtail^{-1}$$

then $\sim$ is an equivalence relation. An equivalence relation on A partitions it into equivalence classes.

A further property relations may have is *anti-symmetry*:

Definition 8 *Let* $\mathbf{r} \in A \leftrightarrow A$; *then* $\mathbf{r}$ *is* anti-symmetric *if:*

$$\mathbf{r} \cap \mathbf{r}^{-1} \subseteq \mathbf{I}$$

◇

Definition 9 *An anti-symmetric pre-order is* a partial order. ◇

The identity relation is a partial order. Every pre-order on A generates a partial order on the equivalence classes generated by $\sim$. Define

$$B \propto C \;\widehat{=}\; \exists b : B, c : C \cdot b \rightarrowtail c$$

then $\propto$ is a partial order over the equivalence classes of $\sim$. Some authorities (for example [BLP76]) use $\propto$ and the equivalence classes of $\sim$ to proscribe information flow, rather than $\rightarrowtail$ directly.

Partial orders are important in the theory of computation (see, for example [Sc76,St77]). They are used to express the fact that one program is better than another in some way. For example, take non-deterministic sequential programs $\mathbf{p}$ and $\mathbf{q}$. If

$$\mathrm{dom}\,\mathbf{p} \supseteq \mathrm{dom}\,\mathbf{q}$$

(so $\mathbf{p}$ terminates successfully at least as often as $\mathbf{q}$) and

$$\forall a : \mathrm{dom}\,\mathbf{q} \cdot \mathbf{p}(\!|\{a\}|\!) \subseteq \mathbf{q}(\!|\{a\}|\!)$$

(so $\mathbf{p}$ is more deterministic on their common domain) then we say that $\mathbf{p}$ is better than $\mathbf{q}$. In the next section we will say what it means for one interacting program to be better than another. It is usual to have a worst program (which does nothing useful), but not a best (which would be a

miracle and do everything useful—this is the reason for the "law of the excluded miracle" [Di76][5]). For non-deterministic sequential programs the worst program is **abort**. It is also usual for upper bounds of ascending chains to exist, so that meaning can be given to recursively defined programs and for operators to be continuous (see [St77] for details). In chapter 3 we present, instead of partial orders, two pre-orders which express the fact that one interacting program is better than another in some circumstances.

2.3 Communicating Sequential Processes

"Communicating Sequential Processes" (CSP) was the title of a paper [Ho78] in which a programming notation was introduced that had communication and concurrency constructs as primitives. A program in this notation describes a machine that interacts *synchronously* with its environment; the name *process* is given to such a machine. The notation was investigated by several researchers, and underwent many changes in the process. The notation as it is now generally accepted is defined and discussed in [Ho85]; it retains the distinguishing feature of synchronous communication. Several models or semantics exist for the language, among them [Ho80], [Br83], [Ro82], [BrRo85], [HeSa86], [ReRo86]. Many of the models are tied together in [OH86].

2.3.1 Observing Processes

[OH86] discusses semantics that are *specification-oriented*; that is they give the meaning of a process in terms of the observations that can be made of it. Such a semantics is called specification-oriented because a specification defines all the observations considered acceptable of a machine; a proposed implementation is correct if all the observations of it are allowed by the specification. Clearly, identifying a machine with its observations facilitates proofs of correctness.

The first observation that is usually taken of a process is its interface to the environment. This is a set of *synchronisation events* (or just: *events*) known as the *alphabet* of the process. For a process P we will denote the alphabet αP.

[5]But see [M87] for a way of including miracles in program development.

The second observation that is commonly taken of a process is the set of *traces* it allows over the elements of its alphabet. (The notation we use for traces, and the definitions of the operators on traces we use are given in appendix A.) Each trace records a possible history of interactions between the process and its environment up to some moment in time. The traces of process P are written τP[6]. Because there is always a time before any interactions have taken place, $\langle\rangle$ is a trace of every process: the traces of a process are always *non-empty*. If a process has engaged in in trace t and $s \leq t$ then at some earlier time the process will have engaged in s: the traces of a process are *prefix-closed*. Of course, a process can only engage in events allowed to it by its alphabet: its traces are a subset of all strings over the alphabet. We summarise:

Definition 10 *A set of traces is the set of traces of some process with alphabet A if it is a prefix-closed non-empty subset of A^*.* ◇

Related to traces are the *initials* of a process. This is the set of all events the process may do first:

Definition 11 *The* initials *of P, written ιP*[7]*, are defined*

$$\iota P \triangleq \{e : \alpha P \mid \langle e \rangle \in \tau P\}$$

◇

Traces do not capture liveness properties, only safety properties. However, if we restrict our attention to those processes satisfying a particular liveness criterion—*determinism*—then the alphabet and traces of a process uniquely determine it. Determinism is the liveness property which says that if a process can do something, then it cannot refuse to do it; it characterises the most live processes. This is the model discussed in [Ho80]. Less usefully, we could restrict our attention to those processes which are least live: they can always refuse to do anything. This also gives a subset of processes distinguishable by their traces.

In order to compare processes with arbitrary liveness properties we have to observe processes in more detail. Instead of traces we observe *failures* and *divergences*. A failure represents a way that a process can deadlock

[6] In [Ho85] the traces of P are written *traces*(P).
[7] In [Ho85] the initials of P are written *initials*(P).

after engaging in a particular trace. The failures of a process P, written ϕP[8], satisfy

$$\phi P \subseteq \tau P \times \mathcal{P}(\alpha P)$$

Note that a set of failures induces a relation between τP and $\mathcal{P}(\alpha P)$; we shall use relational notation where convenient. Let $(t, r) \in \phi P$. If the environment offers P the choice of engaging in any of the events in r after it has engaged in t, the process may refuse all of them and deadlock. For this reason r is known as a *refusal set*. If the environment does not offer the process a choice—that is it offers $\{\}$—the process cannot proceed; the empty set is always a refusal set, at any stage of a process' life. If a process can refuse r and $q \subseteq r$, then it can also refuse q; refusals are subset closed. At every stage of a process' evolution it must always be prepared to engage in an event, or to refuse a set containing it; the failures must take account of every event in the alphabet at every stage. To summarise:

Definition 12 *$F \in A^* \leftrightarrow \mathcal{P} A$ is the failures of some process with alphabet A if it satisfies:*

$$\begin{array}{rcl} F & \neq & \emptyset \\ s \frown t \in \operatorname{dom} F & \Longrightarrow & s \in \operatorname{dom} F \\ t\, F\, (r \cup q) & \Longrightarrow & t\, F\, r \\ (t\, F\, r) \wedge a \in A & \Longrightarrow & s\, F\, (r \cup \{a\}) \vee (s \frown \langle a \rangle)\, F\, \{\} \end{array}$$

◇

The last implication of definition 12 allows a process both to engage in an event $((s \frown \langle a \rangle)\, F\, \{\})$ and to refuse it $(s\, F\, (r \cup \{a\}))$. This gives room for nondeterminism in the process. We can recover the traces of a process from the failures:

$$\tau \mathrel{\widehat{=}} \phi; \operatorname{dom}$$

Another useful concept is the collection of sets a process can refuse initially.

Definition 13 *The* refusals *of a process P are defined:*[9]

$$\rho P \mathrel{\widehat{=}} (\phi P)(\!|\{\langle\rangle\}|\!)$$

◇

[8] In [Ho85] the failures of P are written *failures*(P).
[9] In [Ho85] the refusals of P are written *refusals*(P).

Failures record those situations when a process can deadlock with its environment. There is a further liveness property which they do not capture: livelock or *divergence*. After engaging in some trace a process may engage in an infinite series of internal actions and never communicate with the environment again, neither to accept an event nor to refuse a set of events. This is recorded by listing all the traces after which divergence may occur; this set is known as the *divergences* of the process, which we will write δP[10]. The divergences of a process must be a subset of its traces. As well as an infinite series of internal actions a divergent process may do anything; hence any extension of a divergent trace may be observed, after which any refusal may also be observed. As it cannot be determined that a divergent process has stopped diverging all such extended traces are also divergences of the process. To summarise:

Definition 14 *Let $F \in A^* \leftrightarrow \mathcal{P}A$, and $D \subseteq A^*$. Then F, D are the failures and divergences of some process with alphabet A, if, in addition to the axioms of definition 12:*

$$\begin{aligned} D &= D \hat{\_} A^* \\ F &\supseteq D \times \mathcal{P}A \end{aligned}$$

◇

If the time at which an event happens is important then we have to take *timed observations*. Such models are discussed in [ReRo86]; we will not make use of them in this thesis.

2.3.2 Replacing one process by another

We can compare processes in a similar way to sequential programs. One process, P say, is better than another, Q say, if any observation we make of P could have been made of Q. That is, if we observe P we can get no evidence against the hypothesis that we are observing Q. We record this $P \sqsupseteq Q$. This is, of course, just the 'Turing test' [Tu50].

Example 1 Rather than the usual example of the Turing test taken from [Tu50] consider the following. A car driver has an interface to a car that

[10] In [Ho85] the divergences of P are written $divergences(P)$.

includes the brake pedal and the speedometer. Suppose that the usual implementation of the brakes is changed to one where the pedal is not linked with the wheels at all. Now if the driver never presses the break pedal, or only presses it when the fuel has just run out, he cannot detect the re-implementation. As soon as he presses it when there is plenty of fuel in the tank he will observe depression of the brake followed by steady speed. This observation was impossible before, and he will be aware that a change has been made. △

Of course, the relation $\sqsupseteq$ depends on which observations we are interested in. We can only compare two processes if their alphabets are the same. If we observe traces as well, $\sqsupseteq$ is defined:

$$P \sqsupseteq Q \;\widehat{=}\; \alpha P = \alpha Q \wedge \tau P \subseteq \tau Q$$

In this case the deterministic processes with alphabet A form a complete lattice with top the process with traces $\{\langle\rangle\}$ and bottom the deterministic process with traces A^*. It is usual to use the names $STOP_A$ and RUN_A to stand for for these two processes. $STOP_A$ is better than any other process *as long as we are only interested in traces.* It never does anything, and every process has a state where it has done nothing; we can never detect the difference between a process which has done nothing yet but might in the future, and one which will never do anything.

We will usually be interested in the liveness of processes as well. So we must observe failures and divergences as well. In this case refinement is defined:

Definition 15 *One process, P say, refines another, Q say, written $P \sqsupseteq Q$, if:*

$$\alpha P = \alpha Q \wedge \phi P \subseteq \phi Q \wedge \delta P \subseteq \delta Q$$

◇

Note how this order says that P is better than Q if it fails less often and diverges less often. Sometimes we will write *"is a replacement for"* for *refines.* Every deterministic process is a top element of this order. There is a unique bottom element with alphabet A, which we will call $CHAOS_A$. It is defined formally in appendix B.

The models of [ReRo86] take a very different approach to comparing processes, based on metric spaces rather than partial orders.

2.3.3 Operators on processes

There are many operators on processes; new ones may be defined as convenient. The major operators are defined in terms of their semantics in appendix B; here we will describe them informally and briefly. For fuller descriptions, including the important laws satisfied by the operators, the reader is referred to [Ho85].

We have already introduced the constants $STOP_A$ and $CHAOS_A$. They represent the deadlocked process and the divergent process respectively. $STOP_A$ must refuse any subset of A that the environment offers it, while $CHAOS_A$ may do anything, accept an event, refuse the whole set or never interact with the environment again. When the alphabet is clear from the context we will drop the subscript.

The process "$e : E \rightarrow P_e$" must engage in one of the events of the set E, a say, after which it goes on to behave like P_a. e is a bound variable of the construction. This is known as the *generalised choice* operator. The process "$a \rightarrow P$" is defined

$$a \rightarrow P \;\widehat{=}\; e : \{a\} \rightarrow P$$

The process $a_0 \rightarrow P_0 | \ldots | a_k \rightarrow P_k$ is defined to be equal to

$$e : \{a_0, \ldots, a_k\} \rightarrow \textbf{if } e = a_0 \textbf{ then } P_0 \textbf{ or} \ldots \textbf{or } e = a_k \textbf{ then } P_k \textbf{ fi}_e$$

provided that all of the $a_0, \ldots, a_k$ are distinct.

We will use two parallel operators, one symmetric "$\|$" and one asymmetric "$//$". The former allows two processes to evolve, but forces them to synchronise on their common alphabets. The latter enslaves its first argument to its second; the second is the environment in which the first is to evolve.

There are two kinds of choice operators: internal and external. The former is represented "$P \sqcap Q$"; it allows the process the choice of behaving like P or like Q without the environment's interference. The latter is represented "$P [\!] Q$"; it allows the environment to make the choice (by selecting an initial either of P or of Q) or to delegate the choice to the process (by selecting an event which is an initial of both P and Q).

Sequential composition of two processes is represented "$P; Q$". ';' has unit "$SKIP_A$", the process which does nothing but terminate successfully.

(*SKIP* is different from *STOP*, the process which does nothing but terminate unsuccessfully.)

If F is a function from processes to processes, then we can define processes by recursion. "$\mu X : A \cdot F\,X$" is the least process in the $\sqsupseteq$ ordering that satisfies:

$$P = F\,P \wedge \alpha P = A$$

F is said to be *guarded* if every occurrence of X in $F\,X$ is prefixed by at least one event. When F is guarded the equation $P = F\,P$ has a unique solution, so we may define recursive processes by writing them in this way.

Input and output are modelled by structured events. The event '$c.v$' represents the communication of value v on channel c between two processes. A process generating output is written

$$c!v \rightarrow P$$

for some process P. This is equivalent to $c.v \rightarrow P$. The corresponding process accepting input is written

$$c?x : V \rightarrow Q_x$$

'x' is a bound variable, restricted to the set V. The meaning of this notation is given formally in appendix B. Note that the asymmetry between input and output is syntactic rather than semantic. In the former case the process offers an external choice between many (similar) events, while in the latter the process is determined on a unique event. It is dangerous to assume an asymmetry between input and output when investigating information flow in synchronous systems (see appendix C). An asymmetry only exists when no upper bound is known on the amount of buffering provided on communication channels. This is a rare set of circumstances and so forms an important reason for choosing CSP as the theory of interacting machines used in this thesis over theories such as that of lazy stream-processing functions (see e.g. [Hen80]).

Let 'c' be a name. The process "$c : P$" is one that behaves in a similar way to P, but has name c. Whenever P can engage in an event e the process $c : P$ can engage in $c.e$. Let 'C' be a set of names. The process "$C : P$" is also one that behaves in a similar way to P, but whenever P can engage in the event e, $C : P$ can engage in any of the events $\{c.e \mid c \in C\}$.

Because we will be forming many processes in this way it is convenient to introduce the operator : on sets:

$$C : D \;\widehat{=}\; \{c.d \mid c \in C \wedge d \in D\}$$

This is just the Cartesian product of the two sets, but written so as to match the usual CSP notation.

If B is a set of events and P is a process then "$P\backslash B$" is the process that behaves rather like P, but with the occurrence of events in B hidden from the environment. If P could ever engage in an infinite series events from B, then $P\backslash B$ will diverge in the same circumstances.

2.4 Application of theory to systems serving isolated concurrent environments

We will model systems as processes. Whether or not a system is *shared* is a property of its environment. For a system to be shared we suppose that the environment is composed from concurrently existing sub-environments that are isolated from each other, except by their effect on the system. We will refer to such environments as *isolated concurrent environments*. Such an environment can be identified with a disjoint partition of the alphabet of the system being shared, one *window* for each component or *user* in the environment.

Example 2 Consider a typical operating system, where each user has a unique name, drawn from some set N, say. Each user can issue commands from some set C. Typically C includes commands such as "create a new sub-directory", "send message *text* to user m", etc. The alphabet of the system is $N : C$, while the alphabet of the user with name n is $\{n\} : C$. The set $\{\{n\} : C \mid n \in N\}$ is the disjoint partition of the alphabet that characterises the environment. △

Example 3 A buffer that carries messages from some set M is a shared system. The two windows are that of the writer and that of the reader, $\{in.m \mid m \in M\}$ and $\{out.m \mid m \in M\}$, respectively. △

The restriction operator on traces " $\restriction$ " is going to be important in our study of shared systems. It gives us a way of finding how a trace of the

whole system looks through a window. We can generalise this operator to one on processes "@" in the following way:

Definition 16 *Let $B \subseteq \alpha P$. Then*

$$P@B \triangleq (\!|\tau P|\!) \upharpoonright B$$

◇

$P@B$ is *not* the same as hiding. $P@B$ is a set of traces, not a process; furthermore it does not include traces which are the result of introducing divergence by hiding. It just enables us to ask what traces may be seen through the window B.

Lemma 1 *For any process P and $B \subseteq \alpha B$, $P@B$ is a prefix-closed non-empty set of traces.*
Proof Both properties follow as τP also enjoys them, and $\upharpoonright B$ is distributive. □

In the next two chapters we investigate how to characterise refinement of shared systems and how to measure the maximum information flow through a shared system. Both pieces of theory lead to ways of specifying behaviour of shared systems.

Chapter 3

Local Replacement

3.1 Introduction

3.1.1 The problem

Recall that

$$P \sqsupseteq Q$$

means that if we replace Q by P, then *no* environment can tell that a substitution has been made (chapter 2). This is a very strong criterion. In practice we may only expose our systems to a restricted class of environments, and not care if environments outside this class can detect the substitution.

A case where this holds is when the environment is composed of isolated concurrent sub-environments, either working in co-operation on the same problem or independently on different problems. It is easy to show that $\sqsupseteq$ is too strong when the sub-environments are performing independent computations.

Example 4 Let *OLD* be an existing implementation of two independent services for two users on a sequential machine, defined by:

$$OLD \triangleq$$
$$(a1 \rightarrow a2 \rightarrow b1 \rightarrow b2 \rightarrow STOP) \| (b1 \rightarrow b2 \rightarrow a1 \rightarrow a2 \rightarrow STOP)$$

with one user's interface being $\{a1, a2\}$ and the other $\{b1, b2\}$. (For definiteness, suppose $a1$ is insertion of a coin by one user and $b1$ the extraction

of a chocolate bar by the same user. Similarly, $b1$ and $b2$ are insertion of a coin and extraction of a chocolate bar respectively, but by the other user.) As there is no logical connection between the two services it seems sensible to distribute them as follows:

$$NEW \;\widehat{=}\; (a1 \rightarrow a2 \rightarrow STOP) \| (b1 \rightarrow b2 \rightarrow STOP)$$

Unfortunately $\langle a1, b1 \rangle \in (\tau NEW - \tau OLD)$, and so $NEW \not\sqsupseteq OLD$. $\triangle$

Co-operating isolated sub-environments are more complicated; an example using them is delayed until section 3.2.3. It is no surprise that simple examples for this case are hard to find; communication by shared state is well known to be a complex phenomenon to analyse.

Why does NEW fail to refine OLD, in example 4? Because there is an environment which can detect the difference. For example consider:

$$E \;\widehat{=}\; a1 \rightarrow (b1 \rightarrow ko \rightarrow STOP \;[\!]\; a2 \rightarrow ok \rightarrow STOP)$$

We calculate:

$$OLD \,//\, E = ok \rightarrow STOP \tag{3.1}$$

and

$$NEW \,//\, E = (ko \rightarrow STOP) \sqcap (ok \rightarrow STOP) \tag{3.2}$$

From equation 3.1 we see that the environment will always signal ok with OLD, but equation 3.2 shows that it may signal ko with NEW. It is this possibility that leads $\sqsupseteq$ to reject NEW as a suitable replacement for OLD. The property that E has which enables it to detect the switch of NEW for OLD is that it is a single sequential environment whose alphabet spans both $\{a1, a2\}$ and $\{b1, b2\}$. We cannot rewrite E as a parallel composition of two isolated processes, with alphabets $\{a1, a2\}$ and $\{b1, b2\}$ respectively, as the occurrence of $b1$ must occur after that of $a1$.

Our intuition tells us that if independent isolated users access these systems they cannot tell the difference between OLD and NEW. In this chapter we develop two new notions of refinement, one for independent users, and one for co-operating users.

3.1.2 A further generalisation of the Turing test

We have already explained (section 2.3.2) how $\sqsupseteq$ captures the Turing test. The local refinements we are looking for can be viewed in the same light: in this version of the test we allow the environment to be not a single tester, but several. Further, the testers are not allowed to communicate while the test is in progress, other than by their effect on the shared machine under test. If we are interested in judging the machine's suitability for co-operating users the testers are allowed to have a joint strategy and to compare notes at the end of the test; when judging the suitability for independent users these facilities are denied to the testers.

Example 5 Continuing example 1, we note that the interface to the brake system of a car is more than just the pedal and sense of speed. At the rear of the car are red lamps which come on when the brake pedal is depressed. The driver of the car cannot see these lamps, and so he would not notice if a reimplementation omitted to wire them up. Another driver following the first car can see these lamps. What strategies can the two drivers come up with to test the brake system?

First, we consider the case of co-operating users. After a journey the two drivers compare the number of times the first depressed the brake pedal with the number of times the second driver saw the lights illuminated. Unless the number of times the brake is depressed is zero this test will detect the reimplementation.

When the users are not allowed a joint strategy and comparison afterwards the task of the testers is harder. The leading driver has no information at all; the following driver can see both the state of the lights and the behaviour of the car they are installed in. It is not enough for the following driver to see the lead car slow down without the brake lights being illuminated, as the lead driver may be slowing down by use of the gears. If the following driver is suitably experienced he can distinguish deceleration by use of brakes from deceleration by use of gears. In this case the faulty lights can be detected. △

In the next two sections we formalize these tests. We tackle the more demanding environment—that is, one constructed from co-operating users—before the less knowledgeable.

