Interaction and Security in Distributed Computing

Lars Wulf

Wolfson College

Thesis submitted for the degree of
Doctor of Philosophy at the University of Oxford

Hilary Term, 1997

Interaction and Security in Distributed Computing

Abstract

Computer security is concerned with the preservation of critical system properties even in the face of attack by a hostile intruder. A secure system is typically required to prevent the unauthorized disclosure of information (confidentiality), to deny unwarranted attempts of modifying information (integrity), and to guarantee minimum levels of service delivery (availability).

In this thesis, the theory of Communicating Sequential Processes is employed to advance techniques for the modelling, development, and verification of systems with multi-level security requirements. We are primarily concerned with security conditions which guarantee *non-interference*, i.e. the absence of information flow between interfaces of a system.

We propose the condition of *independence* as the most suitable criterion for non-interference, since it can be equally applied to deterministic as well as non-deterministic systems while guaranteeing the absence of covert channels of information flow through exploitation of internal choices. In a bi-level system, mutual independence is equivalent to the symmetric condition of strong separability – the ability to express the system equivalently as the parallel composition of deterministic components with disjoint interfaces.

The cornerstone of our approach is the use of *process abstractions* to model the types of interaction between interfaces of a computational process. We investigate the properties of various forms of process abstraction, and find that independence is properly defined as maximality under suitable forms of abstraction. A collection of laws will be presented that allow the compositional development of secure systems.

The primary application areas of our modelling framework include computing systems shared concurrently by multiple users. The approach to system development is illustrated with case studies concerning the readers-writers problem, a secure multi-level file system, and a concurrency control algorithm for a secure database system.

Acknowledgements

First and foremost I would like to thank Bill Roscoe, my supervisor, for accepting me as one of his research students, and for providing invaluable encouragement and guidance throughout the course of this work. Having gone through several stages of refining the ideas, notation, and terminology of this thesis, it was his patience and support which kept me on track during its preparation.

The Computing Laboratory of Oxford University has provided a stimulating research environment. I am greatly indebted to several members of the Programming Research Group for their constructive comments on aspects of this work. Jim Woodcock provided the seminal insight, while the attendees of the Oxford security seminar series (which I had the pleasure of organising) served as a useful sounding board for subsequent ideas. Micheal Goldsmith and Jim Davies deserve thanks for detailed reading and commenting on parts of the thesis draft. I would also like to thank Ana, Axel, Jason, Jesús, Maurice, and Scott for their friendship and advice while sharing an office with me at the OUCL.

Wolfson College has been a most enjoyable and pleasant place to live and study, and I am grateful to the college principals for encouraging students to strike a balance of academic achievement, sporting success, and active social life.

I gratefully acknowledge the financial support of my parents as well as the Engineering and Physical Sciences Research Council (previously the Science and Engineering Research Council) in the form of a studentship (award no. 93313826), without which this work could not have been carried out.

On a personal note, I sincerely thank Mari for her unwavering love and support throughout our shared time and beyond.

"To honour the most high God,
and to benefit my neighbour."

Johann Sebastian Bach
Orgelbüchlein (ca. 1714)

Contents

Chapter 1

Introduction

1.1 Critical Systems

It is no exaggeration to state that our health and wealth depends to a large measure on the correct and efficient workings of computer systems. Such systems are increasingly employed in circumstances where serious consequences may be a result of either system failure or flawed system specification or implementation. Critical (computer) systems are those whose malfunction could have unacceptable consequences, such as loss of life, environmental damage, or the disclosure of sensitive information.

There are four, often very distinct, approaches to the development of critical systems [Rus94]. The concept of *dependability* grew out of the need to produce ultra-reliable systems for which reliance may justifiably be placed on their correctness and service delivery [Lap85]. In this framework, it is common to identify different types of failures (such as benign, catastrophic, byzantine etc.) and apply methods to detect, mask, or recover from them. *Safety engineering* is concerned with the external consequences of system failures. A cornerstone of the approach is hazard analysis; potential hazards are identified, and those considered unacceptable dealt with (e.g.) by the incorporation of safety features or devices. *Secure systems* are those which are required to preserve confidential information and safeguard privacy. Finally, in the field of *real-time systems* it is necessary to formulate timing constraints for computational tasks and implement systems in a manner guaranteed to meet them.

These four fields differ significantly in the system properties that are considered critical, the mechanisms employed to safeguard these properties, and the development methods to design and implement systems.

A good approach to understand the relationship between critical properties is to focus on very general attributes of these properties. Charles Perrow [Per84] based his analysis of safety-critical systems on two such attributes: interaction and coupling.

Interaction refers to the extent individual components of a system can affect the behaviour of other components. In a "linear" system, interaction occurs in a simple directed manner and can thus be more readily analysed. Distributed systems maintaining global conditions of correctness (i.e. consistency in databases) can exhibit "complex" interactions between logically or physically isolated parts. *Coupling* refers to the closeness of features or components of a system. According to Perrow, tight coupling can exacerbate system failures because it leaves less flexibility for countermeasures when a malfunction occurs. On the other hand, loosely-coupled components may cooperate in a flexible and safe but not efficient manner.

Security

Security is a concern that demands few interactions. In a secure information system, subjects are typically allocated security classifications which permit or deny them access to sensitive data. In the strictest possible interpretation of security, the different classifications should be dealt with by separate, non-interacting subsystems. With a more liberal interpretation, some interaction between subjects of different classifications may occur, but only in a very controlled, well-specified manner.

One of the first to describe the problem of unwanted paths of interaction was Lampson in his note on the confinement problem [Lam73]. Introducing the notion of a "covert channel", it describes how illegitimate information is transmitted at the same time with proper communication. Two main types of covert channels can be identified. With a *storage channel*, information is passed through the presence or absence of items in storage. With a *timing channel*, information is passed through the speed at which certain system events occur.

Subsequent research proposed techniques for information flow analysis in order to detect and avoid covert channels. Using a lattice of security levels in order to determine possible leakage of information to lower levels, flow analysis has been applied to the validation of operating system kernels [Mil76] as well as to application programs [Den76].

In the 1980s, researchers at SRI investigated the foundations of security using abstract state-machine representations. Based on earlier work by Feiertag et al. [FLR77], a seminal paper by Goguen and Meseguer [GM82] proposed a theory of non-interference as a technique for formal security analysis, later applied to verification of an operating system kernel [HY86, HKMY87]. The theory of non-interference was elegant and deep enough to avoid distinctions between covert channel analysis and access control. The verification of non-interference may be achieved by constraining individual transitions between the states of the system model, a technique originally called "unwinding" [GM84] and later lucidly re-presented by Rushby [Rus85, Rus92].

In its original formulation, the non-interference analysis is only applied to deterministic systems. Subsequent research was aimed at developing non-interference and information flow models suitable for systems exhibiting non-deterministic behaviour. In one type of model it is assumed that probability distributions affect the outcome of non-deterministic choices. Some of these models are closely related to Shannon's classical theory of information, such as Millen's analysis of channel capacity [Mil87] and Wittbold and Johnson's theory [WJ90], developed in response to Sutherland's theory based on logical deduction [Sut86]. Logical characterisations of security in systems with probabilistic choices have been proposed by Gray and Syvverson [Gr91, Gr92, GS92].

The other type of model, sometimes referred to as *combinatoric*, does not assume that non-deterministic choices are taken in accordance with an underlying probability distribution. A prominent example is McCullough's "hook-up" security in the context of an event system model, later developed as a theory of restrictiveness. Restrictiveness has the desirable property of being composable, so that the parallel composition of two restrictive systems also satisfies this property. This work was extended with the discovery of a weaker condition with similar properties, forward correctability [JT88], which was later given an unwinding result by Millen [Mil95].

Systems and their security properties in combinatoric models are typically described in terms of traces or sets of traces, i.e. the sequences of events which capture the observable behaviour of the system. Process algebras such as CSP and CCS give precise meaning to systems described in terms of their traces and have thus become increasingly popular for research in computer security.