3.2 Local refinement for co-operating users

3.2.1 Local Equivalence

Consider a shared system with alphabet A, and a disjoint partition $\mathcal{A}$ of A, each element of $\mathcal{A}$ being the window or interface to a different user. For any observation of the whole system the user with sub-alphabet B (for any $B \in \mathcal{A}$) sees only the projection of this observation onto B. We use this as the basis of equivalence relations on the (important) observations of the system.

Traces First we deal with traces. The local view, or projection, of a trace, t say, is given by $t \upharpoonright B$, for some set of events B representing a user's window onto the system. For two system traces to be equivalent each user's view of both traces must be identical.

Definition 17 *Let A be a set of events, $\mathcal{A}$ be a partition of A and s,t be traces drawn from A^*. We call s and t* locally equivalent *w.r.t. $\mathcal{A}$, written $s \cong_{\mathcal{A}} t$, if*

$$\forall B : \mathcal{A} \cdot s \upharpoonright B = t \upharpoonright B$$

◇

This relation on traces is an equivalence relation:

Lemma 2 *Let $\mathcal{A}$ be a partition of alphabet A, then $\cong_{\mathcal{A}}$ is an equivalence relation on A^*.*
Proof Follows as $=$ is an equivalence relation. □

The importance of this relation is that when two traces are locally equivalent they have an identical effect on an environment constructed as independent concurrent sub-environments. We can formalize this:

Lemma 3 *Let $E1$ and $E2$ be a pair of processes representing the only two users of a system, with the property $\alpha E1 \cap \alpha E2 = \{\}$, and let $\mathcal{A} = \{\alpha E1, \alpha E2\}$. Let s and t be two traces drawn from $\tau(E1 \| E2)$, then*

$$s \cong_{\mathcal{A}} t \Longrightarrow (E1 \| E2)/s = (E1 \| E2)/t$$

Proof

$$
\begin{array}{ll}
(E1 \| E2)/s & \\
= & [\text{properties of } \restriction] \\
E1/(s \restriction \alpha E1) \| E2/(s \restriction \alpha E2) & \\
= & [s \cong_{\mathcal{A}} t] \\
E1/(t \restriction \alpha E1) \| E2/(t \restriction \alpha E2) & \\
= & [\text{properties of } \restriction] \\
(E1 \| E2)/t &
\end{array}
$$

□

Failures Now we extend this relation to failures. Two local views of failures are the same if the local views of the traces are the same and the local views of the refusals are the same. Unlike traces, we can recover a set if we know all of its pieces, as their union. This motivates the following definition:

Definition 18 *Let A be a set of events, $\mathcal{A}$ be a partition of A and (s, q), (t, r) be failures drawn from $A^* \times \mathcal{P}A$. We call (s, q) and (t, r) locally equivalent w.r.t. $\mathcal{A}$, written $(s, q) \cong_{\mathcal{A}} (t, r)$, if*

$$s \cong_{\mathcal{A}} t \wedge q = r$$

◇

There is no ambiguity in using the same symbol for local equivalence of failures as well as of traces as the context will always tell us which is intended. Local equivalence of failures is also an equivalence relation:

Lemma 4 *Let $\mathcal{A}$ be a partition of alphabet A, then $\cong_{\mathcal{A}}$ is an equivalence relation on $A^* \times \mathcal{P}A$.*
Proof Follows as $\cong_{\mathcal{A}}$ is an equivalence relation on traces (lemma 2). □

There is no analogue of lemma 3 as there is no analogue of the after operator "/" for failures.

Divergences Finally we come to divergences. As these are traces we just use the same definition as before (definition 17). Lemma 2 still applies.

3.2.2 Local Refinement

Now we are in a position to define *local refinement* or *local replaceability* (we use the terms interchangeably). This is a relation between processes that describes when one process is a suitable replacement for another, as long as its environment is composed of isolated concurrent sub-environments. It is dependent upon the alphabet partition generated by the alphabets of the sub-environments.

Definition of local refinement

Consider the definition of $\sqsupseteq$:

$$NEW \sqsupseteq OLD \;\widehat{=}\; \phi NEW \subseteq \phi OLD \wedge \delta NEW \subseteq \delta OLD$$

We can rewrite the right-hand side:

$$(\forall g : \phi NEW \cdot \exists f : \phi OLD \cdot f = g) \wedge (\forall c : \delta NEW \cdot \exists d : \delta OLD \cdot c = d)$$

This highlights the relationship between the equality relation on observations, $=$, and the global refinement relation, $\sqsupseteq$: whenever an observation of NEW is possible there is an observation of OLD which stands in the relationship "$=$" to it. The equality relation is appropriate as, potentially, an environment can tell *exactly* the difference between observations; failure to distinguish between two observations is only guaranteed if they are equal.

To define local replacement, we just substitute local equivalence for equality: potentially, co-operating users can distinguish between observations which are not locally equivalent; they can never distinguish between observations which are locally equivalent.

Definition 19 *Let $\mathcal{A}$ be disjoint partition of some set of events. One system, NEW, is* a local refinement *w.r.t. $\mathcal{A}$ of another, OLD, written*

$$NEW \trianglerighteq_{\mathcal{A}} OLD$$

if they both have the same alphabet, $\bigcup \mathcal{A}$, and

$$(\forall g : \phi NEW \cdot \exists f : \phi OLD \cdot f \cong_{\mathcal{A}} g) \wedge (\forall c : \delta NEW \cdot \exists d : \delta OLD \cdot c \cong_{\mathcal{A}} d)$$

$\diamond$

This definition captures the properties:

- Any user's view of an observation of the new system must be a possible view of an observation of the old system.
- The users must be able to agree on at least one observation of the old system which could account for all of their individual views under the new system.

Example 6 [1] Let $A = \{a, b\}$ and $\mathcal{A} = \{\{a\}, \{b\}\}$. Three processes with alphabet A are:

$$\begin{aligned} P &\triangleq (a \rightarrow b \rightarrow a \rightarrow STOP_A) [\![(b \rightarrow a \rightarrow b \rightarrow STOP_A) \\ Q &\triangleq (a \rightarrow STOP_A) \sqcap (b \rightarrow STOP_A) \\ R &\triangleq (a \rightarrow b \rightarrow Q) [\![(b \rightarrow a \rightarrow Q) \end{aligned}$$

We can see from examination of the failures and divergences that $P \trianglerighteq_{\mathcal{A}} R$, $R \trianglerighteq_{\mathcal{A}} P$ and $P \neq R$. The failures of P are

1	$(\langle\rangle, \{\})$								
2	$(\langle a\rangle, \{\})$	3	$(\langle a\rangle, \{a\})$	4	$(\langle b\rangle, \{\})$	5	$(\langle b\rangle, \{b\})$		
6	$(\langle a, b\rangle, \{\})$	7	$(\langle a, b\rangle, \{b\})$	8	$(\langle b, a\rangle, \{\})$	9	$(\langle b, a\rangle, \{a\})$		
10	$(\langle a, b, a\rangle, \{\})$	11	$(\langle a, b, a\rangle, \{a\})$						
12	$(\langle a, b, a\rangle, \{b\})$	13	$(\langle a, b, a\rangle, \{a, b\})$						
14	$(\langle b, a, b\rangle, \{\})$	15	$(\langle b, a, b\rangle, \{a\})$						
16	$(\langle b, a, b\rangle, \{b\})$	17	$(\langle b, a, b\rangle, \{a, b\})$						

The failures of R include all of those of P and additionally,

18	$(\langle a, b\rangle, \{a\})$	19	$(\langle b, a\rangle, \{b\})$
20	$(\langle a, b, b\rangle, \{\})$	21	$(\langle a, b, b\rangle, \{a\})$
22	$(\langle a, b, b\rangle, \{b\})$	23	$(\langle a, b, b\rangle, \{a, b\})$
24	$(\langle b, a, a\rangle, \{\})$	25	$(\langle b, a, a\rangle, \{a\})$
26	$(\langle b, a, a\rangle, \{b\})$	27	$(\langle b, a, a\rangle, \{a, b\})$

Given a failure of P we can give the identical failure of R as a locally equivalent failure; this proves $P \trianglerighteq_{\mathcal{A}} R$. Given a failure of R that is identical to one of P's we can just take this as the locally equivalent failure. For a

[1] I am grateful to Paul Gardiner for this example.

failure of R whose number is 18 or greater in the list, the following table gives a locally equivalent failure of P:

$$\begin{array}{cccc} (18,9) & (19,7) & & \\ (20,14) & (21,15) & (22,16) & (23,17) \\ (24,10) & (25,11) & (26,12) & (27,13) \end{array}$$

This proves $R \trianglerighteq_A P$. $P \neq R$ is proved by noting that there are failures of R which are not failures of P $\triangle$

Simple properties of local refinement

Relationship with refinement An immediate consequence of definition 19 is that local refinement is, as we wanted, a weaker relation than ordinary refinement:

Lemma 5 *For any disjoint partition, $\mathcal{A}$ say, of the common alphabet of systems OLD and NEW,*

$$NEW \sqsupseteq OLD \Longrightarrow NEW \trianglerighteq_A OLD$$

Proof

$$\begin{array}{ll} & NEW \sqsupseteq OLD \\ \Longleftrightarrow & \text{[definition of } \sqsupseteq\text{]} \\ & \forall g : \phi NEW \cdot \exists f : \phi OLD \cdot f = g \\ \wedge & \forall c : \delta NEW \cdot \exists d : \delta OLD \cdot c = d \\ \Longleftrightarrow & \text{[lemmas 2–4]} \\ & \forall g : \phi NEW \cdot \exists f : \phi OLD \cdot f \cong_A g \\ \wedge & \forall c : \delta NEW \cdot \exists d : \delta OLD \cdot c \cong_A d \\ \Longleftrightarrow & \text{[definition 19]} \\ & NEW \trianglerighteq_A OLD \end{array}$$

□

Corollary *Let $A = \bigcup \mathcal{A}$, then $CHAOS_A$ is a bottom of $\trianglerighteq_A$.* □

Order properties Many useful properties of $\sqsupseteq$ follow from it being a complete partial order. Unfortunately, $\trianglerighteq_A$ is not always a partial order. It is always a pre-order, however.

Lemma 6 *Let $\mathcal{A}$ be a disjoint partition of some alphabet, then $\trianglerighteq_{\mathcal{A}}$ is a pre-order.*
Proof

Reflexivity: Follows from lemma 5, as $\sqsupseteq$ is reflexive.

Transitivity: Let P,Q and R be systems such that $P \trianglerighteq_{\mathcal{A}} Q$ and $Q \trianglerighteq_{\mathcal{A}} R$. We must show that given a failure (divergence) of P there is a failure (divergence) of R locally equivalent to it. Let $f \in \phi P$. Then there is a failure $g \in \phi Q$ such that $g \cong_{\mathcal{A}} f$. Also, there is a failure $h \in \phi R$ such that $h \cong_{\mathcal{A}} g$. $h \cong_{\mathcal{A}} f$ follows by the transitivity of $\cong_{\mathcal{A}}$ (lemma 4). A similar argument holds for the divergences.

□

Lemma 7 *Let $\mathcal{A}$ be a disjoint partition of some alphabet, with at least two non-empty members. Then $\trianglerighteq_{\mathcal{A}}$ is not anti-symmetric.*
Proof Example 6 provides two processes that locally refine each other, but which are not equal. □

3.2.3 A large example

The problem that follows is (a simplified version of) the motivating example for the work reported on in this chapter. It is about replacing a monolithic mainframe, where communication between co-operating users is via shared store, with several small machines networked together to form a distributed mainframe. A desired property of the new implementation is *upward compatibility*; that is: *every* program that ran on the monolithic mainframe must run in the same way on the distributed mainframe, but with possibly different timings. Of course, the hope is that all programs run to completion faster on the new machine than on the old one. As the programs that run on the old machine are constructed as concurrent combinations of isolated but co-operating users, the appropriate correctness criterion for a replacement is local refinement of the original.

The interface

The first thing we must fix is the common alphabet of the machines, and the disjoint partition of this alphabet among the users. The important

actions are operations on the store. As a simplification we only consider instructions which either only read the store or only write to it; instructions which do both, such as test-and-set, are not considered here.

We use

A for the set of names of the store locations (addresses),

I for the set of names of the users,

$C \triangleq \{read, write\}$ for the set of commands available, and

M for the set of storable values

The common alphabet, written U, is defined by:

$$U \triangleq A : I : C : M$$

The reason for this order will become apparent below. Typical events from this alphabet are that of the i^{th} user reading the value m from location a:

$$a.i.read.m$$

and the j^{th} user writing the value n to location b:

$$b.j.write.n$$

The interface to the i^{th} user, written U_i, is defined by:

$$U_i \triangleq A : \{i\} : C : M$$

and the disjoint partition of U which represents this division among users is:

$$\mathcal{U} \triangleq \{U_i \mid i \in I\}$$

Monolithic shared store

Definition of monolithic shared store Now we can define the original monolithic machine. We do this by defining small components and then assembling them.

A single location, written LOC, behaves like a simple variable of type M, with the provision that it refuses to output a value until it has been initialised.

$$\begin{aligned} LOC &\triangleq write?m : M \rightarrow LOC1_m \\ LOC1_m &\triangleq \mu X : (C : M) \cdot (LOC [\![read!m \rightarrow X) \end{aligned}$$

LOC behaves like a variable with an unstructured environment. We can make LOC suitable for use by several users by prefixing it with the set of user names: '$I : LOC$' describes a process that behaves like a variable whose environment is structured as several named users. We can name a multi-user location by prefixing it with its address: '$a : I : LOC$' describes a shared variable with name a.

Finally we can assemble these named shared variables into a complete store:

$$OLD \triangleq \|_{a:A}\, a : I : LOC$$

OLD is the process that describes the behaviour of the original machine.

Properties of monolithic shared store Two properties will be of use to us in our later analysis. These are the properties of *regularity* and *consistency*.

A read is said to be regular if the value obtained is the same as the value last written to the location in question:

Definition 20 *A particular occurrence of the event $a.i.read.m$ in a trace t is* regular *if:*

$$\exists u, v \cdot (t = u \,\hat{}\, \langle a.i.read.m \rangle \,\hat{}\, v) \wedge (lw\, u\, a = m)$$

where '$lw\, u\, a$' is the last value written to location a in the trace u:

$$\begin{aligned} lw\, u\, a &\triangleq value(last(u \restriction (\{a\} : I : \{write\} : M))) \\ value\, a.i.o.m &\triangleq m \\ last(t \,\hat{}\, \langle e \rangle) &\triangleq e \end{aligned}$$

$\diamond$

Note that if there are several occurrences of the same read event in a trace each one may or may not be regular. This definition extends in a natural way to traces, failures and processes.

Definition 21 *A trace t is* regular *if all its reads are regular.* ◇

Definition 22 *A failure is* regular *if its trace is regular.* ◇

Definition 23 *A process is* regular *if all its failures are regular.* ◇

If a read, trace, failure or process is not regular we call it *irregular*. Regularity is essentially a property of the traces of a process.

Some simple lemmas about regularity:

Lemma 8 *$s \hat{} t$ is a regular trace $\Longrightarrow$ s is a regular trace.*
Proof Direct from the definition. □

Lemma 9 *If P is a regular process, then P does not diverge.*
Proof Suppose $d \in \delta P$, and let $a \in A$, $i \in I$ and $m, n \in M$. Then, as divergences are post-fix closed, the irregular trace $d \hat{} \langle a.i.write.m, a.i.read.n \rangle \in \delta D$. □

Lemma 10 *If P is a regular process, and $Q \sqsupseteq P$, then Q is regular*
Proof Immediate as $Q \sqsupseteq P \Longrightarrow \phi Q \subseteq \phi P$. □

The next lemma characterises the traces of OLD:

Lemma 11 $\tau OLD = \{t : U^* \mid t \textit{ is regular}\}$
Proof The proof is by induction on the length of the trace.

Base case $\langle\rangle$ is a regular trace and is a trace of OLD.

Induction step Suppose that every regular trace of length k is a trace of OLD. Let s be such a trace, and let

$$Z \triangleq \mathrm{dom}\,(lw\ s)$$

(Z is the set of locations which have been written to.) We can calculate

$$\begin{aligned}
&\{e : U \mid s \hat{} \langle e \rangle \text{ is regular}\} \\
&\quad = (A : I : \{write\} : M) \cup \{a.i.read.(lw\ s\ a) \mid a \in Z \wedge i \in I\} \\
&\quad = \iota((\|_{a:Z}\ a : I : LOC1_{(lw\ s\ a)}) \| (\|_{a:A-Z}\ a : I : LOC)) \\
&\quad = \iota(OLD/s)
\end{aligned}$$

So we see that any single event extension to s that maintains regularity also gives a trace of OLD, and vice versa.

□

We can also characterise the failures of OLD:

Lemma 12

$$\phi OLD = \{(t, r) \mid t \in \tau OLD \wedge r \subseteq (Y\, t)\}$$

where

$$\begin{array}{lcl} Y\, t & \widehat{=} & \cup \begin{array}{l} \bigcup_{a:(Z\, t)} \{a\} : I : \{read\} : (M - \{lw\, t\, a\}) \\ \bigcup_{a:A-(Z\, t)} \{a\} : I : read : M \end{array} \\ Z\, t & \widehat{=} & \mathrm{dom}\,(lw\, t) \end{array}$$

Proof As OLD is deterministic, a refusal after a trace s may be any subset of the complement, relative to its alphabet, of its initials after s. The result follows by simple calculation and lemma 11. □

Corollary *OLD is regular.* □

A read is consistent if the value read had been previously written to the location, but not necessarily on the last write.

Definition 24 *A particular occurrence of a read event $a.i.read.m$ in a trace t is* consistent *if:*

$$\exists u, v \cdot (t = u \mathbin{\hat{}} \langle a.i.read.m \rangle \mathbin{\hat{}} v) \wedge (u \upharpoonright (\{a\} : I : \{write\} : \{m\}) \neq \langle\rangle)$$

◇

Again, we can extend this definition to traces, failures and processes.

Definition 25 *A trace is* consistent *if all of its reads are consistent.* ◇

Definition 26 *A failure is* consistent *if its trace is consistent.* ◇

Definition 27 *A process is* consistent *if all of its failures are consistent.* ◇

We will call a read, trace, failure or process *inconsistent* if it is not consistent.

We can summarise these definitions: an inconsistent read is one that should never have happened at all; an irregular read is one that either occurs too late or should never have happened at all. It is easy to show that regularity is a stronger condition than consistency:

Lemma 13 *a.i.read.m is regular in t* $\Longrightarrow$ *a.i.read.m is consistent in t.*
Proof As *a.i.read.m* is regular in t we can, by definition 20, find $j \in I$ and traces u, v and w such that—$(*)$:

$$(t = u \,\hat{}\, \langle a.i.read.m \rangle \,\hat{}\, v) \wedge (u \restriction (\{a\} : I : write : M) = w \,\hat{}\, \langle a.j.write.m \rangle)$$

Then we have:

$$\begin{array}{ll}
u \restriction (\{a\} : I : \{write\} : \{m\}) & \\
= & [\{m\} \subseteq M] \\
(u \restriction (\{a\} : I : \{write\} : M) \restriction (\{a\} : I : \{write\} : \{m\}) & \\
= & [*] \\
(w \,\hat{}\, \langle a.j.write.m \rangle) \restriction (\{a\} : I : \{write\} : \{m\}) & \\
\neq & [a.j.write.m \in \{a\} : I : \{write\} : \{m\}] \\
\langle\rangle &
\end{array}$$

which is the condition for consistency given in definition 24. □

Corollary

1. *t is a regular trace* $\Longrightarrow$ *t is a consistent trace.*
2. *f is a regular failure* $\Longrightarrow$ *f is a consistent failure.*
3. *P is a regular process* $\Longrightarrow$ *P is a consistent process.*

□

Some simple lemmas about consistency:

Lemma 14 *s* $\hat{}$ *t is a consistent trace* $\Longrightarrow$ *s is a consistent trace.*
Proof Direct from the definition. □

Lemma 15 *If P is a consistent process, then P cannot diverge before every value has been written to every location.*
Proof Let $a \in A$, $i \in I$ and $m \in M$. Suppose $d \in \delta P$ is such that $d \restriction \{a.i.write.m\} = \langle\rangle$. Then, as divergences are post-fix closed, the inconsistent trace $d \hat{} \langle a.i.read.m \rangle \in \delta P$. □

Lemma 16 *If P is a consistent process, and $Q \sqsupseteq P$, then Q is consistent*
Proof Immediate as $Q \sqsupseteq P \Longrightarrow \phi Q \subseteq \phi P$. □

Lemma 17 *Let $\mathcal{A}$ be a disjoint partition of some alphabet, and let P and Q be processes with this alphabet. If P is consistent and $Q \trianglerighteq_{\mathcal{A}} P$ then Q is consistent.*
Proof Suppose $s \hat{} \langle a.i.read.m \rangle \hat{} t \in \tau Q$ is such that $a.i.read.m$ is an inconsistent read. By definition 24, s cannot contain a write of m to a. As the traces of a process are prefix-closed, $s \hat{} \langle a.i.read.m \rangle \in \tau Q$. But this trace cannot be re-ordered to be consistent, and so is not locally equivalent to any trace of P. □

As we shall see below, when constructing a local replacement for *OLD* we can relax the restriction of regularity. However, consistency gives us a bound on how far regularity can be relaxed.

Lemma 18 *$NEW \trianglerighteq_U OLD \Longrightarrow NEW$ is consistent*
Proof From the corollaries to lemmas 12 and 13, and from lemma 17. □

Replicated shared store

We give two different implementations of replicated store. One will turn out to locally refine *OLD*, the other not. We wish to replace a single machine by several—one for each user—networked together. At the top level both the distributed architectures have the form:

$$NETWORK \,//\, (\|_{i:I}\, NODE_i)$$

Our problem is to make this arrangement look like a single monolithic machine. As we are assigning one user to a node we insist that I is a finite set.

Accessing store on a remote machine is relatively slow when compared with accessing local store. If all the shared store was on one machine it

is likely that there would be no advantage in the new architecture. The solution is to put a copy of the entire store in each node. Now every access to the store is local, and each access happens as fast as a local node will allow. The effect of a write will have to be sent to all other stores on the network. The two new machines we design differ in the bandwidth of the network along which updates are sent. In one case, $NEW1$, we provide a separate network for each location, in the other, $NEW2$, all locations must share a network, and each node must preserve the order of the updates made at it when forwarding to the network.

Again we proceed by defining component processes and then combining them to form more complex processes. The first of these models the behaviour of a local copy of a location. Its alphabet contains channels for reading and writing on, just as LOC, and also two channels for communicating with the network, *dlvr* and *net*. We set

$$\begin{aligned} C' &\triangleq \{dlvr, net\} \\ D &\triangleq C \cup C' \\ INT &\triangleq A : I : C' : M \end{aligned}$$

Here INT is the set of internal events. The alphabet of the local copy is $D : M$. Its behaviour, $COPY$, is defined by the mutually recursive equations:

$$\begin{aligned} COPY &\triangleq (write?m : M \rightarrow COPY'_m) [\!] (dlvr?m : M \rightarrow COPY''_m) \\ COPY'_m &\triangleq \mu X : (D : M) \cdot ((net!m \rightarrow COPY''_m) [\!] (dlvr?n : M \rightarrow X)) \\ COPY''_m &\triangleq \mu X : (D : M) \cdot (COPY [\!] read!m \rightarrow X) \end{aligned}$$

$COPY$ is similar to LOC. The major differences are the presence of two channels for input and its insistence on the output of a value received on the *write* channel to the *net* channel before proceeding. Note that $COPY$ is always ready to communicate any value on the *dlvr* channel. We have also taken the decision to let the local copy ignore any value sent to it between receiving one on *write* and forwarding it on *net*. This choice leads to a simpler analysis than that of accepting the latest input.

The nodes of $NEW1$ are simply defined as a parallel array of local copies of locations. The i^{th} node, for $i \in I$, is:

$$NODE1_i \triangleq \|_{a:A}\, a : i : COPY$$

The nodes of $NEW2$ are a little more complex, as we ensure that the order updates are sent out on the *net* channel is the same as that received on the *write* channel. This is done by composing each $NODE1_i$ in parallel with a process, B_i, that makes this demand:

$$\begin{array}{lcl} B_i & \triangleq & \underline{*}([\!]_{a:A}\, a : i : (B' [\!] B'')) \\ B' & \triangleq & write?m : M \rightarrow net?m : M \rightarrow SKIP \\ B'' & \triangleq & read?m : M \rightarrow SKIP \end{array}$$

Putting B_i in parallel with $NODE1$ is the same as restricting communications between a node and the network to flow through a zero-length buffer. Setting the length of the buffer to zero is another design decision to make analysis simpler.

The i^{th} node of $NEW2$ is simply defined:

$$NODE2_i \triangleq NODE1_i \| B_i$$

Now we come to the two networks. A network which accepts just one value from any location and then distributes it to all the other nodes before terminating successfully is defined:

$$\begin{array}{lcl} NET & \triangleq & [\!]_{i:I}\, i.net?m : M \rightarrow DLVR_{m,I-\{i\}} \\ DLVR_{m,J} & \triangleq & \begin{cases} \sqcap_{j:J}\, j.dlvr!m \rightarrow DLVR_{m,J-\{j\}} & \text{if } J \neq \{\} \\ SKIP & \text{otherwise} \end{cases} \end{array}$$

A network with one simple network for each location is defined by:

$$NET1 \triangleq \|_{a:A}\, a : (\underline{*}NET)$$

while that composed of one sequential network serving all the locations is given by:

$$NET2 \triangleq \underline{*}([\!]_{a:A}\, a : NET)$$

This completes our modelling of the two new systems. All that remains is to give their formal definitions:

$$\begin{array}{lcl} NEW1 & \triangleq & NET1 \,//\, (\|_{i:I}\, NODE1_i) \\ NEW2 & \triangleq & NET2 \,//\, (\|_{i:I}\, NODE2_i) \end{array}$$

A result that will be of use later is:

Lemma 19 $\delta NEW2 = \{\}$

Proof None of the constituent processes of $NEW2$ diverge (by construction), so we just need to check that we have not hidden an infinite series of network events. Let n be the cardinality of I. Each write leads to n network events, one on a *net* channel and $n-1$ on *dlvr* channels. In any trace t of $NEW2$ there are at most $\#t$ unforwarded writes, so this can lead to at most $\#t \times (n-1)$ uninterrupted internal events. After these network events the next event must be external. □

The original machine can replace both of the reimplementations, in any circumstances:

Lemma 20

$$OLD \sqsupseteq NEW1 \ \wedge \ OLD \sqsupseteq NEW2$$

Proof Pick a failure of OLD, (t, r). Let u be the trace formed by inserting the network events associated with each write in t immediately following the write, first the output to the network, and then the deliveries in some standard order. It is routine to check that $(u, r \cup INT)$ is a failure of processes defined in the same way as $NEW1$ and $NEW2$, but with $\|$ instead of $//$. The definition of hiding tells us that $(u, r \cup INT) \upharpoonright INT = (t, r)$ is a failure of both $NEW1$ and $NEW2$. OLD has no divergences, by construction, so we have nothing more to check. □

Replicated shared store does not refine monolithic shared store

It is easy to show that neither $NEW1$ nor $NEW2$ refines OLD in the usual sense.

Lemma 21 *Let the cardinality of A be at least one, and those of I and M be at least two, then $NEW1$ is irregular.*

Proof Let $a \in A$, $i, j \in I$ and $m, n \in M$, with $i \neq j$ and $m \neq n$. Also, let

$$t \triangleq \langle a.i.write.m, a.i.write.n, a.j.read.m \rangle$$

It is routine to check that t is both irregular and a trace of $NEW1$. □

Corollary *Under the conditions of the lemma, $NEW1 \not\sqsupseteq OLD$.* □

Lemma 22 *Let the cardinality of A be at least one, and those of I and M be at least two, then $NEW2$ is irregular.*
Proof Identical to the proof of lemma 21, but with $NEW2$ for $NEW1$. □

Corollary *Under the conditions of the lemma, $NEW2 \not\sqsupseteq OLD$.* □

Some replicated shared stores do not locally refine monolithic shared store

As neither $NEW1$ nor $NEW2$ invents values they are consistent. Lemma 18 tells us that they are worth investigating as candidates to locally refine OLD. Unfortunately $NEW1$ is too irregular to be such a refinement.

Lemma 23 *Let the cardinalities of A, I and M be at least two, then*

$$NEW1 \not\succeq_U OLD$$

Proof Let $a, b \in A$, $i, j \in I$ and $m, n \in M$ be such that $a \neq b$, $i \neq j$ and $m \neq n$. Define

$$t \triangleq \langle a.i.write.m, a.i.write.n, b.i.write.n, b.j.read.n, a.j.read.m \rangle$$

It is routine to check that $t \in \tau NEW1$. For a trace s to be locally equivalent to t we must have

$$\begin{aligned} s \restriction U_i &= t \restriction U_i = \langle a.i.write.m, a.i.write.n, b.i.write.n \rangle \\ s \restriction U_j &= t \restriction U_j = \langle b.j.read.n, a.j.read.m \rangle \\ s \restriction U_k &= t \restriction U_k = \langle \rangle \qquad \text{if } i \neq k \neq j \end{aligned}$$

For s to be consistent the event $b.j.read.n$ must follow the event $b.i.write.n$. So,

$$s = (s \restriction U_i)^\frown (s \restriction U_j) = t$$

This fixes the only consistent trace locally equivalent to t as t itself. However t is not regular, so all traces locally equivalent to t are irregular. Hence, all the failures locally equivalent to $(t, \{\})$ are irregular. The lemma follows by the corollary to lemma 12. □

This result shows that consistency is not sufficient for local refinement of OLD; the implication in lemma 18 is in one direction only.

Some replicated shared stores locally refine monolithic shared store

$NEW2$, like $NEW1$, is irregular but consistent. Unlike $NEW1$ it is not too irregular to locally refine OLD. The proof of this fact is long, but not complicated. We can tackle the divergences very easily.

Lemma 24 $\forall c : \delta NEW2 \cdot \exists d : \delta OLD \cdot c \cong_U d$
Proof Immediate from $\delta NEW2 = \{\}$ (lemma 19) □

Showing the analogous result for the failures is much more work. We consider $NEW2$, but with its internal working exposed. Define

$$IMP \triangleq NET2 \| (\|_{i:I} NODE2_i)$$

This is just the definition of $NEW2$, but with // replaced by $\|$.

We are going to show that every failure of IMP can be re-ordered to give a trace which is a trace of OLD when the network events are hidden. The re-ordering is captured by a relation, **reg**. This relation is defined in terms of the composition of two others:

$$\mathbf{reg} \triangleq \mathbf{rwr}; \mathbf{rrd}$$

The relation **rwr** reorganizes the writes in the trace, while **rrd** reorganizes some of the reads.

We use iteration of relations to define **rwr**:

$$\mathbf{rwr} \triangleq \mathbf{later}^*$$

later relates a trace to a similar one, but with the number of writes separated from their associated outputs to the network reduced by one. It is defined by:

$$v \frown \langle a.i.write.m \rangle \frown w \frown \langle a.i.net.m \rangle \frown x$$
$$\mathbf{later}$$
$$v \frown w \frown \langle a.i.write.m, a.i.net.m \rangle \frown x$$
$$\text{if } w \neq \langle\rangle \wedge w \restriction \{a.i.net.m\} = \langle\rangle$$

We give this relation the name "**later**" as it "moves" writes later in the trace.

Similarly to **rwr**, **rrd** is defined by:

$$\mathbf{rrd} \triangleq \mathbf{earlier}^*$$

earlier relates a trace to a similar one, but with fewer reads caught between a write to the read location and the associated delivery of the new value to the reader's site. It is defined by:

$$(v \,\hat{}\, \langle a.i.write.m, a.i.net.m\rangle \,\hat{}\, w \,\hat{}\, \langle a.j.dlvr.m\rangle \,\hat{}\, x)$$
earlier
$$(v \,\hat{}\, (w \restriction R_j) \,\hat{}\, \langle a.i.write.m, a.i.net.m\rangle \,\hat{}\, (w \setminus R_j) \,\hat{}\, \langle a.j.dlvr.m\rangle \,\hat{}\, x)$$
$$\text{if } i \neq j \wedge w \restriction R_j \neq \langle\rangle \wedge w \restriction \{a.j.dlvr.m\} = \langle\rangle$$

where

$$R_j \;\widehat{=}\; A : \{j\} : \{read\} : M$$

This relation "moves" reads earlier in the trace.

As we have noted above (section 2.1), there is a close relation between relations and non-deterministic sequential programs. We exploit this fact in the following lemmas to prove, in effect, liveness and partial correctness of **reg** with initial condition $t \in \tau IMP$. First we deal with liveness:

Lemma 25 $\tau IMP \subseteq \mathrm{dom}\,\mathbf{reg}$

Proof We exploit the analogy with non-deterministic sequential programs by giving bound functions on the "bodies" of the two loops. The remaining proof obligations are trivial.

For **later**, we define

$$b1\,t \;\widehat{=}\; \#\{n : \mathcal{N} \mid p1\,t\,n\}$$

where

$$p1\,t\,n \;\widehat{=}\; \exists\, a : A, i : I, m : M \cdot t[n] = a.i.write.m \wedge t[n+1] \neq a.i.net.m$$

($t[n]$ is the n^{th} element of the trace t.) '$b1$' counts the number of separated write/net pairs in the trace. It has range $\mathcal{N}$. It is easy to see that

$$b1\,y = (b1\,z) - 1$$

if z **later** y. Clearly, if $b1\,t = 0$, then t cannot be matched against

$$v \,\hat{}\, \langle a.i.write.m\rangle \,\hat{}\, w \,\hat{}\, \langle a.i.net.m\rangle \,\hat{}\, x$$

with $w \neq \langle\rangle \wedge w \restriction \{a.i.net.m\} = \langle\rangle$, and so $t \notin \mathrm{dom}\,\mathbf{later}$.

Similarly, for **earlier** we define

$$b2\,t \;\widehat{=}\; \#\{(n,n') : \mathcal{N}^2 \mid (n < n') \wedge (p2\,t\,n\,n')\}$$

where

$$\begin{aligned} &p2\,t\,n\,n' \;\widehat{=}\\ &\quad \exists\, a : A, i, j : I, m : M \cdot\\ &\qquad (t[n] = a.i.write.m) \wedge (t[n'] = a.j.dlvr.m)\\ &\qquad \wedge (t[n+1..n'-1] \upharpoonright R_j \neq \langle\rangle) \end{aligned}$$

The relation '$b2$' counts the number of gaps between a write and the associated delivery to another user that contain reads by that user. Like $b1$, ran $b2 = \mathcal{N}$. It is clear that

$$b2\,y \;=\; (b2\,z) - 1$$

if z **earlier** y. Also, if $b2\,t = 0$, then t cannot be matched against

$$v \,\hat{}\, \langle a.i.write.m, a.i.net.m\rangle \,\hat{}\, w \,\hat{}\, \langle a.j.dlvr.m\rangle \,\hat{}\, x$$

with

$$(i \neq j) \wedge (w \upharpoonright R_j \neq \langle\rangle) \wedge (w \upharpoonright \{a.j.dlvr.m\} = \langle\rangle)$$

and so $t \notin \mathrm{dom}$ **earlier**. □

Now we tackle the partial correctness of **reg** in a series of lemmas, each of which deals with a different property of interest. First, the internal events of two traces related by **reg** are in the same order.

Lemma 26 $t\,\mathbf{reg}\,u \Longrightarrow t \upharpoonright INT = u \upharpoonright INT$

Proof We show that both **later** and **earlier** preserve this property. Because $\upharpoonright$ distributes through $\hat{}$ we have, for **later**,

$$\begin{aligned} &(v \,\hat{}\, \langle a.i.write.m\rangle \,\hat{}\, w \,\hat{}\, \langle a.i.net.m\rangle \,\hat{}\, x) \upharpoonright INT\\ &\quad = \quad (v \upharpoonright INT) \,\hat{}\, (w \upharpoonright INT) \,\hat{}\, \langle a.i.net.m\rangle \,\hat{}\, (x \upharpoonright INT)\\ &\quad = \quad (v \,\hat{}\, w \,\hat{}\, \langle a.i.write.m, a.i.net.m\rangle \,\hat{}\, x) \upharpoonright INT \end{aligned}$$

as $a.i.write.m \notin INT$. A similar proof holds for **earlier**, when we note that

$$(w \upharpoonright R_j) \upharpoonright INT = w \upharpoonright (R_j \cap INT) = w \upharpoonright \{\} = \langle\rangle$$

□

Secondly we show that every "complete" trace is related to one that has each write immediately prior to its associated output to the network. A complete trace is a trace of *IMP* with its full complement of network events for each write event.

Lemma 27 *Let* $(t, r) \in \phi IMP$ *be such that* $INT \subseteq r$. *Then*

$$t \,\mathbf{reg}\, u \Longrightarrow$$
$$\forall a : A, i : I, m : M \cdot \langle a.i.write.m, e\rangle \,\mathbf{in}\, u \Longrightarrow e = a.i.net.m$$

Proof We claim that **rwr** establishes the property and that **rrd** preserves it. The second of these facts is trivial consequence of the definition of **earlier**. The first requires a little more justification: t contains a corresponding output to the network for each write, otherwise the failure would not satisfy $INT \subseteq r$, and each output to the network is later than the write. (both simple properties of $COPY$ defined on page 38). These properties are preserved by **later**. The first claim then follows directly from the necessity of a trace not to be in dom **later** to be in ran **rwr**. □

Next we show that **reg** is a stronger relation than $\cong_u$, when restricted to "complete" traces.

Lemma 28 *Let* $(t, r) \in \phi IMP$ *be such that* $INT \subseteq r$. *Then*

$$t \,\mathbf{reg}\, u \Longrightarrow (t \setminus INT) \cong_u (u \setminus INT)$$

Proof The proof of this lemma is similar to that of lemma 26, but now there are two cases for each relation. For **later**, with $i \neq j$, we have

$$\begin{aligned}(v \frown \langle a.i.write.m\rangle \frown w \frown \langle a.i.net.m\rangle \frown x) \upharpoonright U_j \\ = \; (v \upharpoonright U_j) \frown (w \upharpoonright U_j) \frown (x \upharpoonright U_j) \\ = \; (v \frown w \frown \langle a.i.write.m, a.i.net.m\rangle \frown x) \upharpoonright U_j\end{aligned}$$

as $a.i.write.m, a.i.net.m \notin U_j$. If $i = j$ then $w \upharpoonright U_j = \langle\rangle$ as a simple property of the zero-length buffer, B_j, defined on page 39. This gives us

$$\begin{aligned}(v \frown \langle a.j.write.m\rangle \frown w \frown \langle a.j.net.m\rangle \frown x) \upharpoonright U_j \\ = \; (v \upharpoonright U_j) \frown \langle a.j.write.m, a.j.net.m\rangle \frown (x \upharpoonright U_j) \\ = \; (v \frown w \frown \langle a.j.write.m, a.i.net.m\rangle \frown x) \upharpoonright U_j\end{aligned}$$

Again, a similar proof holds for **earlier**, once we have noted (for $j \neq k$)

$$\begin{aligned} w \restriction U_j &\in R_j^* \\ (w \restriction R_j) \restriction U_k &= \langle\rangle \\ (w \setminus R_j) \restriction U_k &= w \restriction U_k \end{aligned}$$

The first of these properties follows from the behaviour of the network and and argument similar to that in the proof of lemma 27; the other two follow from simple properties of $\restriction$ and $\setminus$. □

Finally we show that every complete trace is related by **reg** to only regular traces.

Lemma 29 *Let $(t,r) \in \phi IMP$ be such that $INT \subseteq r$. Then*

$$t \,\mathbf{reg}\, u \Longrightarrow u \textit{ is regular}$$

Proof Both **rwr** and **rrd** play their part in establishing the regularity of u. We have $u \notin \mathrm{dom}\,\mathbf{earlier}$, as a trivial property of **reg**, and also $u \notin \mathrm{dom}\,\mathbf{later}$, as **earlier** preserves this property (lemma 27). As t is complete and **reg** only relates permutations, so is u. We also have lemmas 26 and 28. From these facts we deduce that if we pick a read $a.i.read.m$ in u then either u is of the form:

$$v \,\hat{}\, \langle a.i.write.m, a.i.net.m\rangle \,\hat{}\, w \,\hat{}\, \langle a.i.read.m\rangle \,\hat{}\, x$$

where

$$w \restriction \{a\} : I : \{write\} : \{m\} = \langle\rangle$$

or u is of the form:

$$v \,\hat{}\, \langle a.j.write.m, a.j.net.m\rangle \,\hat{}\, x \,\hat{}\, \langle a.i.dlvr.m\rangle \,\hat{}\, w \,\hat{}\, \langle a.i.read.m\rangle \,\hat{}\, y$$

where

$$(i \neq j) \wedge ((x \,\hat{}\, w) \restriction \{a\} : I : \{write\} : \{m\} = \langle\rangle)$$

In the second case, we know there are no writes to a in x, by the properties of $NET2$ (defined on page 39). If there are any writes to a making the read irregular, then they must lie in w. In both cases the same argument shows that there are no writes to a in w.

Suppose x contains a write to a. The value written cannot be m, from the form of u. The associated network event for user i must occur later in u, as u is a complete trace of $NEW2$. This network event cannot occur in y, because no reads can occur between a write and the associated network event at the reading node, or u would be in dom **later** or in dom **earlier**. Thus the network event must be in w. There must be a later network event in w that re-establishes the value of m in a for user i, by the definition of $COPY$ (page 38). The write that led to this network event must occur in v, by the form of u. But then all the network events associated with it must also occur in v, or u would be in dom **later** or in dom **earlier**. This contradicts the hypothesis that the network event of the interfering write can occur in w, and hence that there can be an interfering write.

Thus, in u, no write can intervene between a write and a read of the value written by that write. That is: u is regular. □

Finally, we turn our attention to the refusal sets. In the same fashion as the function lw above, we define lo. This is the function that returns the last value output to the network for a given location and trace:

$$lo\, u\, a \;\widehat{=}\; value(last(u \upharpoonright \{a\} : I : \{net\} : M))$$

The first use of this function is in characterising the failures of IMP.