Early work by Foley [Fol87] demonstrated that CSP is well-suited to express conditions related to the absence of information flow. Subsequent work on computer security in a CSP framework has been carried out by Jacob [Jac87, Jac90, Jac90b], Allen [All91], Ryan [Rya91], Graham-Cumming [Gra92, Gra93], O'Halloran [O'H93], and Roscoe et al [RWW94, RW95, Ros95]. Whereas most approaches are closely related to the original theory of non-interference, Jacob follows a different approach by formalising what can be deduced from observed system behaviour in terms of inference functions. Building on Jacob's work, O'Halloran develops an abstract model of information flow within a category theory framework, which may be specialised to trace descriptions in CSP.

Focardi and Gorrieri [FG95] investigate definitions of security in the context of labeled transition systems. They classify several trace-based security formulations of information flow, and relate these to conditions expressed in terms of a finer notion of system behaviour (weak bisimulation).

1.2 Formal Methods

For every computing system, the basic requirement is to operate correctly. A secure system can be said to operate correctly if it can be relied upon to protect sensitive data or information as originally intended by the system designers. In order to state exactly what it means for a system to operate correctly, and under what circumstances this can be achieved, it is necessary to have a definition of security. Without a formal criterion of security, it is difficult to discuss correctness beyond superficial considerations. Furthermore, it becomes impossible to give convincing arguments of system correctness where the highest degree of assurance requires the use of mathematical techniques in the development of systems.

To overcome these difficulties, it has been suggested to employ formal methods of system development, i.e. techniques with a mathematical or logical foundation. A formal method is a notation or language with a specific syntax (grammar) and semantics giving meaning to the legal statements which can be expressed. Frequently a calculus is provided for manipulating the notation, thus aiding the reasoning about properties of the objects defined in the language. Using the calculus, it becomes possible to prove properties of objects with a degree of mathematical clarity that is not available using an informal development technique.

The benefits of the use of formal methods are well-documented [Smi92]. Using a formal method in conjunction with descriptions in natural language, a computing system can be described in a manner that is more precise and often more concise than with English alone. Many complex algorithms can be specified more accurately with a few lines of mathematical notation than with pages of informal descriptions, which may leave room for ambiguities. Furthermore, the use of formal methods can help to abstract from unnecessary detail, by focusing attention on *what* the system is intended to do, not *how* it is achieved. Starting from a very abstract specification, it is possible to progress through a series of refinement steps, each adding some implementation details, until a concrete system representation is derived which can be executed on a computer. Finally, a formal description of a system can serve as an effective and unambiguous medium of communication, both within a project team and between the developer and client.

Meadows [Mea95] observes that formal methods have a particularly important role to play in the context of secure system development:

> "A considerable amount of work has also gone into developing engineering techniques for secure systems as well. These include the exploration of different types of architectures, including specialised hardware and microkernels, the conversion of existing operating systems to secure ones, and the development of engineering techniques for assuring that the system is trustworthy. In this last, formal methods has traditionally played an important role in guaranteeing the trustworthiness in multi-level systems that require a high degree of assurance. Indeed, much of the research in formal methods has been funded by the security community."

On the negative side, there are doubts that all aspects of a complex computing system can be accurately captured. These doubts are well justified, particularly in the context of a complex distributed system with multiple hardware and software components. It could be argued that a complete and exhaustive system description cannot and should not be the aim of the use of formal methods: if the length and complexity of a formal description exceeds that of the actual implementation, then no benefit is gained. A formal method should be employed to capture the key aspects of central objects or algorithms, in order to state their properties precisely.

Stepwise Refinement

The concept of refinement lies at the heart of software engineering. In its informal sense, refinement refers to the process of adding further details or improvements to some artefact or piece of information. The effect of the refinement step is to be more precise about exactly what properties of the artefact can be relied upon. For example, the product information for a watch might claim that a particular watch is splash-proof. After extensive testing the watch manufacturer might claim that the watch is water-proof up to 10m. This more refined information gives more precise detail under which conditions the watch will still function correctly.

Starting from an abstract specification of the system or software, each step of the refinement process yields a more deterministic model which is closer to an actual implementation. A specification P serves as a kind of contract for the next step in the development process: a proposed implementation Q must improve on P, be more accurate, more predictable, and less likely to fail. In particular Q must never exhibit any behaviour outside of that allowed by P, since otherwise the specification is violated. If these conditions are satisfied then we can accept Q as a valid refinement of P.