Lemma 30 *Let $(t, r) \in \phi IMP$ be such that $INT \subseteq r$. Then*

$$r - INT \subseteq \cup \begin{matrix} \bigcup_{a:Z} \{a\} : I : \{read\} : (M - \{lo\, t\, a\}) \\ \bigcup_{a:A-Z} \{a\} : I : \{read\} : M \end{matrix}$$

where

$$Z \;\widehat{=}\; \mathrm{dom}\,(lo\, t)$$

Proof Direct from the definition of IMP and its components. □

Lemma 30 states that when there are no outstanding internal events, every local copy of a location has the same state and this state is directly dependent on the last output to the network for this location. The next lemma shows how to construct a failure of OLD from a failure of IMP.

Lemma 31 *Let $(t, r) \in \phi IMP$ be such that $INT \subseteq r$. Then*

$$t\, \mathbf{reg}\, u \Longrightarrow (u \setminus INT, r - INT) \in \phi OLD$$

Proof First, we establish —(∗):

$$
\begin{array}{ll}
& lo\,t \\
= & \qquad [\text{Lemma 26}] \\
& lo\,u \\
= & \qquad [\text{Lemma 27}] \\
& lw\,u \\
= & \qquad [\text{property of } \setminus] \\
& lw\,(u \setminus INT)
\end{array}
$$

We can conclude that:

$$
\begin{array}{ll}
& r - INT \\
\subseteq & \qquad [\text{lemma 30}] \\
\cup & \bigcup_{a:Z} \{a\} : I : \{read\} : (M - \{lo\,t\,a\}) \\
& \bigcup_{a:A-Z} \{a\} : I : \{read\} : M \\
= & \qquad [*] \\
\cup & \bigcup_{a:Y} \{a\} : I : \{read\} : (M - \{lw\,(u \setminus INT)\,a)\} \\
& \bigcup_{a:A-Y} \{a\} : I : \{read\} : M
\end{array}
$$

where

$$
\begin{array}{rcl}
Z & \triangleq & \mathrm{dom}\,(lo\,t) \\
Y & \triangleq & \mathrm{dom}\,(lw(u \setminus INT))
\end{array}
$$

We know that u is regular from lemma 29, and so is a trace of OLD by lemma 11. But every pair consisting of a trace s of OLD, and a subset of

$$
(\bigcup_{a:Y'} \{a\} : I : \{read\} : (M - \{lw\,s\,a\})) \cup (\bigcup_{a:A-Y'} \{a\} : I : \{read\} : M)
$$

where

$$
Y' \triangleq \mathrm{dom}\,(lw\,s)
$$

is a failure of OLD, by lemma 12. □

At last we are able to prove that every failure of $NEW2$ is locally equivalent to a failure of OLD.

Lemma 32 $\forall g : \phi NEW \cdot \exists f : \phi OLD \cdot f \cong_u g$

Proof Pick $(s, q) \in \phi NEW2$. Choose $(u, p) \in \phi IMP$ that can give rise

to (s, q); that is, choose (u, p) so that $u \setminus INT = s$ and $p - INT = q$. By lemmas 25 and 31 there is a trace v such that

$$u \,\mathbf{reg}\, u$$
$$(v \setminus INT, q) \in \phi OLD$$

Also, a simple corollary of lemma 28 gives us:

$$(v \setminus INT, q) \cong_u (s, q)$$

□

We are now in a position to assert that $NEW2$ is a suitable replacement for OLD, given that the environments which interact with these systems are restricted to those structured as a concurrent combination of isolated users.

Lemma 33 $NEW2 \trianglerighteq_u OLD$
Proof Direct from lemmas 24 and 32. □

This shows that irregularity is not a bar to being a local refinement of OLD.

3.3 Local refinement for independent users

3.3.1 Independent replacement

In section 3.2.2 we showed that in order to replace one system used by co-operating users by another, two properties are necessary:

- Any user's view of an observation of the new system must be a possible view of an observation of the old system.
- The users must be able to agree on at least one observation of the old system which could account for all of the individual views under the new system.

This is too strong if the users are not co-operating, but independent. The first condition is necessary, but the second can be relaxed.

Example 7 Consider again the systems OLD and NEW of example 4. The observation $(\langle a1, b1\rangle, \{\})$ can be made of NEW. There is no locally equivalent failure of OLD, with respect to $A = \{\{a1, a2\}, \{b1, b2\}\}$. Thus $NEW \not\trianglerighteq_A OLD$, which is not what we want. △

Definition of independent refinement

This leads us to a definition of local refinement for independent users, which we shall call *independent refinement* or *independent replacement.*

Definition 28 *Let $\mathcal{A}$ be a disjoint partition of some set of events. One system, NEW, is an independent refinement w.r.t. $\mathcal{A}$ of another, OLD, written $NEW \trianglerighteq_{\mathcal{A}} OLD$, if they both have alphabet $\bigcup \mathcal{A}$, and*

$$\begin{aligned} &\forall B : \mathcal{A} \cdot \\ &\quad \forall g : \phi NEW \cdot \exists f : \phi OLD \cdot f \restriction B = g \restriction B \\ \wedge \;&\forall c : \delta NEW \cdot \exists d : \delta OLD \cdot c \restriction B = d \restriction B \end{aligned}$$

◇

This definition is not of the same form as those of local refinement or ordinary refinement, as we have rewritten it above (section 3.2.2). Universal quantification over members of $\mathcal{A}$ occurs at the outside of the expression, not inside (in $\cong_{\mathcal{A}}$). This is exactly what we need to capture the relaxation of the second condition above.

Simple properties of independent refinement

We outline some of the important properties of independent refinement.

Relationship with local refinement Independent refinement is the weakest of the three refinement relations discussed in this chapter.

Lemma 34 *For any disjoint partition, $\mathcal{A}$ say, of the alphabet of OLD and NEW:*

$$NEW \trianglerighteq_{\mathcal{A}} OLD \Longrightarrow NEW \trianglerighteq_{\mathcal{A}} OLD$$

Proof For an arbitrary $B \in \mathcal{A}$ and given a failure or divergence of NEW we must exhibit a failure or divergence, respectively, of OLD that has the same

projection onto B. $NEW \trianglerighteq_{\mathcal{A}} OLD$ guarantees that there is an observation of OLD that has the same projection onto each $B \in \mathcal{A}$ as the original observation. We take this as our observation of OLD. □

Corollary *$CHAOS_{\bigcup \mathcal{A}}$ is a bottom of $\trianglerighteq_{\mathcal{A}}$.* □

Example 8 Recall the systems OLD and NEW of examples 4 and 7. Set $\mathcal{A} = \{\{a1, a2\}, \{b1, b2\}\}$. Then we have $NEW \trianglerighteq_{\mathcal{A}} OLD$, as hoped for. Note that $OLD \not\trianglerighteq_{\mathcal{A}} NEW$ as $(\langle a1 \rangle, \{b1, b2\}) \in \phi OLD$ does not have an equivalent w.r.t. $\{b1, b2\}$ in NEW. △

Order properties Like local refinement, independent refinement is a pre-order but not a partial order.

Lemma 35 *Let $\mathcal{A}$ be a disjoint partition of some alphabet, then $\trianglerighteq_{\mathcal{A}}$ is a pre-order.*
Proof

Reflexivity Follows from lemmas 6 and 34.

Transitivity Let P, Q and R be such that $P \trianglerighteq_{\mathcal{A}} Q$ and $Q \trianglerighteq_{\mathcal{A}} R$. We must show, for any $B \in \mathcal{A}$, that given an observation, t say, of P there is an observation of R with the same local projection at B. As $P \trianglerighteq_{\mathcal{A}} Q$ there is an observation u of Q such that $t \restriction B = u \restriction B$; as $Q \trianglerighteq_{\mathcal{A}} R$ there is an observation, v say, such that $u \restriction B = v \restriction B$. The result follows by transitivity of $=$.

□

Lemma 36 *Let $\mathcal{A}$ be a disjoint partition of some alphabet, where at least two members are non-empty. Then $\trianglerighteq_{\mathcal{A}}$ is not anti-symmetric.*
Proof Follows by lemma 34 and example 6. □

This concludes our brief look at independent refinement.

Chapter 4

Security

4.1 Introduction

Our goal in the last chapter was to use local views to define what a correct refinement of a shared system was. The old product and the relation $\trianglerighteq_A$ stood as a specification for a new product. Our goal in this chapter is to use local views to measure information flow through a system and specify limits on this flow.

4.1.1 Security is a safety property

For local refinement we showed that the natural local observations are based on failures; for security specifications the natural local observations to work with are based on *traces*. This is because security is a *safety* property rather than a *liveness* property; we want to express properties such as: if the system ever does anything, then it must not leak information. A system cannot leak information by deadlocking, for any user that insists on interacting with it will also deadlock and will not be able to make use of the information.

A simple approach to dealing with systems in which deadlock is detectable by its users is to add a new event to its alphabet, $i.\mathbf{deadlock}$ say, for each user i that can detect the deadlock, and replace the system by a similar one that announces that it is about to deadlock.

4.2 The security model

Security is about preventing one user of a system from discovering what other users of the system have done. Such a user of a system has two pieces of evidence from which it can deduce this information:

- The trace of interactions between the user and the system, *the observation*, and
- Its knowledge of the system's capabilities.

For the second of these, we will take a worse case position: that the user "knows" the complete code of the system. In our formulation this means that, for a system described by the CSP process S, the user knows τS. The set of all observations by a user viewing S through a subset of αS, B say, is $S@B$ (definition 16, page 22).

We will use the following simple examples throughout this chapter:

Example 9 Let $A = \{a, b\}$ and S be defined by:

$$S \mathrel{\widehat{=}} a \rightarrow b \rightarrow STOP$$

This describes a system, S, which engages first in the event a, then in b, and then halts. We calculate:

$$\begin{aligned} \tau S &= \{\langle\rangle, \langle a\rangle, \langle a, b\rangle\} \\ S@\{a\} &= \{\langle\rangle, \langle a\rangle\} \\ S@\{b\} &= \{\langle\rangle, \langle b\rangle\} \end{aligned}$$

△

Example 10 With the same alphabet, we define R by:

$$R \mathrel{\widehat{=}} (a \rightarrow b \rightarrow STOP) \llbracket (b \rightarrow STOP)$$

This system engages in exactly one occurrence of b and one or no occurrences of a; if a occurs it must be before b. We have:

$$\begin{aligned} \tau R &= \{\langle\rangle, \langle a\rangle, \langle b\rangle, \langle a, b\rangle\} \\ R@\{a\} &= \{\langle\rangle, \langle a\rangle\} \\ R@\{b\} &= \{\langle\rangle, \langle b\rangle\} \end{aligned}$$

Notice that the projections of R at $\{a\}$ and $\{b\}$ are the same as for S of example 9, even though its traces are different. $\triangle$

Example 11 Again, with the same alphabet, define:

$$Q \triangleq \mu X \cdot ((a \rightarrow X) [\!] (b \rightarrow STOP))$$

Q allows any number of a's and one b, after which it terminates. Again, we calculate:

$$\begin{aligned} \tau Q &= \{a\}^* \hat{\ } \{\langle\rangle, \langle b \rangle\} \\ Q@\{a\} &= \{a\}^* \\ Q@\{b\} &= \{\langle\rangle, \langle b \rangle\} \end{aligned}$$

$\triangle$

Example 12 Finally, with alphabet $\{a, b\}$, we define:

$$P \triangleq (a \rightarrow STOP) [\!] (b \rightarrow STOP)$$

P engages in either one a or one b, but not both, before terminating. We have:

$$\begin{aligned} \tau P &= \{\langle\rangle, \langle a \rangle, \langle b \rangle\} \\ P@\{a\} &= \{\langle\rangle, \langle a \rangle\} \\ P@\{b\} &= \{\langle\rangle, \langle b \rangle\} \end{aligned}$$

$\triangle$

4.2.1 Inferences

Now we can ask what a user may infer about the behaviour of the other users sharing a system S. For each local observation the user can make through its window, B say, it can calculate *all* the possible traces that S could have engaged in which give rise to that particular observation. (Remember: we are assuming that the user knows τS.) There is no information from within the system for deciding which of these traces actually occurred, but one of them must have.

Definition 29 *For an arbitrary system, S, the* inference function of S, *written* infer_S, *is:*

$$\text{infer}_S\, B\, \ell \;\widehat{=}\; \{t : \tau S \mid t \restriction B = \ell\} \qquad \text{if } B \subseteq \alpha S \wedge \ell \in S@B$$

◇

The functions infer and @ are almost inverses, in the following sense:

Lemma 37 *For any system S, any set of events $B \subseteq \alpha S$ and any trace $\ell \in S@B$:*

$$(\text{infer}_S\, B\, \ell)@B = \{\ell\}$$

Proof

$$\begin{aligned}(\text{infer}_S\, B\, \ell)@B &= \{t : \tau S \mid t \restriction B = \ell\}@B \\ &= \{t \restriction B \mid t \in \tau S \wedge t \restriction B = \ell\} \\ &= \{\ell\}\end{aligned}$$

□

There are two simple identities:

Lemma 38 *For any system S and $\ell \in \tau S$*

$$\begin{aligned}\text{infer}_S\, (\alpha S)\, \ell &= \{\ell\} \\ \text{infer}_S\, \{\}\, \langle\rangle &= \tau S\end{aligned}$$

□

The first equation states that if a system is observed fully (i.e the window $= \alpha S$), then it is possible to deduce exactly what has happened. The second states that if we cannot observe a system (the window $= \{\}$) then anything could have happened, limited only by the system's capabilities. Lemma 38 also shows that traces and inference functions of systems are equivalent: definition 29 shows how to calculate infer_S from τS, while this lemma shows how to calculate τS from infer_S.

Another property of inference functions is:

Lemma 39 *For any system S, and $B \subseteq \alpha S$:*

$$\langle\rangle \in \mathrm{infer}_S\, B\, \langle\rangle$$

Proof Follows from $\langle\rangle \restriction B = \langle\rangle$ and $\langle\rangle \in \tau S$. □

This states that if we have observed nothing it is always possible that no user has engaged in any interactions with the system; we cannot be sure that anything has happened until we have evidence.

We now illustrate this function with the systems of the examples 9–12.

Example 13 The inferences that can be made of S (example 9), through $\{a\}$ and $\{b\}$ are:

$$\begin{aligned}
\mathrm{infer}_S\, \{a\}\, \langle\rangle &= \{\langle\rangle\} \\
\mathrm{infer}_S\, \{a\}\, \langle a\rangle &= \{\langle a\rangle, \langle a, b\rangle\} \\
\mathrm{infer}_S\, \{b\}\, \langle\rangle &= \{\langle\rangle, \langle a\rangle\} \\
\mathrm{infer}_S\, \{b\}\, \langle b\rangle &= \{\langle a, b\rangle\}
\end{aligned}$$

A user viewing S through $\{a\}$ cannot tell when, or if, b occurs. △

Example 14 For R of example 10 we have:

$$\begin{aligned}
\mathrm{infer}_R\, \{a\}\, \langle\rangle &= \{\langle\rangle, \langle b\rangle\} \\
\mathrm{infer}_R\, \{a\}\, \langle a\rangle &= \{\langle a\rangle, \langle a, b\rangle\} \\
\mathrm{infer}_R\, \{b\}\, \langle\rangle &= \{\langle\rangle, \langle a\rangle\} \\
\mathrm{infer}_R\, \{b\}\, \langle b\rangle &= \{\langle b\rangle, \langle a, b\rangle\}
\end{aligned}$$

Note that while the projections of R are the same as S at $\{a\}$ and $\{b\}$, the set of inferences at each point are at least as large (example 13). A user viewing R through $\{b\}$ cannot tell if a has occurred, which is not the case for S. △

Example 15 For Q of example 11 we have:

$$\begin{aligned}
\mathrm{infer}_Q\, \{a\}\, \ell &= \{\ell, \ell\,\hat{}\,\langle b\rangle\} \qquad \text{for each } \ell \in \{a\}^* \\
\mathrm{infer}_Q\, \{b\}\, \langle\rangle &= Q@\{a\} \\
\mathrm{infer}_Q\, \{b\}\, \langle b\rangle &= (Q@\{a\})\,\hat{}\,\{\langle b\rangle\}
\end{aligned}$$

The sets of inferences are at least as big as those for R given in example 14. A user viewing Q through $\{b\}$ cannot tell how many a's have occurred; all it knows is that no more occur after it has seen one b. △

Example 16 Finally, for P of example 12, we calculate:

$$\begin{aligned} \text{infer}_P\,\{a\}\,\langle\rangle &= P@\{b\} \\ \text{infer}_P\,\{a\}\,\langle a\rangle &= \{\langle a\rangle\} \\ \text{infer}_P\,\{b\}\,\langle\rangle &= P@\{a\} \\ \text{infer}_P\,\{b\}\,\langle b\rangle &= \{\langle b\rangle\} \end{aligned}$$

△

4.2.2 The security ordering

We have already remarked (in a comment following lemma 38) that inference functions are equivalent to traces. Why then should we introduce such a description of a system? The function $\text{infer}_S\,B$ tells us what a user viewing the system through B can infer about the past of the system in a more direct manner than the traces. It is also the basis for the definition of an ordering on systems. As we have noted in chapter 2, sets of traces give rise to a natural *safety ordering*: a system S is at least as safe as a system R, written $S \sqsupseteq R$, if $\tau R \supseteq \tau S$. We now develop a similarly natural *security ordering*.

Consider examples 13 and 14; we have:

$$\text{infer}_S\,\{a\}\,\langle\rangle = \{\langle\rangle\} \subset \{\langle\rangle, \langle b\rangle\} = \text{infer}_R\,\{a\}\,\langle\rangle$$

A user viewing R through $\{a\}$ is less sure of the behaviour of the system having observed the trace $\langle\rangle$ than a similar user observing S is. In the latter case the behaviour of the system is known exactly, but in the former there are two possibilities. It is in this sense that we say R is more secure than S through $\{a\}$ at observation $\langle\rangle$. Generally, we say:

Definition 30 *R is at least as secure as S through B at observation ℓ, written $R \succeq_{B,\ell} S$, if:*

$$\ell \in S@B \cap R@B \;\wedge\; \text{infer}_S\,B\,\ell \subseteq \text{infer}_R\,B\,\ell$$

Similarly we say R is at least as secure as S through B, written $R \succeq_B S$, if it is at least as secure at every observation:

$$\forall\,\ell : R@B \;\cdot\; R \succeq_{B,\ell} S$$

◇

Note that the second case implies $R@B \subseteq S@B$.

Lastly, we can state that one system is more secure than another through several sub-alphabets.

Definition 31 *Let $\mathcal{A}$ be a set of subsets of alphabet A. We say R* is at least as secure as *S* through *$\mathcal{A}$, written $\succeq^{\mathcal{A}}$, if it is at least as secure through every member of A:*

$$\forall B : \mathcal{A} \cdot R \succeq_B S$$

◇

Often the set of subsets will be a disjoint partition of the alphabet, with each subset being identified with a different user's interface to the system. We use the symbol $\asymp$, with the appropriate subscripts and superscripts, if both $\succeq$ and $\preceq$ hold. Similarly, we use $\succ$ if $\succeq$ holds, but not $\asymp$.

Before investigating some of the properties of $\succeq$ we give an example of its use:

Example 17 From examples 13–16, and with respect to (the disjoint partition) $\mathcal{A} = \{\{a\}, \{b\}\}$ of $\{a, b\}$ we see:

$$Q \succ^{\mathcal{A}} R \succ^{\mathcal{A}} S$$

and

$$\begin{array}{lll} R & \succ^{\mathcal{A}} & P \\ S & \prec_{\{a\},\langle\rangle} & P \\ S & \succ_{\{a\},\langle a\rangle} & P \\ S & \asymp_{\{b\},\langle\rangle} & P \end{array}$$

S and P are incomparable at $(\{b\}, \langle b\rangle)$. △

We have two simple identities:

Lemma 40 *For any systems S and R with alphabet A:*

$$\begin{array}{rcl} S \succeq_{\{\}} R & \iff & \tau S \supseteq \tau R \\ S \succeq_A R & \iff & \tau S \subseteq \tau R \end{array}$$

Proof Direct from definition 30. □

There is no simple relation between $\sqsupseteq$ and $\succeq$. Both $S@B$ and $\text{infer}_S\, B\, \ell$ as functions of S are decreasing with respect to $\sqsupseteq$, that is:

$$S \sqsupseteq R \Longrightarrow (R@B \supseteq S@B) \wedge (\text{infer}_R\, B\, \ell \supseteq \text{infer}_S\, B\, \ell)$$

While $S@B$ as a function of S is decreasing with respect to $\succeq_B$, $\text{infer}_S\, B\, \ell$ is increasing with respect to $\succeq_{B,\ell}$:

$$\begin{aligned} & S \succeq_B R \Longrightarrow R@B \supseteq S@B \\ \wedge \quad & S \succeq_{B,\ell} R \Longrightarrow (\text{infer}_R\, B\, \ell) \subseteq (\text{infer}_S\, B\, \ell) \end{aligned}$$

The next examples illustrate this point:

Example 18 Let $A \triangleq \{a, b\}$, then

$$STOP_A \sqsupset (a \rightarrow STOP_A)$$

However we also have

$$(a \rightarrow STOP_A) \succ_{\{b\}} STOP_A$$

and

$$STOP_A \succ_{\{a\}} (a \rightarrow STOP_A)$$

The former follows as

$$STOP_A@\{b\} = (a \rightarrow STOP_A)@\{b\} = \{\langle\rangle\}$$

and

$$(\text{infer}_{a \rightarrow STOP_A}\, \{b\}\, \langle\rangle) \supset (\text{infer}_{STOP_A}\, \{b\}\, \langle\rangle)$$

The latter follows as

$$(a \rightarrow STOP_A)@\{a\} \supset STOP_A@\{a\} = \{\langle\rangle\}$$

and

$$(\text{infer}_{a \rightarrow STOP_A}\, \{a\}\, \langle\rangle) = (\text{infer}_{STOP_A}\, \{a\}\, \langle\rangle)$$

$\triangle$

Example 19 For the systems S and R of examples 9 and 10 we have $S \sqsupset R$. Only because $S@\{a\} = R@\{a\}$ and $S@\{b\} = R@\{b\}$ do we also have $R \succ^{\{\{a\},\{b\}\}} S$. $\triangle$

In fact, $\succeq^{\mathcal{A}}$ has neither a top nor a bottom, in general.