Refining a specification P corresponds to the reduction of non-determinism caused by the underspecification of behavioural details. We would accept Q as a refinement of P if its range of behaviour is contained in that of P – in other words, if all possible behaviours of P and Q taken together are given by those of P. That is, we expect Q to refine P if $P = P \sqcap Q$, that is the specification P cannot be distinguished from the non-deterministic choice between specification and implementation.

Computational Frameworks

The choice of a suitable computational framework is critical to any attempt at modelling computer systems. The framework determines how to describe the states and state transitions of an automaton which serves as the model for critical software or hardware components. A poorly chosen framework will result in models which may be difficult to understand and relate to the artefacts whose properties they are supposed to reflect. A well chosen framework allows the software engineer to establish a close correspondence between the properties of the actual computing systems and their models.

Modelling frameworks can be broadly divided into two classes. In the state-based approach the main focus is on the modelling of system states, while in the behavioural approach the emphasis is on the possible events and actions. Abadi and Lamport [AL93] characterise these two classes as follows.

> "The popular approaches to specification are based on either states or actions. In a state-based approach, an execution of a system is viewed as a sequence of states, where a state is an assignment of values to some set of components. An action-based approach views an execution as a sequence of actions. These different approaches are, in some sense, equivalent. An action can be modeled as a state change, and a state can be modeled as an equivalence class of sequences of actions."

The behavioural or action-based approaches often have a strong connection to algebra. A specification is given by a set of algebraic laws and a description of how objects are manipulated algebraically. This type of approach includes function-call specifications as well as communication-event trace specifications, of which Hoare's CSP is the prime example.

Communicating Processes

Hoare's theory of Communicating Sequential Processes [Hoa85] provides a mathematical basis for representing systems composed of concurrent processes. It allows elegant descriptions of systems which have some inherent degree of concurrency, such as distributed systems with physically separate components.

A cornerstone of Hoare's approach is the precise treatment of refinement. A process Q is said to refine process P, written

$$P \sqsubseteq Q$$

if Q behaves in a more predictable and reliable manner than P, and is thus closer to being implementable. More precisely, Q refines P if Q exhibits only behaviours not disallowed by P, so that process Q will satisfy any behavioural specification that is also satisfied by P.

The theory incorporates various models of increasing sophistication depending on what process behaviour is considered relevant. Appendix 1 summarizes the main aspects of the "improved failures" model [BR84] where relevant behaviour includes traces, the communications refused after a given trace (failures), and the traces leading to infinite internal computation (divergences). This model allows the specification of safety and liveness properties of both deterministic and non-deterministic processes. The failures-divergences model of CSP will serve as our mathematical model of computation for the remainder of this thesis. Refinement in this model is given by reverse inclusion of observable behaviours, i.e. $P \sqsubseteq Q$ precisely if every behaviour possible for process Q is also possible for P:

$$\textsc{Fails}(Q) \subseteq \textsc{Fails}(P) \quad \wedge \quad \textsc{Divs}(Q) \subseteq \textsc{Divs}(P) \ .$$

Unless it is explicitly stated otherwise, we assume throughout that processes are free of divergence, since the occurrence of divergence is generally regarded as erroneous behaviour. We also do not consider the issue of termination of processes, whose precise mathematical treatment in the context of multiple concurrent processes would add complexity to our treatment. The assumption of divergence freedom and non-termination significantly streamlines the exposition of mathematical results of the coming chapters.

The refinement relation between processes can be verified automatically by means of the model checker FDR.[1] The availability of this proof tool has revolutionized the way CSP has been used in recent years, and been a major influence on the presentation of communicating processes in this thesis. The refinement checks and verification of security properties of all examples and case studies appearing in later chapters have been carried out using FDR.

[1] The FDR tool is a product of Formal Systems (Europe) Limited, 3 Alfred Street, Oxford OX1 3EH, United Kingdom. Details on the history, implementation, and use of FDR may be found in [FSE94, Ros94, Ros97].

Abstraction

Suppose it is required to verify whether a process Q provides a valid implementation of specification P, but Q is able to communicate events E which are impossible for P. A direct refinement check $P \sqsubseteq Q$ will fail, since any trace including an event from E is not possible for P. However, it is possible to express the correctness criterion in a different way, by taking account of the extra events. There are essentially two ways to deal with those events, by manipulating either the specification or the implementation process.

1. events E are added in the specification process P, for example by including E events in its traces, or
2. events E are removed from in the implementation Q, for example by removing E events from its traces.

There are several possibilities of how this may be achieved in practice. As an example of the second approach, one could simply prohibit the E events occurring, by insisting that process Q synchronizes their occurrence with process $Stop$, which is continuously inactive

$$P \quad \sqsubseteq \quad Q \mathbin{|[E]|} Stop \ .$$

Although the right-hand implementation will never be able to engage an event from E, this is rarely the correctness criterion desired, since the stopping of E may prohibit the occurrence of many other events as well. A frequently better solution is to hide the visible occurrence of E events, without stopping others

$$P \quad \sqsubseteq \quad Q \setminus E \ .$$

This form of concealment has the potential disadvantage of introducing divergence when process P is able to engage an unbounded sequence of E events. As an example of the first approach, one may use interleaving to add E events to the traces of P.

$$P \mathbin{|||} Run_E \quad \sqsubseteq \quad Q \ .$$

We regard these operations as instances of abstracting events in a process, which do not add anything of interest to the process description, but may remove computational details in some useful way. This theme of using different methods of abstraction is developed in Chapter 2.

1.3 Thesis Outline

In this thesis, we propose process abstractions as an effective way to state and analyze properties which restrict or exclude interaction between computational agents in models of distributed systems. In the following chapter, the notion of process abstraction is introduced, and characterised in terms of a number of properties such a function may satisfy. A number of concrete abstraction functions will be discussed, including stopping, hiding, strict projection, lazy abstraction, and interleaving. These will be used to define the condition of *robustness* which could prove useful in the modelling of fault-tolerant systems.

In Chapter 3 abstraction functions are used to state conditions for non-interference between two interfaces of a process. *Invariance* requires the possible observations within an interface to remain unchanged under the occurrence of events outside the interface. *Independence* is a stronger condition which requires the interface also to be deterministic. Independence can be characterised as determinism under application of suitable abstraction functions. Asymmetric security conditions are related to symmetric conditions of separation; it is shown that mutual independence corresponds to the ability to define the system as an interleaving of deterministic components. The chapter closes with remarks on how the independence criterion can be extended to take signal or output events into account.

In Chapter 4, compositional laws for a number of CSP operators are given, which ensure that the resulting processes satisfy our security conditions. Our approach to the construction of secure systems is then generalised to those with more than two computational agents. For multi-level systems, it is necessary to state the security policy which the agents have to satisfy. From the security policy, the required verifications conditions which have to be checked can be readily derived.

Chapters 5 to 7 present case studies to illustrate the range of applicability of the modelling framework. The primary area of application is the specification and design of computing systems shared between multiple users, such as operating systems, file systems, and multi-level transaction processing systems. In a wider perspective, notions based on non-interference and independence have been applied elsewhere to the modelling of safety-critical systems and authentication protocols; this work will be alluded to in Section 2.4 and Chapter 8.

Chapter 5 presents the case study of the readers-writers problem in a security context. In this problem, we are concerned with the conflict between the functional requirement of mutual exclusion, and the security requirement of non-interference. Processes may read data objects to the exclusion of processes attempting to write, and a process may write data objects to the exclusion of other processes attempting to read or write. Processes also have security classifications, and the system is to satisfy the security condition that processes with low classification must be completely unaffected by the actions of highly-classified processes.

The case study of a filesystem with multiple users in Chapter 6 will illustrate our approach to multi-level system. The model encompasses standard operations commonly implemented in storage systems: the creation and removal of files, the reading and writing of file values, and the renaming of files. A security policy for the system will be stated first, followed by a compositional development of the implementation. Finally it is shown that the system satisfies the policy.

Chapter 7 presents a further case study on the problem of transaction scheduling for secure multi-level database systems. Individual operations of transactions are scheduled to provide consistent view of data, but this requirement for concurrency atomicity may conflict with the requirement that transactions with different classifications must not interfere with each other. We consider a scheduling algorithm using timestamps which is both secure and results in serializable schedules.