Lemma 41 *If $\mathcal{A}$ has at least two non-empty disjoint members $\succeq^{\mathcal{A}}$ has no top*

Proof *STOP* is the only solution for P that minimizes the domain of $\text{infer}_P\, B$ for each $B \in \mathcal{A}$, and so is the only candidate for top. However, from example 14 we see that

$$R \succ_{\{a\},\langle\rangle} STOP_{\{a,b\}}$$

and that

$$STOP_{\{a,b\}}@\{a\} \subset R@\{a\}$$

That is, R and *STOP* are incomparable with respect to the order $\succeq^{\{\{a\},\{b\}\}}$. □

Lemma 42 *If $\mathcal{A}$ has at least two non-empty disjoint members $\succeq^{\mathcal{A}}$ has no bottom*

Proof Similarly to the previous argument, *RUN* is the only candidate for bottom, on the grounds that it is the only candidate for P that maximizes the domain of $\text{infer}_P\, B$. S of example 13 is not comparable with $RUN_{\{a,b\}}$ with respect to the order $\succeq^{\{\{a\},\{b\}\}}$. □

4.3 The specification of security properties

As an example of the use of infer to specify security we specify the security properties of systems discussed in [BLP76]. These are the externally observable properties of systems used by the military, and *not* the ss-property, the *-property and the ds-property which certain types of implementation must satisfy in order to enforce the external properties. We use the same names as [BLP76] do for the section headings; however, we have renamed the properties themselves.

4.3.1 The Military Multi-Level Security Scheme

A good description of the military multi-level security scheme is given in [Lan81]. The scheme has a number of *levels*, often given names such as TOP SECRET, SECRET, UNCLASSIFIED etc., and an ordering on the levels.

Every user of a system and every piece of information (or, rather, every *document*) is given a security level. A low-level user is not allowed to know information that is at a higher level. In addition to the security levels there are *topics*, each of which covers a different area of interest (for example, groups might be ATOMIC, COMMUNICATIONS and SDI). Every user has a set of topics. Information can only flow from one user to another if the receiving user has all of the sender's topics in his set.

The complete multi-level system employs both of these mechanisms to ensure that information does not flow. Together they generate a pre-order on users. A pair (B, C) is in the pre-order if, and only if, the level of B is no higher than the level of C and the topics of interest to B are a subset of the topics of interest to C. It is a theorem that every pre-order can be expressed in terms of levels and topics in this way (a version of this theorem is proved in, for example, [Ru85a]). In what follows we work directly with pre-orders on users, or rather, on the interfaces to users. We use the symbol $\rightarrowtail$ to stand for an arbitrary pre-order.

4.3.2 Mandatory security

An important security property is called *mandatory security* in [BLP76]. It is called *mandatory* because the restrictions on information flow are not under the control of the users of the system, but are imposed from outside. The particular restriction [BLP76] give this name to is one which can be represented by a pre-order or a military multi-level security scheme.

Example 20 Let F be the interface to a file, and B be the interface to a user. If $B \rightarrowtail F$ then the user is not prohibited from deducing some facts about the history of the file. The user is not guaranteed to be able to find out *all* about the file; for example the user may only be able to find out the size of the file, rather than its contents. If $B \not\rightarrowtail F$ the user is prohibited from deducing anything about the file: not its size, nor its contents, nor its date of last access, etc. △

As a user with interface B potentially has access to all the knowledge gained by the users lower than it in the pre-order; that is: the knowledge it has of the system comes from an enlarged window. We write this enlarged window $\lceil B \rceil$, and define it:

$$\lceil B \rceil \triangleq \bigcup \{C \mid B \rightarrowtail C\}$$

Definition 32 *A system S* restricts information flow *w.r.t.* $(\mathcal{A}, \rightarrowtail)$ *if:*

$$\forall B : \mathcal{A},\ \ell : S@B \cdot \lceil B \rceil^* \cap (\text{infer}_S\ B\ \ell) \neq \{\}$$

$\diamond$

This condition asserts that whatever observation is made through B, it is possible that the system has not engaged in any events outside $\lceil B \rceil$. That is, although the user viewing the system through B may know what behaviours the other users are restricted to, it cannot know how much of those behaviours have occurred. As nothing may have occurred and each behaviour starts from nothing, there is no information in ℓ to distinguish them. This condition is related to the condition of *noninterference* of [GM82] and [Ru85a]; we will discuss this further in chapter 5.

Example 21 From example 13 we see that the system S of example 9 enforces restriction of information flow with respect to the disjoint partition $\{\{a\}, \{b\}\}$ and the pre-order defined as the transitive, reflexive closure of $\{\{b\} \mapsto \{a\}\}$. $\triangle$

We now show that increasing the security of a system that enforces restriction of information flow preserves the restriction.

Lemma 43 *Let S and R be systems with alphabet A, such that*

(H1) *S enforces restriction of information flow with respect to $(\mathcal{A}, \rightarrowtail)$.*

(H2) *$R \succeq^{\mathcal{A}} S$.*

Then

(D) *R enforces restriction of information flow with respect to $(\mathcal{A}, \rightarrowtail)$.*

Proof Let (H3) $B \in \mathcal{A}$ and $\ell \in R@B$; then:

(1)	$\ell \in S@B$	(H2), (H3)
(2)	$\lceil B \rceil^* \cap (\text{infer}_S\ B\ \ell) \neq \{\}$	(H1), (1)
(3)	$\lceil B \rceil^* \cap (\text{infer}_R\ B\ \ell) \neq \{\}$	(H2), (H3), (2)
(D)		(H3), (3)

$\square$

Example 22 From lemma 43, example 17 and example 21 we see that R of example 10 restricts information flow, with respect to the disjoint partition $\{\{a\},\{b\}\}$ and pre-order defined as the reflexive closure of $\{\{b\} \mapsto \{a\}\}$. Of course, we could have deduced this directly from example 14 $\triangle$

A special case of restriction of information flow is complete isolation.

Definition 33 *Let* id *be the identity relation (and so a pre-order). We say a system* enforces isolation with respect to $\mathcal{A}$ *if it enforces restriction of information flow with respect to* $(\mathcal{A}, \mathrm{id})$ $\diamond$

Example 23 The system

$$(\mu X : \{a\} \cdot a \longrightarrow X) \;|||\; (\mu X : \{b\} \cdot b \longrightarrow X)$$

enforces isolation with respect to $\{\{a\},\{b\}\}$. $\triangle$

4.3.3 Trusted Users

The requirement on a system that it must restrict information flow with respect to some partition and pre-order can be relaxed in the presence of *trusted* users[1]. A trusted user is one whose good behaviour the system need not enforce. There may be observations through the trusted user's interface which reveal facts about the system behaviour. We can trust a user with interface B, if it has been proved never to attempt to engage in any of these observations. That is, it never engages in observations $\ell \in B^*$ such that

$$\lceil B \rceil^* \cap (\mathrm{infer}_S\, B\, \ell) = \{\}$$

This is a sufficient condition to trust a low classified user; in the absence of any constraints on the behaviour of the higher users it is also necessary.

We may also trust a low user if no higher user ever engages in a detectable behaviour. Suppose the higher users are represented by (the CSP process) H, then we need to prove that $S \parallel H$ restricts flow of information, with respect to $(\mathcal{A}, \rightarrowtail)$. This condition and the previous one can be combined in the obvious way: a low user can be trusted if it never tries to observe one of the detectable behaviours of the higher users. That is, we

[1] [BLP76] uses the term *trusted subjects.*

can trust the user with interface B if it never engages in a trace $\ell \in B^*$ such that

$$\lceil B \rceil^* \cap (\text{infer}_{(S \| H)}\, B\, \ell) = \{\}$$

Example 24 Let $A \triangleq \{a1, a2, b\}$ and $\mathcal{A} \triangleq \{\{a1, a2\}, \{b\}\}$ and pick a pre-order $\rightharpoonup$ that satisfies $\{a1, a2\} \not\rightharpoonup \{b\}$. Then the system

$$S \triangleq (b \longrightarrow a1 \longrightarrow STOP) [\!] (a2 \longrightarrow b \longrightarrow STOP)$$

does not restrict of information flow with respect to $(\mathcal{A}, \rightharpoonup)$ as

$$(\text{infer}_S\, \{a1, a2\}\, \langle a1 \rangle) = \{\langle b, a1 \rangle\}$$

and

$$\langle b, a1 \rangle \notin \lceil \{a1, a2\} \rceil^*$$

It is easy to see that $\langle a1 \rangle$ is the only insecure trace, so for a user U (with interface $\{a1, a2\}$) to be trusted we must prove $\langle a1 \rangle \notin \tau U$. An example of such a user is:

$$a2 \longrightarrow STOP_{\{a1, a2\}}$$

$\triangle$

A good way of making a system S restrict information flow to B is to find a user U with interface B that can be trusted completely, and replacing S by $S \| U$. This system does not allow a user with interface B to engage in any traces which violate the security constraint. The most general U is defined as that deterministic CSP process with:

$$\tau U = \{\ell : S@B \mid \lceil B \rceil^* \cap (\text{infer}_S\, B\, \ell) \neq \{\}\}$$

We should show that this set actually defines the traces of a real process. It is non-empty by lemma 39. For prefix-closure we remark that: if $v \in \lceil B \rceil^* \cap (\text{infer}_S\, B\, (s \hat{} t))$ then $v \restriction B = s \hat{} t$; we may split v into two, $v = w \hat{} x$ with $w \restriction B = s$ and $x \restriction B = t$; it is not too hard to see that $w \in \lceil B \rceil^* \cap (\text{infer}_S\, B\, s)$.

Example 25 With the system S and safe user U of example 24, we can calculate a secure system, S':

$$\begin{aligned} S' &= S \| U \\ &= (b \longrightarrow STOP_A) [\!] (a2 \longrightarrow b \longrightarrow STOP_A) \end{aligned}$$

Of course, S' may not possess all the desirable properties that S does. $\triangle$

4.3.4 Integrity

The next property we specify is *integrity*. It is a dual to the restriction of information flow discussed above. Whereas we were concerned to say that the future behaviour of a low-classed user cannot be influenced by the past behaviour of a high-classed user, we now want to say that the past behaviour of a low-classed user cannot influence the future behaviour of a high-classed user. This is almost like restriction of "downward" information flow, but now we are concerned with preventing "upward" flow. The same mathematical structure will do.

Example 26 Let F be the interface to a high-security file, and B be the interface to a low-security process. Then we ensure the integrity of the file by enforcing restriction of information flow with respect to a pre-order $\rightarrowtail$ that satisfies $F \not\rightarrowtail B$. $\triangle$

When we discuss integrity it is perhaps more intuitive to talk about restriction of *instruction* flow, rather than *information* flow. However, there is no difference in the mathematics as we are not attaching directions or meanings to events in the model.

4.3.5 Discretionary security

In contrast to mandatory security, discretionary security is under the control of the users. Typically one user will give and withdraw permission for another user to discover its past behaviour. Many ways of organising the users' control of the permission is possible and the formulation given below is somewhat arbitrary. It should be treated as a model around which other formulations can be tailored.

Suppose that B is the interface to one user, C that to another, and $B \rightarrowtail C$ (so that it is not forbidden for B to discover something about C). C contains, among others, the two events "on_{CB}" and "off_{CB}". The occurrence of "off_{CB}" marks the removal from B of its rights to knowledge of C; "on_{CB}" marks their restoration. We can partition B into three disjoint sets: while the privilege of knowledge is granted B can engage in events from BC_{on}, while the privilege is withdrawn B can engage in events from BC_{off} and it is always allowed to engage in events from BC_{both}.

Example 27 Let C be the interface to a file called "f" and B be the interface to some user of the file. B can always engage in the events from the set

$$BC_{both} \triangleq \{f.write.d, f.read, \ldots \mid d \in DATA\}$$

where the first event is that of requesting a write to the file, to which it expects a result indicating success or failure, and the second is that of requesting a value to be read from the file, to which it expects either a value drawn from $DATA$ or a failure message. The events indicating these successes are members of the set

$$BC_{on} \triangleq \{f.success, f.value.d \mid d \in DATA\}$$

The first is received following a successful write, the others following successful read requests. When permission to access the file is denied the single failure message *f.permission-denied* is received; this event is the only member of the set BC_{off}.

The file engages in events on_{CB} and off_{CB} after messages from its owner. △

Definition 34 *A system, S,* enforces discretionary security *if:*

$$\forall C : \mathcal{A} \cdot (\text{infer}_{S'} B\, u) \cap \{s : (\lceil B \rceil - C)^* \mid s \upharpoonright \{on_{CB}\} = \langle\rangle\} \neq \{\}$$

for all S', t and u such that

$$\begin{aligned} S' &= S/(t \hat{} \langle off_{CB} \rangle) \\ u &\in (S/t \hat{} \langle off_{CB} \rangle)@B \cap (B - BC_{on})^* \end{aligned}$$

◇

This condition is very like that of mandatory security. However, there are two principal differences. Firstly, we are only concerned with the system's behaviour since the last time that it was known that knowledge of C was allowed (that is, S/t). Secondly, under mandatory security *nothing* can be known about C by B, but here B knows that the privilege of (detailed) knowledge has been withdrawn when it sees the first event from BC_{off}. There is a connection between this definition and that of *conditional non-interference* [GM84,Ru85a], which we discuss further in chapter 5.

4.3.6 Communication Paths

This section discusses a topic which lies beyond the scope of the model of [BLP76]. It is the detection of *covert channels* between users. To quote [BLP76]:

> By [the problem of covert channels] is meant the indirect disclosure of sensitive information, as opposed to the direct disclosure of information. [...] Indirect disclosure can be effected by transmitting data piecemeal using observable system characteristics as the code medium.

A covert channel is a suitable coding of system characteristics.

In [BLP76] covert channels are divided into two classes, named "synchronous" and "nonsynchronous". For synchronous channels they assert:

> Possibly the most difficult medium to rule out as a communication path is real time: intervals of real time, delimited by *prearranged* observable events, [...] can be used to transmit information in bit strings.

The model of [BLP76] does not help in showing absence of such channels. Our formulation does not help either, because traces abstract away from the time between events. If we moved from the model of CSP in [BrRo85] to the model of *Timed* CSP in [ReRo86] then there is hope that an analogous theory based on *timed-traces* could be developed. This is not done here.

[BLP76] offer little solace to those concerned with nonsynchronous channels:

> Indirect communication using nonsynchronous paths remains a very complicated problem.

(Where "paths" is a synonym for channels.) The problem lies more in the proof that a particular system has the desirable property than in specifying the property. The covert channel may involve subtle combinations of shared data, system variables and the like. Proof of correctness involves

> [...] close and careful consideration of every possible action [of the] system.

The specification is easy in our formulation, but the proof is just as hard.[2] As an example we will give a specification for (a paraphrase of) one of the examples [BLP76] use to illustrate this problem.

Example 28 A system has internal state given by an array of n Boolean variables. Interface B contains events to set and unset the variables: $i.set$ and $i.unset$ for each $i \in \{1,\ldots,n\}$. Interface C contains events to read the value of the variables: $i.val.T$ and $i.val.F$ for each $i \in \{1,\ldots,n\}$. The desired security is specified by demanding that the system restrict flow of information with respect to a pre-order, $\rightarrow$, for which $C \not\rightarrow B$.

When this specification is completed it may not be possible to find a system that satisfies it together with other demands which may be made in the full specification, for example: the system must always correctly and immediately obey requests to reveal or change its state. $\triangle$

4.4 Generalised security specifications

The properties we have specified above are useful in many cases, but sometimes they are too gross for some purposes. In particular, they do not allow us to state that some, but not all, information can flow from B to C. Consider the next example.

Example 29 A system that fills in tax forms for its customers is to be installed. A customer enters his personal details, which we suppose drawn from the set D, and a short while later receives a correctly filled in tax form. We suppose that for each collection of personal details, $d \in D$ the form is given by the expression $\mathrm{fill}\, d$, for some function fill. The interface to the i^{th} customer (for i in some set of names I) is given by:

$$A_i \;\widehat{=}\; \{i.req.d, i.form.f \mid d \in D \wedge f \in \mathrm{ran}\,\mathrm{fill}\}$$

We require of the system that it always gives the correct answer. This can be specified using the notation of [Ho85].

$$S@A_i \subseteq \{t : A_i^* \mid t \downarrow i.form \leq^1 \mathrm{fill}^*(t \downarrow i.req)\}$$

[2]The proof may be eased by the many techniques being developed for the specification, verification and refinement of CSP processes. See for example [WH86].

However, there is a catch: the service is not free. Each customer is charged for each use of the system. In order to do this there is a clerk whose interface is just the names of the users, that is: I. Sometime after a customer obtains a form his name is printed on the clerk's terminal, and the clerk then prepares and sends a bill.

Problem: we must specify that no user, customer or clerk, can discover another user's personal details. If we insist that isolation is enforced with respect to $\{A_i \mid i \in I\} \cup \{I\}$, the clerk is prevented from knowing how many times customers have used the system. If we replace the identity relation implicit in isolation by the reflexive, transitive closure of $\{I \mapsto A_i \mid i \in I\}$ then the clerk (but no-one else) could discover something about a customer's behaviour. This specification is satisfied by a system that sent the clerk a user's details coded as a bit stream, where one user's name is chosen for 0 and another's for 1[3].

The real problem: how can we specify a limit to the amount of information the clerk can obtain? △

We could solve the problem raised by example 29 by finding (an inference function of) a CSP process that was "obviously correct", and insisting that any candidate solution must be more secure than this. Such an inference function contains much unnecessary information; for example it contains the sets of inferences that may be made from observing the system partly through one user's interface and partly through another's. We can generalise inference functions to avoid this inconvenience and we name the generalisations *security specifications*. A security specification is an inference function, but omitting irrelevant information. The generalisation has a similar relation to inference functions that predicates with a free trace variable (equivalent to sets of traces) have to prefix-closed sets of traces. Where a predicate with a free trace variable specifies a lower limit of safety , a security specification gives, for each window and observation through that window, an upper limit on the inferences that can be made. A security specification must satisfy some consistency properties:

[3]Of course this method only works if there are at least two customers. With only one customer the time between events can be used to code the bits. As already mentioned, we would need timed traces to analyse this.

Definition 35 *A* security specification over an alphabet A *is a partial function* f *of type* $\mathcal{P}A \to A^* \to \mathcal{P}A^*$ *which, for each* $B, C \in \mathrm{dom}\, f$ *and* $\ell \in \mathrm{dom}\,(f\ B)$, *satisfies the properties:*

1. $\mathrm{dom}\,(f\ B)$ *is a prefix-closed subset of* B^*
2. $\forall t : (f\ B\ \ell) \cdot t \upharpoonright B = \ell$
3. $\forall t : (f\ B\ \ell) \cdot t \upharpoonright C \in \mathrm{dom}\,(f\ C)$

◇

The first condition ensures that the observations made through B form a coherent set; that is, it represents a possible view through B. The second states that each trace in '$f\ B\ \ell$' projects only onto the relevant local observation. The last insists that if f allows B to infer that C has observed s, then f legislates on the maximum that can be inferred by C when observing s.

Example 30 For the alphabet $\{a, b\}$, partitioned into $\{a\}$ and $\{b\}$, we want to specify that nothing can be discovered through $\{b\}$ about the rest of the system, other than it has started. No restriction is placed on knowledge gained through $\{a\}$. We give the specification, f:

$$\begin{array}{rcll} f\,\{a\}\,s & = & \{\} & \text{for each } s \in \{a\}^* \\ f\,\{b\}\,\langle\rangle & = & \{\langle\rangle, \langle a\rangle\} & \\ f\,\{b\}\,(\langle b\rangle \mathbin{\hat{}} s) & = & \{\langle a, b\rangle \mathbin{\hat{}} s\} & \text{for each } s \in \{b\}^* \end{array}$$

The first clause says that *no* restrictions are placed on $\{a\}$. The next two give the restrictions on $\{b\}$: after the initial b has been seen all that can be deduced is that at least one a has occurred. △

Note a security specification may return $\{\}$ for some combinations of its parameters. This is never true for an inference function.

Our next lemma shows that security specifications are indeed generalisations of inference functions.

Lemma 44 *For any system* S, infer_S *is a security specification over* αS.

Proof infer_S is a total function of type $\mathcal{P}A \to A^* \to \mathcal{P}A^*$, so it is a partial function of the correct type. Now we check the conditions of definition 35:

1. By lemma 1, as $\mathrm{dom}\,(\mathrm{infer}_S\, B) = S@B$.
2. Directly from the definition of infer_S.
3. Follows as $\mathrm{dom}\,(\mathrm{infer}_S\, C) = S@C$.

□

The security orderings on systems given above are really orderings on the inference functions. We can extend these orderings to security specifications:

Definition 36 *Let f and g be two security specifications over A, then f is at least as secure as g, written $f \succeq g$, if, for each $B \in \mathrm{dom}\, f$ and $\ell \in \mathrm{dom}\,(f\, B)$:*

1. $\mathrm{dom}\, f \supseteq \mathrm{dom}\, g$
2. $\mathrm{dom}\,(f\, B) \subseteq \mathrm{dom}\,(g\, B)$
3. $(f\, B\, \ell) \supseteq (g\, B\, \ell)$

◇

The principal difference between this and the ordering $\succ^A$ on systems is the first condition. We do not need this as infer_S is always total. Now we show that the ordering on inference functions is an extension of that on systems:

Lemma 45 *Let S and R be two systems with alphabet A, then*

$$S \succeq^{PA} R \iff \mathrm{infer}_S \succeq \mathrm{infer}_R$$

Proof Direct from definitions 31 and 36. □

Unlike the restricted space of inference functions, there are top and bottom security specifications.

Definition 37 *For an alphabet A define:*

$$\begin{aligned} \bot_A &\triangleq \emptyset \\ \top_A &\triangleq K\emptyset \end{aligned}$$

where $\emptyset$ is the empty function and K is the constant combinator. ◇

Lemma 46 *$\top_A$ is the top of the order $\succeq$, and $\bot_A$ is its bottom.*
Proof The first claim follows as $K\emptyset$ is total and $\mathrm{dom}\,\emptyset = \{\}$, the second as $\mathrm{dom}\,\emptyset = \{\}$. □

Neither $\top_A$ nor $\bot_A$ correspond to systems.

Not only do security specifications have a top and a bottom, but they form a complete lattice.

Definition 38 *Let F be a set of security functions. Define:*

$$(\inf F)\, B\, \ell \triangleq \bigcap_{f:F} f\, B\, \ell$$

for each $B \in \bigcap_{f:F} \mathrm{dom}\, f$ and $\ell \in \bigcup_{f:F} \mathrm{dom}\,(f\, B)$. Similarly, define:

$$(\sup F)\, B\, \ell \triangleq \bigcup_{f:F} f\, B\, \ell$$

for each $B \in \bigcup_{f:F} \mathrm{dom}\, f$ and $\ell \in \bigcap_{f:F} \mathrm{dom}\,(f\, B)$. ◇

Lemma 47 *For any set of security specifications F the greatest lower and least upper bounds exist and are* $\inf F$ *and* $\sup F$ *respectively.*
Proof Follows as $\bigcap X$ and $\bigcup X$ exist for all sets of sets X and are the greatest lower and least upper bounds in the $\supseteq$ ordering. □

We use the order to define the satisfaction relation between security specifications and systems.

Definition 39 *A system, S,* satisfies a security specification, *f, written $S \succeq f$ if:*

$$\mathrm{infer}_S \succeq f$$

◇

Of course, no system can satisfy $\top_A$.

Example 31 All of the following systems satisfy the specification of example 30:

$$a \longrightarrow b \longrightarrow STOP$$
$$a \longrightarrow b \longrightarrow (\mu X : \{a, b\} \cdot a \longrightarrow X)$$
$$a \longrightarrow ((\mu X : \{a\} \cdot (a \longrightarrow X)) \;|||\; (b \longrightarrow STOP))$$

△

We can easily transfer the definitions of the specific properties from inference functions to specifications. For example, we will redefine restriction of information flow:

Definition 40 *A security specification over A, f* restricts information flow w.r.t. $(\mathcal{A}, \rightarrowtail)$ *if:*

$$\mathcal{A} \subseteq \mathrm{dom}\, f$$

and

$$\forall B : \mathcal{A}, \ell : (\mathrm{dom}(f\, B)) \cdot \lceil B \rceil^* \cap (f\, B\, \ell) \neq \{\}$$

◇

We have added the condition that f is defined everywhere in $\mathcal{A}$, as f need not be total, and replaced $S@B$ by $\mathrm{dom}(f\, B)$ and infer_S by f. Also, similar to lemma 43 we can prove:

Lemma 48 *Let f and g be security specifications over A, such that*

(H1) *f enforces restriction of information flow with respect to $(\mathcal{A}, \rightarrowtail)$.*

(H2) *$g \succeq f$.*

Then

(R) *g enforces restriction of information flow with respect to $(\mathcal{A}, \rightarrowtail)$.*

□

We can proceed similarly with the other properties.

Finally we specify the security for the tax service (example 29).

Example 32 The security specification for the tax service is a function f with:

$$\mathrm{dom}\, f \;\widehat{=}\; \{A_i \mid i \in I\} \cup \{I\}$$

Now we must give the value of f for each point in its domain. First we do this for an arbitrary customer. The value of f at the i^{th} customer is given by:

$$\mathrm{dom}(f\, A_i) \;\widehat{=}\; \{t : A_i^* \mid t \downarrow i.form \leq^1 \mathrm{fill}^*(t \downarrow i.req)\}$$

That is, f is defined for any number of uses of the service by the i^{th} customer. For each point in this domain:

$$f\, A_i\, \ell \;\widehat{=}\; \{\ell\}$$

This allows the i^{th} customer the minimum knowledge—that at any stage it is possible that only it has engaged in any events.

The specification for the clerk is:

$$\begin{aligned} \mathrm{dom}\,(f\, I) &\;\widehat{=}\; I^* \\ f\, I\, \langle\rangle &\;\widehat{=}\; ONE \\ f\, I\, (\ell^\frown \langle i \rangle) &\;\widehat{=}\; (\max i\, (f\, I\, \ell))^\frown_{-} \{\langle i \rangle\}^\frown_{-} ONE \end{aligned}$$

where the set ONE is the set of traces of a single use of the service:

$$ONE \;\widehat{=}\; \{t : A^* \mid \exists i : I, d : D \cdot t \leq \langle i.req.d, i.form.\mathrm{fill}\, d \rangle\}$$

and $(\max i\, T)$ is the set of longest traces drawn from T that end in a transaction with customer i:

$$\max i\, T \;\widehat{=}\; \{t : T \mid (\forall s : T \cdot \#s \leq \#t) \wedge (\exists d : D \cdot \bar{t}_0 = i.form!\mathrm{fill}\, d)\}$$

By specifying these limits to the inferences that the clerk can make we have prevented any coding of a customer's personal details: *all* that the clerk can be sure of is that he will know a customer has used the service sometime after the customer has done so. The specification f states that the clerk *may* know a customer has used the service immediately after the customer has received his form; $(f\, I\, \ell)$ is a minimum set—in an implementation it may be larger, so that the clerk may be informed of a consultation a long time after it has occurred.

Suitable implementations that satisfy the security specification f are:

$$\mu X : A \cdot [\!]_{i:I}\, i.req?d : D \longrightarrow i.form!\mathrm{fill}\, d \longrightarrow i \longrightarrow X$$

and

$$|||_{i:I}\ (\mu X : A_i \cdot i.req?d : D \longrightarrow i.form!\mathrm{fill}\, d \longrightarrow i \longrightarrow X)$$

The first lets the clerk know of a consultation immediately after it has occurred. The second allows consultations with other customers to occur before informing the clerk. △

Chapter 5

Related work

5.1 Local refinement

5.1.1 A CCS approach

Whatever notions are developed in CSP there is parallel work in CCS [Mi80], and *vice versa*. In this case, the parallel idea is that of *Context Dependent Bisimulation* [Lar86,LM86].

The idea is that to achieve a given *SPEC*, a design of the form

$$C[SUBSPEC]$$

is used, with the specification and design equivalent in some sense. As [Lar86] is working in CCS this sense is bisimulation equivalence. To implement *SUBSPEC* by *SUBIMPL* the two have to be proved equivalent in some, possibly different, sense. This second equivalence need not be bisimulation equivalence as the context for the sub-specification and sub-implementation, $C[\,]$, cannot discriminate between some behaviours. All that is needed is a bisimulation relative to the given context.

A slightly weaker notion is equivalence with respect to an environment, an environment being a context that only absorbs actions (events) from the set *Act* but never produces any. An environment can be defined in a similar way to a CCS agent (process), and is, in effect, a CCS agent that is composed in parallel with the agents under consideration. Let $E \stackrel{v}{\Rightarrow} E'$ mean that environment E can absorb action string v and then behave like environment E'. A bisimulation relative to an environment E is a family

of relations R_E, one for each possible state of the environment E, such that whenever $P\, R_E\, Q$:

$$\begin{aligned}
&\forall v : Act^* \cdot \\
&\quad E \stackrel{v}{\Rightarrow} E' \Longrightarrow \\
&\qquad (P \stackrel{v}{\Rightarrow} P' \Longrightarrow \exists Q' \cdot Q \stackrel{v}{\Rightarrow} Q' \wedge P'\, R_{E'}\, Q' \\
&\qquad \wedge Q \stackrel{v}{\Rightarrow} Q' \Longrightarrow \exists P' \cdot P \stackrel{v}{\Rightarrow} P' \wedge Q'\, R_{E'}\, P')
\end{aligned}$$

If a bisimulation between P and Q exists with respect to environment E the fact is recorded: $P \sim_E Q$. This definition coincides with the ordinary definition when the environment is the universal environment, U, which can accept *any* string of actions, after which it behaves like itself:

$$\forall v : Act^* \cdot U \stackrel{v}{\Rightarrow} U$$

The definition of relative bisimulation is rather like the CSP definition:

$$P \sim_E Q \;\hat{=}\; P \| E = Q \| E$$

as P and Q are only compared for those executions that E is prepared to agree to; they may differ when E prevents both from progressing.

In our work we have been interested in environments that are constructed from independent users in parallel. We could try and construct the most general such environment in CSP as

$$ICE' \;\hat{=}\; \sqcap \{\|_{B:A} P_B \mid P_B \sqsupseteq CHAOS_B\}$$

Unfortunately this is equivalent to $CHAOS_{\bigcup A}$, and as $CHAOS$ is a zero of (intersection) concurrency, makes all processes equal. Nor is the upper bound,

$$ICE'' \;\hat{=}\; \bigsqcup \{\|_{B:A} P_B \mid P_B \sqsupseteq CHAOS_B\}$$

of any use, as it does not exist for CSP processes, in general. What is needed is equivalence or refinement when placed in parallel with each of the members of

$$ICE \;\hat{=}\; \{\|_{B:A} P_B \mid P_B \sqsupseteq CHAOS_B\}$$

That is,

$$NEW \trianglerighteq_A OLD \Longleftrightarrow \forall P : ICE \cdot NEW \| P \sqsupseteq OLD \| P$$

(This is essentially lemma 3, page 26.) Thus our definition is subtly different to that of [Lar86].

5.1.2 Transaction Processing

A relation similar to local refinement can be used to define suitable transaction processing machines in an abstract and concise way. Let A be a set of events and $\mathcal{A}$ be a disjoint partition of A. Each member of $\mathcal{A}$ represents the sub-alphabet of one transaction.

Transaction processing systems are supposed to execute the transactions presented to them in an efficient way, while preserving the property of serializeability [P79]. This property states that when all the transactions have completed it is possible to assign a serial order in which they *appear* to have occurred. It need not be an order possible on the given system, but the illusion of such an order must be maintained. We can easily specify a machine which only executes the transactions in a serial order:

$$SERIAL_{\mathcal{A}} \mathrel{\widehat{=}} \forall B : \mathcal{A}, a, b : B, s : A^* \cdot (\langle a \rangle \mathbin{\hat{}} s \mathbin{\hat{}} \langle b \rangle) \text{ in } tr \Longrightarrow s \in B^*$$

When looking at a trace of a concurrent transaction processing machine, it is necessary to know which transactions have terminated. In order to record this we take $END \subseteq A$, with the property that $\forall B : \mathcal{A} \cdot B \cap END \neq \{\}$. A transaction is only allowed to engage in one END event, and this must be its last event. We can capture this:

$$ONE\text{-}FINAL_{\mathcal{A}} \mathrel{\widehat{=}} \forall B : \mathcal{A}, s : A^*, e : B \cap END \cdot (\langle e \rangle \mathbin{\hat{}} s) \text{ in } tr \Longrightarrow s \in (A - B)^*$$

For a given $\mathcal{A}$, $SERIAL_{\mathcal{A}} \wedge ONE\text{-}FINAL_{\mathcal{A}}$ is a safety specification; in general there will be extra liveness constraints, which we ignore for our present purposes. For shorthand, we define:

$$BASIC_{\mathcal{A}} \mathrel{\widehat{=}} SERIAL_{\mathcal{A}} \wedge ONE\text{-}FINAL_{\mathcal{A}}$$

Any machine which gives the illusion that it satisfies $BASIC_{\mathcal{A}}$ will do as a transaction processing machine. To define "gives the illusion" we just need to alter the definition of local refinement a little. Local refinement says that a new system must appear the same as the old one whenever the system and environment halt together. For transaction processing machines we only require this for failures in which there is no partially complete transaction: the machine is not allowed to halt with partially complete

transactions; if necessary it must abort started but incomplete ones. (To this end every transaction must have the liveness property that it can never refuse to be aborted.) In addition, each transaction in the middle of its life must have the illusion of exclusive use of the machine. Define the whole transactions of a failure to be those which have either not started or have engaged in at least one *END* event:

$$whole_{\mathcal{A}}(t,r) \triangleq \{B : \mathcal{A} \mid t \restriction B = \langle\rangle \vee t \restriction (B \cap END) \neq \langle\rangle\}$$

Now we can define *transaction refinement* w.r.t. $\mathcal{A}$, written $\underline{\gg}_{\mathcal{A}}$, by:

Definition 41

$$\begin{aligned} &NEW \underline{\gg}_{\mathcal{A}} OLD \triangleq \\ &\quad \forall f : \phi NEW \cdot \\ &\qquad \exists g : \phi OLD \cdot f \cong_{whole_{\mathcal{A}} f} g \\ &\qquad \wedge \forall B : (\mathcal{A} - whole_{\mathcal{A}} f) \cdot \exists h : \phi OLD \cdot f \cong_{\{B\} \cup (whole_{\mathcal{A}} f)} h \end{aligned}$$

◇

This definition is like local refinement for transactions which are not executing and like independent refinement for transactions which are active.

The set of all suitable transaction processing machines is simply defined:

$$TPS \triangleq \{P \mid \exists Q \cdot Q \,\mathbf{sat}\, BASIC_{\mathcal{A}} \wedge P \underline{\gg}_{\mathcal{A}} Q\}$$

[W87] proves that a particular system, namely "optimistic concurrency control", is a member of *TPS* by giving a relation between its traces and those of a system that satisfies a criterion similar to $BASIC_{\mathcal{A}}$. This relation fulfills the same role as **reg** of chapter 3.

5.2 Security

Several other approaches to specifying security are interesting. We discuss here five others.

5.2.1 The Model of Bell and La Padula

[BLP76] specify security properties in terms of an idealised implementation. The style of the idealised implementation owes much to the Multics operating system [O72], for an important goal was to prove that this operating system enforced security.

The entities that share a system are classified as either *subjects* or *objects*; the two classes cover the space of entities and may overlap. Subjects are active entities (or the active side of an entity which is both subject and object) while objects are passive entities (or the passive side of an entity which is both subject and object). A subject can have access to an object and the access may be any combination of extraction of information (*observation*) and insertion of information (*alteration*). This gives four kinds of *access attribute*:

e neither observation nor alteration,

r observation with no alteration,

a alteration with no observation,

w both alteration and observation.

The names **e**, **r**, **a** and **w** are abbreviations for **execute**, **read**, **append** and **write** respectively, but these full names are not to be taken too literally. For example, executing a piece of code usually allows information to be deduced about the code from observation of its effects, and data can be altered without observation in other ways than appending.

A basic state of a system has four components. The first is a function associating subject-object pairs with the set of access attributes that the subject currently has to the object, the *current access set*[1]. The second is also a function , the *access permission matrix*, which relates each subject-object pair to the largest possible set of access attributes allowed for this pair. The third is a set of three level functions, which map subjects and objects to security levels. The set of security levels are treated as a partially-ordered set. An object has a single security level, while subjects have both a current and maximum security level. Finally there is a *hierarchy* on objects, which structures the space of objects as a forest (set of disjoint trees); this

[1] [BLP76] actually treat this as a set of triples.

component was added to model Multics directories. [BLP76] captures this as a quadruple. The third component is itself a triple. The components of the state, together with their types, are:

$$\begin{array}{lcl} (cas & : & OBJ \nrightarrow SUBJ \rightarrow \mathcal{P}\, ACCESS, \\ apm & : & OBJ \rightarrow SUBJ \rightarrow \mathcal{P}\, ACCESS, \\ (mlevel & : & SUBJ \rightarrow LEVEL, \\ clevel & : & SUBJ \rightarrow LEVEL, \\ olevel & : & OBJ \rightarrow LEVEL), \\ hierarchy & : & Forest\ OBJ) \end{array}$$

where

$$\forall s : SUBJ \cdot clevel\ s \leq mlevel\ s$$

The system is then defined as a finite-state machine which at each stage can input a request, and then output a decision and progress to a new state, together with an initial state and a relation on successive states.

[BLP76] say that the model can be used in two ways, for analysis and synthesis. For analysis the inputs and outputs need to be specified and the next-state relation determined. For synthesis the desired security properties are specified and then a suitable next-state relation determined.

The desired security properties are not of the sort that we have described in chapter 4. Instead they are properties of the state that guarantee the properties we have defined earlier. The first is called the *simple security property*, or *ss-property*. This is satisfied if no subject is observing an object of higher security level:

$$\begin{array}{l} \forall s : \mathrm{dom}\ cas, o : \mathrm{dom}\ (cas\ s) \cdot \\ \quad (cas\ s\ o) \subseteq \{\mathbf{w}, \mathbf{r}\} \Longrightarrow mlevel\ s \leq olevel\ o \end{array}$$

The second is called the **-property* (pronounced "star-property"). This prevents a subject copying information from a high object to a low object.

$$\begin{array}{l} \forall s : \mathrm{dom}\ cas, o_1, o_2 : \mathrm{dom}\ (cas\ s) \cdot \\ \quad (cas\ s\ o_1) \subseteq \{\mathbf{w}, \mathbf{r}\} \wedge (cas\ s\ o_2) \subseteq \{\mathbf{w}, \mathbf{a}\} \Longrightarrow olevel\ o_1 \leq olevel\ o_2 \end{array}$$

or, alternatively:

$$\forall s : \mathrm{dom}\ cas,\ o : \mathrm{dom}\ (cas\ s) \cdot cas\ s\ o = \left\{ \begin{array}{c} \mathbf{a} \\ \mathbf{r} \\ \mathbf{w} \end{array} \right\} \Longrightarrow clevel\ s \left\{ \begin{array}{c} \leq \\ = \\ \geq \end{array} \right\} olevel\ o$$

The *-property does not apply to *trusted subjects*, who would not break security if they could. Any system which is a refinement of one that enforces both the ss-property and the *-property is one that enjoys *mandatory* or *non-discretionary security*.

Each change in the state may change any of the components. To limit this change there is one further property, the *discretionary security property* or *ds-property*:

$$\forall s : \text{dom } cas, o : \text{dom } (cas\ s) \cdot (cas\ s\ o) \subseteq (apm\ s\ o)$$

The major result of [BLP76] is the *basic security theorem*, which states that security, defined in terms of satisfying the ss-property, *-property and ds-property, is inductive in the sense that it is only necessary to show that the starting state is secure and that all state changes preserve security: security is an invariant of the machine's state. However, as [McL85,McL87] demonstrate, this is actually a very trivial statement, and in practice has mislead people to put more reliance in this model than it deserves ([McL87] cites [US83b] as an example).

The approach described in this thesis is very different to that of the model of [BLP76]. Rather than describe (an abstract) implementation of a machine which is then defined to be secure, we are only concerned with the external view of the machine available to its users. This allows us to give definitions of security properties in a direct manner rather than as "the things that this machine does".

5.2.2 The SRI noninterference model

The SRI model is closer in flavour to the model presented in chapter 4. The model began life in [Fe77], in order to provide a basis for the tool reported on in [Fe80]. Notation and terminology were altered in [GM82], and the summary given in [Ru85a] follows this. Many other papers have been produced by the group at SRI, including [GM84] and [Ru85b].

As in [BLP76] the SRI model is based on finite state machines, but rather than concerning itself with the internal state of the machine it concerns itself with the traces that can be observed of the machine. The actions, as with [BLP76], are either commands (inputs) or outputs. The actions are tagged with the name of the user who engages in input or output.