Chapter 8 concludes this thesis with a review and comparative remarks on previous results in the field of the formal modelling of secure systems. We focus mainly on combinatoric theories for the modelling of standard deterministic and non-deterministic systems, allowing a direct comparison with our theory of independence. We also investigate the application of independence conditions in a probabilistic setting, with the aim of extending the scope of our approach to include systems exhibiting random behaviour. The final section summarizes our work, highlights its distinguishing features, and outlines possible areas of further research.

The Appendix on page 146 serves as reference for the notation used in this thesis and the failures-divergences model of CSP on which this work is based.

Chapter 2

Interaction

In this chapter we consider functions on processes which allow us to make statements about process behaviour without reference to the whole set of events the process can communicate. The effect of these functions, which we call *process abstractions*, is to remove computational detail from a process. Frequently, the purpose of using these functions is to describe how the interactions with the process within some set of events affect the possible interactions and observations in another.

Section 2.1 presents some general algebraic properties we expect an abstraction function to satisfy. In CSP, there are a number of concrete functions which meet these criteria, and these are described in Section 2.2. In the final section, a condition for non-interaction, expressed in terms of process abstractions, is given.

2.1 Process Abstraction

A process abstraction is a function on processes which may remove computational detail from a process. The process behaviours removed by application of an abstraction function may correspond to behaviours whose action or events cannot directly be observed, even if their overall effect can. Suitably chosen abstractions can be used to model how some external agent interacting with a process perceives this interaction in the context of other agents' concurrent interaction. In order to facilitate a meaningful discussion of this type of interaction, we will make the following assumptions and conventions.

There is an overall system alphabet Σ which captures the complete set of communication events which individual processes of the system may engage in. The set of possible communications for a process P, a subset of Σ, is called the alphabet of the process, conventionally written αP. The alphabet describes the scope of interaction of the process with other concurrent processes through synchronization, or with some agent in the system environment. An interface to process P is a non-empty subset of its alphabet. There may be multiple interfaces associated with a single process.

A basic assumption we will make is that each external agent interacting with a process can control or observe the events in its interface. Events outside the interface cannot be directly observed, although it is possible to make deductions about their occurrence by analysing what can and cannot be observed in this interface. Process abstractions make it possible to describe formally how interactions by some agent in one interface affects another interface. Below we attempt to capture some of the general algebraic properties we may expect a process abstraction to enjoy.

Basic Properties

In this subsection, we list a number of algebraic properties which a process abstraction $\mathcal{A}_H(P)$, which removes computational details associated with the occurrence of events in interface H in process P, may or may not enjoy. These properties below are not intended as a formal definition of abstraction, but serve as a collection of desirable properties which may serve as a foundation for comparing various concrete forms of abstraction.

A1. Abstracting the empty set leaves the process unchanged:

$$H = \emptyset \quad \Rightarrow \quad \mathcal{A}_H(P) = P \ .$$

A2. The order of abstracting two sets $H1$ and $H2$ does not matter:

$$\mathcal{A}_{H2}(\mathcal{A}_{H1}(P)) \quad = \quad \mathcal{A}_{H1 \cup H2}(P) \ ,$$

and consequently abstracting the same set more than once has no further effect: $\mathcal{A}_H(\mathcal{A}_H(P)) = \mathcal{A}_H(P)$.

A3. Under abstraction, it does not matter how non-deterministic P is with regard to H events:

$$\mathcal{A}_H(P) \;=\; \mathcal{A}_H(P \mathbin{|[H]|} Chaos_H) \;,$$

in which case we say that the H refusals are transparent under $\mathcal{A}_H(.)$.

A4. An interleaved user process U_H, communicating in H only, leaves the abstract process unchanged:

$$\mathcal{A}_H(P \mathbin{|||} U_H) \;=\; \mathcal{A}_H(P) \;.$$

A5. Abstraction distributes over internal choice:

$$\mathcal{A}_H(P \sqcap Q) \;=\; \mathcal{A}_H(P) \sqcap \mathcal{A}_H(Q) \;.$$

Any concrete abstraction function which satisfies property **A5** will preserve the refinement order: if $R \sqsubseteq S$, that is $R = R \sqcap S$, then

$$\begin{aligned}\mathcal{A}_H(R) &= \mathcal{A}_H(R \sqcap S)\\ &= \mathcal{A}_H(R) \sqcap \mathcal{A}_H(S)\end{aligned}$$

by distribution of $\mathcal{A}_H(.)$, which means that $\mathcal{A}_H(R) \sqsubseteq \mathcal{A}_H(S)$.