The essential concept of the SRI model is that of *interference*, which captures information flow from one user to another. An input by one user interferes with a second user if the second user can later obtain an output which was impossible without the input. If there is no command the second user can issue which results in an output that depends on the first user's input, the first user is *noninterferring* with the second user; the system does not allow information to flow from the first user to the second. Write C_i for the set of commands (inputs) that can be issued by user i, O_i for the set of outputs that can be sent to user i and define $A_i \triangleq C_i \cup O_i$. The formal definition of noninterference is expressed[2]:

Definition 42 *User i is* noninterferring *with user j, for a system S if*

$$\forall t : \tau S, c : C_j \cdot$$
$$\{e \mid \langle e \rangle \in ((S/(t \hat{} \langle c \rangle))@O_j)\} = \{e \mid \langle e \rangle \in ((S/((t \backslash A_i) \hat{} \langle c \rangle))@O_j)\}$$

$\diamond$

Except for the fact that inputs and outputs are distinguished in this model, this criterion is the same as restriction of information flow when $i \not\rightarrow j$. It says that given the observation $\ell \hat{} \langle c, o \rangle$ both of the traces t and $t \backslash A_i$ are members of $(\text{infer}_S\ A_j\ (\ell \hat{} \langle c, o \rangle))$. If i represents all the users above j in the pre-order, then $t \backslash A_i \in \lceil A_j \rceil^*$. These two facts give:

$$(\text{infer}_S\ A_j\ (\ell \hat{} \langle c, o \rangle)) \cap (\lceil A_j \rceil^*) \neq \{\}$$

In the SRI model the complement of a reflexive relation $\rightarrow$ is known as a *security policy*. The intuitive meaning of $i \rightarrow j$ is that j is allowed to gain information about i. (As we have stated in chapter 4, if $\rightarrow$ is also transitive, and so a pre-order, then it represents a multi-level scheme.) A system is secure with respect to a security policy $\not\rightarrow$ if i is noninterferring with j whenever $i \not\rightarrow j$.

An important result is the *unwinding theorem*. This shows how to verify the security of systems that are defined in terms of state changes. In this it is similar to the fundamental security theorem of [BLP76]. It may be stated:
For a system S and reflexive relation $\rightarrow$, for all $s, t \in \tau S$, users i and j, and $c \in C_i$, if

[2]In our concrete notation, not the concrete notation in [Ru85a], etc.

1. $i \not\rightarrow j \Longrightarrow (S/(t\hat{}\langle c\rangle))@A_j = (S/t)@A_j$, *and*

2. $(S/s)@A_j = (S/t)@A_j \Longrightarrow (S/(s\hat{}\langle d\rangle))@A_j = (S/(t\hat{}\langle d\rangle))@A_j$

then S is secure with respect to $\not\rightarrow$.
We can translate this more directly into the notation of chapter 4, with the specialisation to pre-orders:

Lemma 49 *For a system S, disjoint partition $\mathcal{A}$ of αS and pre-order $\rightarrow$, for all $s,t \in \tau S$, $B,C \in \mathcal{A}$ and $b \in B$, if*

1. $B \not\rightarrow C \Longrightarrow (S/(t\hat{}\langle b\rangle))@C = (S/t)@C$, *and*
2. $(S/s)@C = (S/t)@C \Longrightarrow (S/(s\hat{}\langle b\rangle))@C = (S/(t\hat{}\langle b\rangle))@C$

then S restricts information flow w.r.t. $(\mathcal{A},\rightarrow)$. □

(A proof is given in [Ru85a].) Further specialisations of this theorem are given in [Ru85a], adding more detailed assumptions about the structure of the state space and structure of the multi-level scheme underlying $\not\rightarrow$. [Ru85a] also shows that the converse of the unwinding theorem holds; hence every system can be viewed in this way. This gives a recipe for building secure systems where the policy can be expressed as $\not\rightarrow$ for some reflexive relation $\rightarrow$.

A further refinement of noninterference, *conditional noninterference*, is defined in [GM82], [GM84]. It is a tricky notion to define: in [GM84] they say

> One thing we have discovered since our 1982 paper is that conditional noninterference is a rather subtle business; in particular, we know of three major different notions of noninterference, and numerous variations among them.

The first precise definition they give is in terms of read and write instructions on a finite-state machine. The definition may be written

Definition 43 *Let $c \subseteq C_i$ for some user i, and let P be the predicate defined by:*

$$P\,t \Longleftrightarrow t \in c^*$$

Let j be another user, then i is noninterferring with j unless P, in system S *if*

$$\forall t : \tau S \cdot \iota((S/t)@O_j = \iota((S/(s \hat{} u))@O_j)$$

where s is the longest prefix of t s.t. $P\ (s \restriction C_i)$

$u = (t/s) \restriction \mathcal{A}_i$

$\diamond$

This condition has some similarities with the definition of discretionary security (definition 4.3.5, page 65). Here, knowledge by one user of another is only allowed up to the last command of the latter that is allowed to pass information to the former; in our definition knowledge is allowed up to the last time that permission was withdrawn for information flow.

Apart from the use of finite state machines rather than CSP, which leads to a difference of flavour and style only, the difference between the SRI approach and ours is the starting point of noninterference. For them this is the fundamental notion; all definitions of security properties are defined in terms of it and its refinements. In the papers quoted, there is no way of specifying the information flow characteristics of a system like the tax-form service (example 29, page 68). By taking the inferences one user can make of another as primitive we are able to write specifications of more subtle security properties than interference or conditional interference allows. By taking CSP as the basis for our work we also inherit elegant notions of refinement and can abstract from input and output.

5.2.3 The SRI inference control model

In [GM84] the notion of *inference control* is introduced. This is a very general approach to inferences on data bases. It starts with a logical system $(S, \vdash)$, where S is a set of sentences and the inference relation, $\vdash$, is a preorder on S. The logical system comes with a conjunction operator, $\wedge$, with the property that the system is closed under it, that is:

$$\begin{aligned} s_1, s_2 \in S &\implies (s_1 \wedge s_2) \in S \\ s_1 \vdash s_1', s_2 \vdash s_2' &\implies (s_1 \wedge s_2) \vdash (s_1' \wedge s_2') \end{aligned}$$

If $T \subseteq S$ and $s \in S$, $T \vdash s$ is defined to mean $\bigwedge T \vdash s$. The *inferential closure* of a subset T of S is defined as the smallest set, T^*, with the

properties:

$$T \subseteq T^* \subseteq S$$
$$T^* \vdash s \implies s \in T^*$$

[GM84] prove that every subset of S has an inferential closure.

A *database* is defined to be any subset of S, and, if D is a database, then a view of it is any $V \subseteq D^*$. Views are related by a pre-order, which we write here as $\propto$, and which is pronounced "subview":

$$V_1 \propto V_2 \;\hat{=}\; V_1 \subseteq V_2^*$$

The pre-ordered set of views of a database D are denoted $\mathit{View}\ D$.

A *classification* of views is a further preorder, which we will write $\rightharpoonup$, that satisfies:

$$V_1 \propto V_2 \Longrightarrow V_1 \rightharpoonup V_2$$

Such a preorder captures a multi-level security scheme, as it does in chapter 4. Let $\mathcal{V}$ be a finite set of views. Their aggregate is the upper bound over $\propto$, $\bigsqcup_{\propto} \mathcal{V}$. It is a theorem that

$$\bigsqcup_{\rightharpoonup} \mathcal{V} \rightharpoonup \bigsqcup_{\propto} \mathcal{V}$$

That is, an aggregate has at least as much right to information as the sum of its components. $\mathcal{V}$ is *a potential security violation by aggregation* if, furthermore

$$\bigsqcup_{\rightharpoonup} \mathcal{V} \not\rightharpoonup \bigsqcup_{\propto} \mathcal{V}$$

That is, the aggregate has strictly more right to information than the sum of the components.

In our formulation the set of sentences, S, is the set of predicates with (possibly) a free variable tr, which represents a trace. A database is the set of predicates true of a given trace. Views are sets of predicates which contain predicates of the form:

$$tr \restriction B = \ell \wedge tr \in \tau R$$

The closure of such a view is the set of predicates true of every member of $(\mathrm{infer}_R\ B\ \ell)$:

$$\{P \mid \forall tr : \mathrm{infer}_R\ B\ \ell \cdot P\}$$

These are the predicates which must hold. The remaining predicates in S fall into two classes: those which must be false (the negation of predicates in this set) and those which are true for some of the traces in $(\mathrm{infer}_R\, B\, \ell)$ and false for others. A user holding additional information can be modelled by adding further predicates to the view.

What we have done in chapter 4 is to identify a way of coding views and inferences for a system described in terms of traces over its interface. Such an approach is very general and is, we believe, a very natural one.

5.2.4 Foley's Theory of Information Flow

Foley's approach to information flow is described in [Fo87a] and [Fo87b] and is intended to be part of his doctoral thesis. Like the work reported on here, it is in terms of CSP.

Foley starts by capturing what it means for information to be able to flow from one user of a system to another (rather than the reverse as in the SRI noninterference model). In words, the definition says that information may flow from one user to another through a system if some candidates for the first user can restrict the behaviour of the interface to the second user. Transcribing his notation into that used here, his definition is:

Definition 44 *Let B and C be disjoint subsets of the alphabet of a system S. Then* information can flow from B to C through S *if:*

$$\exists t : S@C, V : Proc_B \cdot V \textbf{ valid } S \wedge t \notin (S \| V)@C$$

$\diamond$

This relation he denotes $B \overset{S}{\vdash} C$ or $B \overset{S}{\models} C$, depending on the definition of **valid**. The relation **valid** captures the fact that V is in some way well-behaved with respect to the system S. In [Fo87a] a definition is given over the syntax of CSP which is intended to assert that:

> Each user of [a system] is considered 'passive', in that it can only engage [in] events offered to it by the process. A consequence of this is that a user cannot abnormally stop, or deadlock the system.

The purpose of this is to let him make liveness assumptions about information flow: "not only is information able to flow, it does flow". In this case $\vdash$ is used.

In [Fo87b] he defines V **valid** S to be universally true, that is he allows any user (with suitable alphabet) to be a potential user; this restricts the theory to safety statements. In this case $\models$ is used and the definition reduces to

$$\exists t : S@C \cdot t \notin (S \| STOP_B)@C$$

or, equivalently, information does *not* flow from B to C through S if

$$S@C = (S \| STOP_B)@C$$

When B is taken to be all the users above C in some pre-order on the users of S, this is identical to restriction of information flow:

$$\forall \ell : S@C \cdot \lceil C \rceil^* \cap \mathrm{infer}_S\, C\, \ell \neq \{\}$$

To see that Foley's condition is stronger than ours, pick any ℓ in $S@C$. This is also a member of $(S \| STOP_{\alpha S - \lceil C \rceil})@C$, and so can be due to some trace entirely composed of members of $\lceil C \rceil$. To see the reverse, we just have to show $S@C \subseteq (S \| STOP_{\alpha S - \lceil C \rceil})@C$. Pick any member, ℓ, of $S@C$, then there is some trace composed entirely of members of $\lceil C \rceil$ which is also a trace of S and which has ℓ as its projection onto C. As this trace has no members of $\alpha S - \lceil C \rceil$ it is a trace of $S \| STOP_{\alpha S - \lceil C \rceil}$ which has ℓ as its projection onto C.

[Fo87a] also has a notion of information flow with *co-operating users*. A user co-operates with another if the second has some information about the first which allows it deduce facts it could not otherwise deduce.

Example 33 The following system has three users, b, c and d:

$$SUM \;\widehat{=}\; (b?x : Z \rightarrow SKIP \| c?y : Z \rightarrow SKIP); d!(x+y) \rightarrow STOP$$

where Z is the set of integers. All d can deduce is that b and c have output integers. However if d knows from some external source that b outputs, for example, 0 it can deduce the exact value of c's output. $\triangle$

He defines a boolean valued function *Canleak*, such that $Canleak\, S\, B\, C$ if C can deduce something about B with knowledge of another user:

$$Canleak\; S\; B\; C \;\widehat{=}\; \exists V : Proc_{(\alpha S - (B \cup C))} \cdot V \textbf{ valid } S \wedge B \overset{S \| V}{\vdash} C$$

In our style we give the inferences that can be made about the past behaviour of a system, given an observation. These inferences are captured by a set of traces that the system may have engaged in. Extra information about a user can be used to discount some of these traces, and so increase the amount of information gained.

Example 34 Continuing example 33, we see that

$$\begin{aligned}
&\text{infer}_{SUM}\,\{d.z \mid z \in Z\}\,\langle\rangle\ = \\
&\quad \{\langle\rangle, \langle b.z1\rangle, \langle c.z2\rangle, \langle b.z1, c.z2\rangle, \langle c.z2, b.z1\rangle \mid z1, z2 \in Z\} \\
&\text{infer}_{SUM}\,\{d.z \mid z \in Z\}\,\langle d.z\rangle\ = \\
&\quad \{\langle b.z1, c.z2, d.z\rangle, \langle c.z1, b.z2, d.z\rangle \mid z1, z2 \in Z \wedge z = z1 + z2\}
\end{aligned}$$

The information that c satisfies the specification $tr \leq \langle c.0\rangle$ reduces these sets to

$$\{\langle\rangle, \langle b.z1\rangle, \langle c.0\rangle, \langle b.z1, c.0\rangle, \langle c.0, b.z1\rangle \mid z1 \in Z\}$$

and

$$\{\langle b.z, c.0, d.z\rangle, \langle c.0, b.z, d.z\rangle\}$$

respectively. All the traces in the latter set satisfy the property that the communication with b is the same as that with d. $\triangle$

If some facts about another user are known we can capture it by a process with the largest set of traces all of which satisfy these facts, U say and then calculate $\text{infer}_{S\|U}$. This is equivalent to Foley's idea.

As with the SRI noninterference model, the difference between our approach and Foley's is the starting point: inferences or information flow. Again, detecting the possibility of information flow is too coarse a notion to measure the *amount* of information flowing, which our system of inferences can.

Another difference between this thesis and the approach of SRI and Foley is the lack of an order. Inferences about the past behaviour provide a means to measure security/information flow through a system. Noninterference and information flow just indicate whether interference happens or information flows respectively. Our order allows a measurement of the amount of interference or information flow.

5.2.5 Denning's Theory of Information Flow

Denning's theory was developed in her thesis [De75]. It is phrased in terms of programming language concepts. For example, the following statements, in Dijkstra's notation of guarded commands [Di76], cause a flow from y to x:

$$x := y$$
$$\text{if } y \leq 0 \rightarrow x := 0 \,[\!]\, y > 0 \rightarrow x := 1 \text{ fi}$$

The first is an explicit flow, the second implicit. The amount of information flowing can be quantified as a number of bits (see, for example [De82]) and is different in these two cases. This model, like ours, quantifies the amount of information flowing, rather than just the presence or absence of flow. The difference with our model is the choice of interface: we are far more abstract, we do not assume that the interface is a programming language.

5.2.6 Landwehr's classification

In 1981, between [BLP76] and [De75] on one hand and [Fo87a] and the SRI work on the other, Landwehr surveyed available models of security [Lan81]. The survey is summarised by a table. In the table he classifies models by *motivation*, *view of security* and *approach*, and also notes whether the model *separates protection mechanism and security policy* and whether *systems based on or represented by this model have been implemented*. Below we add the models discussed above to the table.

By "motivation" he means whether the model is primarily for the representation of existing systems or to guide construction of future systems. Both Bell & La Padula's model and Denning's model he places in the latter category. None of the other models discussed so far fit either of these categories, although the purpose of the unwinding theorem is to suggest an implementation. Rather, they are means of analysing given systems and defining suitable properties of proposed systems *without* suggesting an implementation.

Under "view of security" there are three classifications. The first are those that *model access to objects without regard to contents*; Bell & La Padula's model fits into the this category. Denning's model falls into the second: those which *model flow of information among objects*. The last

class contains those which *model inferences that can be made about protected data*. Our model falls across both of these categories, but information flow is between subjects and the inferences are about other users' behaviour and not protected data. Both the SRI noninterference model and Foley's model are about information flow between subjects and the lack thereof.

The categories under "approach" are whether the model focuses on *system structures*, such as files and processes, on *language structures*, such as variables and statements, or on *operations on capabilities*. Bell and La Padula are in the first, Denning in the second and the other models mentioned above in none of these categories. Rather, the other models focus on *the behaviour at the interface to the system.*

Alone of the models we have discussed, Bell and La Padula's model does not separate the policy, or specification, from the protection mechanism, or implementation. Neither the work reported on here, nor Foley's methods have been used to build a secure system. Attempts have been made using the models of Bell and La Padula and of SRI. [Lan81] does not record Denning's model as having given rise to an implementation.

5.3 The Knowledge Calculus

The knowledge calculus is explained in [ChMi86]. It is an alternative formalism in which shared systems can be discussed. Let $\mathcal{A}$ be a collection of user windows, b a predicate with (possibly) a free trace variable tr, then the predicate $\mathcal{A}$ *knows* b is defined by

$$\mathcal{A} \textit{ knows } b \;\hat{=}\; \forall s \cdot s \cong_{\mathcal{A}} tr \Longrightarrow b[s/tr]$$

This predicate expresses the fact that every trace in the same equivalence class as tr satisfies b, so even if it is not known which trace from the class occurred it is still known that b holds.

We can give an alternative definition of local refinement, restricted to traces, in terms of *knows*: NEW locally refines OLD w.r.t. $\mathcal{A}$ if

$$tr \in \tau NEW \Longrightarrow \overline{\mathcal{A} \textit{ knows } (tr \notin \tau OLD)}$$

where $\overline{b}$ denotes *not* b. This may be read "NEW locally refines OLD w.r.t $\mathcal{A}$ if the environment cannot tell that a trace of NEW is not a trace of OLD".

Clearly it would not be hard to generalise these ideas from traces to any observations of processes, nor to rephrase it for independent refinement.

[ChMi86] is concerned with results which track information flow, such as: "if *not* $(\{B\}$ *knows* $b)[s/tr]$ and $(\{C\}$ *knows* $(\{B\}$ *knows* $b))[(s \hat{} t)/tr]$ then there must be a chain of communication from B to C in t. They are also able to find lower bounds on the number of messages to solve various problems. This is a way a of determining sequences of messages from one user to another which do not carry secret information. Security is not directly addressed in [ChMi86], however.

Chapter 6

Summary and Conclusions

6.1 Summary and future directions

We have now investigated two important topics in the study of shared systems: how to check whether one shared system can be a suitable replacement of another (chapter 3) and how to measure and specify maximum information flows across shared systems (chapter 4). There are interesting ways of continuing the study of both topics.

6.1.1 Refinement of shared systems

Refinement of shared systems is easier than that of single-user systems, because the environment of a shared system is less discriminating than that of a single user system (lemma 5). There are more shared systems that locally refine a given shared system than refine it in the ordinary sense. The local refinements may often be freer in their behaviour than the original system, which gives the possibility of distributed implementations; indeed we have spent some time investigating a distributed re-implementation (3.2.3).

On the debit side, the local refinement relation is not a partial order, but only a pre-order (lemmas 6 and 7). This is not a real problem as we can always form equivalence classes of processes to obtain a partially ordered set, as explained in chapter 2. The properties of this order deserve investigation. If it turned out that all descending chains in the set had limits then the set could be used to give a semantics to a notation including recursion for describing shared systems directly. The investigation of the

properties of this order and the design of such a notation is an interesting topic, but is not pursued further here.

We have only presented two techniques for proving local refinement: exhaustive examination of the failures (example 6) and the exhibition of a relation between observations of the two systems (chapter 3.2.3). Algebraic proof methods, and correctness proving transformations in the style of [A87] are also a topic for further research.

Similar remarks apply to the independent refinement relation. This is an even easier relation to satisfy than local refinement (lemma 34). It is a pre-order but not a partial order (lemmas 35 and 36), and the same remarks about producing a partially ordered space apply as do for local refinement.

It would be possible to treat local and independent refinement in an identical fashion by adding a special user in the co-operating case, who reads the final state, but does nothing else. It is also necessary to add new events for the other users to signal to the system that they have terminated and to insist that every user does sign off. We could then treat all systems as if they were serving independent users. The choice is between having an extra relation for the co-operating case on the one hand and more complex modelling of the system together with liveness assumptions about the users of the system on the other. We choose the former in chapter 3 as it makes fewer assumptions about the users; it remains to work out the details for the latter method.

In chapter 5 we showed how to generalise a characterisation of transaction processing machines given in [W87] by using a relation similar to local and independent refinement (in fact a cross between the two). The identification of other cases where such constructions can be usefully employed is an interesting area for research. *Event refinement* is one case, which happens to be closely related to transaction processing. Event refinement is a refinement of processes where 'big' events (for example, "*greet*") are replaced by processes with an alphabet of 'small' events (for example, "*greet*" could be replaced by the process

$$\begin{aligned} GREET \;\widehat{=}\; & (\mathit{grasp\text{-}hands} \rightarrow \mathit{shake} \rightarrow \mathit{release} \rightarrow \mathit{SKIP} \\ & \mid \mathit{raise\text{-}hand} \rightarrow \mathit{wave} \rightarrow \mathit{drop\text{-}hand} \rightarrow \mathit{SKIP}) \end{aligned}$$

in a refinement step). Processes with an alphabet of small events, such as $GREET$, can be identified with transactions; a transaction on a sequential machine can be identified with a big event, such as *greet*. Event refinement

and transaction refinement are clearly related; their precise relationship is left for later research.

6.1.2 Security

The maximum amount of information flow between isolated users of a system is captured by its inference function (definition 29, page 55). By comparing inference functions we have shown how to measure the security of one system relative to another. By generalising the notion of an inference function to that of a security specification (definition 35, page 69) we have provided a tool for specifying the desired maximum degree of information flow across a system.

We have seen how all sorts of security properties can be expressed using inference functions and security specifications. This is the beginning of the design of a notation for expressing security properties of systems. As security specifications form a complete lattice (lemma 47) they can be used as a model for a notation that includes recursion; we have shown that this is a useful concept in example 32. What other features such a notation should include, indeed its general shape, is a matter for further research.

When we discussed trusted users we noted that secure systems can be derived by taking a system and a trusted user of that system in parallel. It would be a valuable exercise to attempt to derive secure systems, starting from a simple operating system, and a security property such as isolation and progressing to more complicated systems and security properties. Such a project—if successful—would provide a reliable means of producing systems in the 'Beyond class (A1)' category of [US83b]. This is also left for future study.

6.1.3 General

The theory of Communicating Sequential Processes is a suitable vehicle for studying shared systems. The mapping of a system to a process and an individual user's interface to a subset of the alphabet of the process is a natural and fruitful one. It leads to simple definitions of the basic concepts we have invented in our study: a shared system, local and independent refinement, inference functions and security specifications. Their use is

somewhat harder, but this should be helped by experimental case studies and the development of suitable tools.

6.2 Discussion

The genesis of this theory was the incidental discovery of the implementation of which *NEW2* (section 3.2.3, page 39) is an abstraction and simplification. It was not clear that *NEW2* was a correct re-implementation of *OLD*—although the implementors *knew* this intuitively. One of the implementors, who had some knowledge of formal methods, believed *NEW2* was correct because each node progresses through the same sequence of states. A lot of time was wasted attempting to use this fact to frame a definition of correctness and a proof. The difficulty lies in trying to phrase properties of a system in terms of a particular implementation strategy. Once the definitions of local equivalence and local refinement were found, based on behaviour at the interface and not the internal state of the system, the proof was obtained quickly.

Shortly after this discovery—again incidentally—I was asked to comment on [BLP76]. This paper discusses the security aspects of shared systems by considering the structure of their states and the state transitions. By applying the same discipline as we did for discussing refinement of shared systems the theory of specification of security properties contained in this thesis was developed with little trouble.

The moral to be drawn from these two cases is: reasoning about a shared system is easier when based on the behaviour at its interface and not on its internal state.

6.3 Conclusion

We have achieved an understanding of shared systems based on a simple mathematical model. This provides a basis for sound engineering practices. In particular, the possibility of building verifiably secure systems by calculation seems hopeful and is possibly the most exciting outcome of this study. But let us not forget the manufacturers and (human) users of the machines, modelled in section 3.2.3, who can have greater confidence in the upward compatibility of the replacement machine with the original one.

Bibliography

[A87] A.E. Abdallah, *Transformational Development of Parallel Algorithms*, D.Phil thesis, University of Oxford, draft 1987

[BLP76] D.E. Bell & L.J. La Padula, *Secure computer system: unified exposition and multics*, Report ESD-TR-75-306, The MITRE Corporation, March 1976

[Br83] S. Brookes, *A Model for Communicating Sequential Processes*, D.Phil Thesis, University of Oxford, January 1983

[BrRo85] S. Brookes & A.W. Roscoe, *An Improved Failures Model for Communicating Sequential Processes*, in Proceedings NSF-SERC Seminar on Concurrency, Springer-Verlag, New York, 1985

[ChMi86] K.M. Chandy & J. Misra, *How processes learn*, Distributed Computing, **1** (1), 1986, pp 40–52

[Da81] C.J. Date, *An Introduction to Database Systems, 3rd ed.*, Addison-Wesley, 1981

[De75] D.E.R Denning, *Secure Information Flow in Computer Systems*, Ph.D. Thesis, Purdue University, 1975

[De82] D.E.R. Denning, *Cryptography and Data Security*, Addison-Wesley, 1982

[Di76] E.W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, 1976

[Fe77] R.J. Feiertag, K.N. Levitt & L. Robinson, *Proving Multi-level Security of a System Design*, Proc. 6th ACM Symposium on Operating Systems Principles, November 1977 (cited in [Ru85a])

[Fe80] R.J. Feiertag, *A technique for Proving Specifications are Multi-level Secure*, Technical report CSL109, Computer Science Laboratory, SRI International, Menlo Park, CA., January 1980 (cited in [Ru85a])

[Fo87a] S.N. Foley, *A Universal Theory of Information Flow*, Proc. 1987 IEEE Symp. on Security & Privacy, Oakland, CA.

[Fo87b] S.N. Foley, Private Communication

[GM82] J.A. Goguen & J. Meseguer, *Security Policies and Security Models*, in Proc. 1982 Symposium on Security & Privacy, Oakland, CA., IEEE Computer Society, April 1892

[GM84] J.A. Goguen & J. Meseguer, *Unwinding and Inference Control*, in Proc. 1984 Symposium on Security & Privacy, IEEE Computer Society, 1984

[Gr81] D. Gries, *The Science of Programming*, Springer-Verlag, 1981

[Ha87] I. Hayes (ed), *Specification Case Studies*, Prentice-Hall International, 1987

[HeSa86] J. He & J.W. Sanders, *A Predicate Model for Communicating Sequential Processes*, Programming Research Group internal paper, Oxford University, 1986

[Hen80] P. Henderson, *Functional Programming: Application and Implementation*, Prentice-Hall International, London, 1980

[Ho78] C.A.R. Hoare, *Communicating Sequential Processes*, Communications of the ACM, **21** 8 pp 666–677, August 1978

[Ho80] C.A.R. Hoare, *A model for Communicating Sequential Processes*, in: On the Construction of Programs (R.M. McKeag &

A.M. McNaughton eds.), pp 229–243, Cambridge University Press, 1980

[Ho84] C.A.R. Hoare, *Programming: Sorcery or Science?*, IEEE Software, **1** 2 pp 5–16, April 1984

[Ho85] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall International, London, 1985

[HoHe85] C.A.R. Hoare & Jifeng He, *The Weakest Prespecification*, PRG monograph 44, University of Oxford, June 1985

[Jo80] C. Jones, *Software Development: A Rigorous Approach*, Prentice-Hall International, London, 1980

[I84] Inmos Ltd., *The occam^{TM} programming manual*, Prentice-Hall International Series in Computation, 1984

[Lan81] C.E. Landwehr, *A Survey of Formal Models of Computer Security*, U.S. Naval Research Laboratory, Report 8489, 30 September 1981

[Lar86] K.G. Larsen, *Context-Dependent Bisimulation Between Processes*, Ph.D. Thesis, University of Edinburgh, May 1986

[LM86] K.G. Larsen & R. Milner, *A Complete Protocol Verification using Relativised Bisimulation*, University of Edinburgh Lab. for Foundations of Comp. Sci. report ECS-LFCS-86-13, September 1986

[Mi80] R. Milner, A Calculus of Communicating Systems, Springer-Verlag LNCS 92, 1980

[McL85] J. McLean, *A Comment on the 'Basic Security Theorem' of Bell and La Padula*, Information Processing Letters **20** 2, pp 67–70, 15 Feb. 1985

[McL87] J. McLean, *Reasoning About Security Models*, Proceedings of the 1987 Security and Privacy Conference, Oakland, IEEE Press, pp 123–131, 1987

[OH86] E.-R. Olderog & C.A.R. Hoare, *Specification Oriented Semantics for Communicating Sequential Processes*, Acta Informatica **23**, pp 9–66, 1986

[O72] E.I. Organick, *The Multics Systems*, The MIT Press, Cambridge, Mass., 1972 (cited in [BLP76])

[M87] C.C. Morgan, *Data Refinement Using Miracles*, submitted to Information Processing Letters.

[P79] C.H. Papadimitriou, *The Serializability of Concurrent Database Updates*, Journal of the ACM, **26** (4), 1979, (cited in [W87])

[ReRo86] G.M. Reed & A.W. Roscoe, *A Timed Model for Communicating Sequential Processes*, in Automata, Languages and Programming: Proceedings of the 13^{th} International Colloquium. Ed. L. Kott, Springer-Verlag LNCS 226, 1986

[Ro82] A.W. Roscoe, *A Mathematical Theory of Communicating Processes*, D.Phil. Thesis, Oxford University, 1982

[Ru85a] J. Rushby, *The SRI Security Model*, Internal paper of Comp. Sci. Lab., SRI International, 6 March 1985

[Ru85b] J. Rushby, *Networks are Systems*, in Proc. DoD Computer Security International Workshop on Network Security, 1985

[Sc76] D. Scott, *Data Types as Lattices*, SIAM Journal on Computing, **5**, 1976

[St77] J.E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, The MIT Press Series in Computer Science, 1977

[Ta41] A. Tarski, *On the Calculus of Relations*, Journal of Symbolic Logic, 6, 1941

[Tu50] A. Turing, *Computing Machinery and Intelligence*, reprinted in Computers and Thought eds. F.A. Feigenbaum & J. Feldman, Prentice-Hall, New York, 1961.

[Tn85] D.A. Turner, *Programs as Executable Specifications*, in Mathematical Logic and Programming Languages, ed. Hoare & Shepherdson, Prentice-Hall International, 1985

[US83a] United States Department of Defense, *Reference Manual for the Ada Programming Language*, January 1983

[US83b] United States Department of Defense, *Department of Defense Trusted Computer System Evaluation Criteria*, CSC-STD-001-83, U.S. National Computer Security Center, August 1983

[W87] J.C.P. Woodcock, *Transaction Processing Primitives and CSP*, to be published in The IBM Journal of Research and Development

[WH86] J.C.P. Woodcock & C.A.R. Hoare,*The Problem with Lifts...*, PRG internal document, 1986

Appendix A

Traces and operators on traces

We use the following notation for traces:

$\langle\rangle$ The empty trace.

$\langle e \rangle$ The singleton trace whose only member is e.

$\langle e_1, \ldots, e_k \rangle$ The trace whose members are $e_1, \ldots, e_k$ in that order.

$t[n]$ is the n^{th} element of the trace t. $t[l..n]$ is the subtrace of t from the l^{th} element to the n^{th} element inclusive.

The *catenation* of two traces, s and t say, is written $s \hat{} t$. If S and T are two sets of traces we define

$$S \hat{} T \triangleq \{s \hat{} t \mid s \in S \wedge t \in T\}$$

We use two partial orders over traces. If s is an *initial prefix* of t we write $s \leq t$. If s is a *subsequence* of t we write $s \,\mathbf{in}\, t$. Note that

$$s \leq t \Longrightarrow s \,\mathbf{in}\, t$$

but that the reverse implication does not hold.

The set of all finite traces over the alphabet A is written A^*.

A function $f \in A^* \longrightarrow B^*$ is *strict* if $f\langle\rangle = \langle\rangle$ and *distributive* if $f(s \hat{} t) = (f\,s) \hat{} (f\,t)$. Distributive functions are always strict and are defined by their behaviour on singleton sequences.

If $f \in A \longrightarrow B$ then f^* is the distributive function in $A^* \longrightarrow B^*$ defined on singletons by:

$$f^*\langle a \rangle \triangleq \langle f\,a \rangle$$

For any set B the *restriction* operator to B, written postfix $\restriction B$, is the distributive operator defined on singletons by:

$$\langle a \rangle \restriction B \triangleq \begin{cases} \langle a \rangle & \textbf{if } a \in B \\ \langle\rangle & \textbf{otherwise} \end{cases}$$

For any name c the *strip* operator for c, written postfix $\downarrow c$, is the distributive operator defined on singletons by:

$$\langle a \rangle \downarrow c \triangleq \begin{cases} \langle m \rangle & \textbf{if } a = c.m \\ \langle\rangle & \textbf{otherwise} \end{cases}$$

Appendix B

Semantic definition of standard CSP operators

We present each operator either in the vertical form:

Process
Conditions
Alphabet
Failures
Divergences

or in the horizontal:

Process	Conditions	Alphabet	Failures	Divergences

A condition of *True* means no conditions are imposed.

The constants:

$STOP_A$	*True*	A	$\{\langle\rangle\} \times PA$	$\{\}$
RUN_A	*True*	A	$A^* \times \{\{\}\}$	$\{\}$
$CHAOS_A$	*True*	A	$A^* \times PA$	A^*

The prefixing operator:

$e : E \rightarrow P_e$
$\forall e : E \cdot E \subseteq \alpha P_e = A$
A
$(\{\langle\rangle\} \times \mathcal{P}(A - E)) \cup \{(\langle e \rangle \hat{} t, r) \mid e \in E \wedge (t, r) \in \phi P_e\}$
$\{\langle e \rangle \hat{} d \mid e \in B \wedge d \in \delta P_e\}$

Deterministic and non-deterministic choice: The difference between $[]$ and $\sqcap$ is that, *initially*, $P[]Q$ can refuse less than $P \sqcap Q$.

$P[]Q$
$\alpha P = \alpha Q = A$
A
$((\phi P \cup \phi Q) - (\{\langle\rangle\} \times \mathcal{P} A)) \cup (\phi P \cap \phi Q) \cup (\delta(P[]Q) \times \mathcal{P} A)$
$\delta P \cup \delta Q$

$P \sqcap Q$	$\alpha P = \alpha Q$	$\alpha P \cup \alpha Q$	$\phi P \cup \phi Q$	$\delta P \cup \delta Q$

The parallel operator:

$P \| Q$
$True$
$\alpha P \cup \alpha Q = A$
$\{f \mid f \upharpoonright \alpha P \in \phi P \wedge f \upharpoonright \alpha Q \in \phi Q\} \cup (\delta(P \| Q) \times \mathcal{P} A)$
$\{d \mid (d \upharpoonright \alpha P \in \delta P \wedge d \upharpoonright \alpha Q \in \tau Q) \vee (d \upharpoonright \alpha P \in \tau P \wedge d \upharpoonright \alpha Q \in \delta Q)\} \hat{} A^*$

where the restriction operator on failures is defined:

$$(t, r) \upharpoonright B \triangleq (t \upharpoonright B, r \cap B)$$

Sequential composition:

$P;Q$
$\alpha P = \alpha Q = A$
A
$\{(t,r) \mid (t, r \cup \{\surd\}) \in \phi P\}$ $\cup \{(s \hat{} t \mid s \hat{} \langle\surd\rangle \in \tau P \wedge (t,r) \in \phi Q\}$ $\cup \delta(P;Q) \times \mathcal{P} A$
$\{s \mid s \in \delta P \wedge \langle\surd\rangle \not\in s\} \cup \{s \hat{} t \mid s \hat{} \langle\surd\rangle \in \tau P \wedge \langle\surd\rangle \not\in s \wedge t \in \delta Q\}$

Renaming by a total injection:

$f\,P$
f is a total injection from αP
$f(\alpha P)$
$\{(f^* t, f(r)) \mid (t,r) \in \phi P\}$
$f^*(\!\lvert \delta P \rvert\!)$

Renaming by a surjection:

$f^{-1}\,P$
f is a surjection to αP
$f^{-1}(\alpha P)$
$\{(t,r) \mid (f^* t, f(r)) \in \phi P\}$
$f^{-1*}(\!\lvert \delta P \rvert\!)$

Hiding:

$P \setminus E$
$True$
$\alpha P - E = A$
$\{(t \restriction A, r) \mid (t, r \cup E) \in \phi P\} \cup \delta(P \setminus E) \times \mathcal{P} A$
$((\!\lvert \delta P \cup \{d \mid \forall n : \mathcal{N} \cdot \exists s : E^* \cdot \#s > n \wedge (d \hat{} s) \in \tau P\}\!\rvert) \restriction A) \hat{\_} A^*$

Recursion:

$\mu X : A \cdot F\,X$
$\alpha P = A \Longrightarrow \alpha(F\,P) = A$
A
$\bigcap_{n:N}\ \phi(F^n\,CHAOS_A)$
$\bigcap_{n:N}\ \delta(F^n\,CHAOS_A)$

Some derived operators:

A constant:

$$SKIP_A \ \widehat{=}\ \surd \rightarrow STOP_A$$

A slaving parallel operator:

$$P\,//\,Q \ \widehat{=}\ (P \parallel Q) \setminus \alpha P \qquad \text{if} \qquad \alpha P \subseteq \alpha Q$$

Iteration:

$$\underline{*}P \ \widehat{=}\ \mu X : \alpha P \cdot P; X$$

Output:

$$c!v \rightarrow P \ \widehat{=}\ c.v \rightarrow P$$

Input:

$$c?x : V \rightarrow P_x \ \widehat{=}\ e : \{c.x \mid x \in V\} \rightarrow (strip_c; P)_e$$

where P is a function from values in V to processes and

$$strip_c\,(c.e) \ \widehat{=}\ e$$

Naming:

$$c : P \ \widehat{=}\ strip_c^{-1}\,P$$

where

$$strip_c\,(c.e) \ \widehat{=}\ e$$

Naming by a set:

$$C : P \ \widehat{=}\ Strip_C^{-1}\,P$$

where

$$Strip_C\,(c.e) \ \widehat{=}\ e \text{ whenever } c \in C$$

Appendix C

Code for a buffer

The purpose of this appendix is to demonstrate that when synchronous communication is a primative in a programming notation information flows in both directions at each communication. A buffer can be built from two components, one which is syntactically restricted not to engage in output and one which is syntactically restricted not to engage in input. Information is transferred from the input-only component to the output-only component. In CSP the algorithm is easy to write, in *occam*[1] [I84] it is a little harder as a choice is not allowed between output guards. We give algorithms in both notations below.

Similar algorithms could be given in a language such as *Ada*[2] [US83a], and any language where an upper bound is known on the amount of buffering provided on each channel. An algorithm does not exist if the amount of buffering on each channel is unknown, and so the lazy functional languages (such as Miranda [Tn85]) have the property that information flows from outputs to inputs but not *vice versa*.

The number of channels needed to connect the two components is constant in the width of the channels, as one bit can be sent at a time. For the CSP solution we assume that we are buffering booleans; the occam solution we give works for any width of channel.

In CSP a suitable algorithm has two channels between the components, c_0 and c_1:

$$B \triangleq P \| Q$$

[1] *occam* is a trademark of Inmos Ltd.

[2] *Ada* is a trademark of the US Government

where

$$
\begin{array}{lcl}
P & \widehat{=} & source?x \rightarrow c_x?y \rightarrow P \\
Q & \widehat{=} & (\ c_0!0 \rightarrow sink!0 \rightarrow Q \\
 & & \mid\ c_1!0 \rightarrow sink!1 \rightarrow Q)
\end{array}
$$

In *occam*, an algorithm with three channels connecting the components is:

```
DEF bits = ...                          :
PROC buffer(CHAN in, out) =
  CHAN link[2], control :
  PROC p(CHAN source, con[], ack) =
    ... input on source, con[] and ack, no free variables
  PROC q(CHAN sink, con[], ack) =
    ... output on sink, con[] and ack, no free variables
  PAR
    p(in, link, control)
    q(out, link, control) :
```

The processes p and q have the gross structure:

```
PROC p(CHAN source, con[], ack) =
  PROC transmit.bit(VALUE x, i, CHAN con[], ack) =
    ... communicate value of i'th bit of x on con[]
  WHILE TRUE     -- forever
    VAR x :
    SEQ
      source?x  -- fetch value
      SEQ i = [0 FOR bits] -- for each bit
        transmit.bit(x, i, con, ack) : -- send it

and

PROC q(CHAN sink, con[]) =
  PROC receive.bit(VAR x, VALUE i, CHAN con[], ack) =
    ... receive bit on con[] and store it in i'th bit of x
  WHILE TRUE
    VAR x:
```

```
    SEQ
      SEQ i = [0 FOR bits] -- for each bit
        recieve.bit(x, i, con, ack)
      sink!x :  -- deliver value
```

Now we just need the algorithm to communicate one bit. The sender codes the bit as the order of the channels it is prepared to input on.

```
PROC transmit.bit(VALUE x, i, CHAN con[], ack) =
  VAR j :
  SEQ
    get.bit(j, x, i)      -- j := i'th bit of x
    con[j]?ANY            -- input on channel corresp. to j
    ack?ANY               -- accept acknowledgement
    con[1-j]?ANY          -- input on channel corresp. to not j
```

The receiver just detects in which order the sender is willing to accept input.

```
PROC receive.bit(VAR x, VALUE i, CHAN con[]) =
  CHAN int[2] :
  PROC one.one.buff(CHAN left, right) =
    ... a one-shot one-place buffer, from left to right
  PROC controller(VAR x, VALUE i, CHAN int[], ack) =
    ... ties the one-one buffers and q together
  PAR
    PAR j = [0 FOR 2]    -- one detector for each channel
      one.one.buff(con[j], int[j])
    controller(x, i, int, ack) :
```

A one-place, one-shot buffer is a very simple process.

```
PROC one.one.buff(CHAN left, right) =
  VAR y :
  SEQ
    left?y
    right!y :
```

The controller is a little more complicated.

```
PROC controller(VAR x, VALUE i, CHAN int[], ack) =
  SEQ
    ALT j = [0 FOR 2] -- for both channel detectors
      int[j]?ANY -- wait for first to achieve output
        PAR
          set.bit(x, i, j) -- i'th bit of x := j
          ack!ANY          -- acknowledge bit
          int[1-j]?ANY :   -- catch remaining communication
```

The global procedures

```
get.bit(VAR j, VALUE x, i) =
  ... j := i'th bit of x
```

and

```
set.bit(VAR x, VALUE i, j) =
  ... i'th bit of x := j
```

are easy to write in a purely sequential style and we do not give them here.

This algorithm is not the most efficient, in terms of channels or communications. With a change to the gross structure of q only two communications rather than three are required. If the number of communications is increased to four, then the acknowledgement channel can be dispensed with.