Whether or not a concrete abstraction function satisfies the above conditions depends on the computational model in which the analysis takes place. Even in the simplest model for CSP, the traces model, there are few functions that satisfy all conditions simultaneously. In more detailed computational models, some concrete functions retain certain properties while other functions lose them. A comparison of these algebraic properties allows us to distinguish between concrete functions, and discard any not deemed suitable for a particular purpose.

The application of an abstraction function to a process usually has the effect of introducing or increasing the amount of non-determinism in the process. The non-determinism is caused by the removal of computational details in one interface, thus leaving some of the overall process behaviour underspecified. If the overall behaviour is to a high degree dependent on the activity in some interface H, then a high degree of non-determinism can be expected to be the result of abstracting H.

Process abstractions make it possible to describe the degree of interaction between one interface and the overall process behaviour. Given a suitable abstraction function, we can define an interaction order between processes with respect to some interface H.

Definition 2.1.1 Interface H has *less interaction* in Q than P if, and only if, the abstract process $\mathcal{A}_H(Q)$ refines $\mathcal{A}_H(P)$.

$$P \leq_H^{\mathcal{A}} Q \quad \Leftrightarrow \quad \mathcal{A}_H(P) \sqsubseteq \mathcal{A}_H(Q)\ .$$

There is a different interaction order for each method of abstraction. Two processes are regarded as equivalent with respect to interface H under the interaction order if the process abstraction has the same effect in both processes.

Definition 2.1.2 Interface H has the same amount of interaction in P and Q if, and only if, both abstract processes are equivalent

$$P \equiv_H^{\mathcal{A}} Q \quad \Leftrightarrow \quad \mathcal{A}_H(P) = \mathcal{A}_H(Q)\ .$$

The observations and behaviours possible in any one of the interfaces of a process depend on the complete process definition. The application of a process abstraction can therefore be used not only to model the amount of interaction with the overall behaviour, but also to investigate the effect of events in one process interface on the possible behaviours in another. In particular we will use concrete process abstractions to investigate the interactions between interface H (high) and interface L (low), which are assumed to partition the system alphabet Σ.

2.2 Concrete Methods of Abstraction

A number of concrete methods of abstraction will be presented in the following subsections, namely stopping, hiding, strict projection, lazy abstraction, and interleaving.

Stopping

One of the simplest methods to abstract H activity is to prohibit it from occurring. This is achieved by requiring the process to synchronize on H with process $Stop$ which is continually unable to engage any event.[1]

Definition 2.2.1 Given a process P, the *stopping* of H in P is defined by

$$\mathcal{S}_H(P) \quad \widehat{=} \quad P \,|[H]|\, Stop\ .$$

In the parallel composition $P\ |[H]|\ Stop$ both component processes have to synchronize on each occurrence of an event from H. Since $Stop$ can never engage any event, the effect of the composition is to prohibit the occurrence of H events. Therefore every trace of $\mathcal{S}_H(P)$ can contain only events outside H.

Example 2.2.2 Let $H = \{x\}$ in

$$\begin{array}{lcl} P_1 & \widehat{=} & c \to (x \to P_1 \ \Box\ a \to P_1) \\ & & \Box\ a \to b \to P_1\ , \\ P_2 & \widehat{=} & c \to x \to P_2 \\ & & \Box\ a \to b \to P_2\ . \end{array}$$

Stopping H in P_1 has the effect of disabling x without affecting the complement interface:

$$\begin{array}{lcl} \mathcal{S}_H(P_1) & = & c \to a \to \mathcal{S}_H(P_1) \\ & & \Box\ a \to b \to \mathcal{S}_H(P_1)\ . \end{array}$$

In P_2, however, the disabling of x results in deadlock after the choice is made in favour of event c; for if $s \frown \langle c \rangle \in \textsc{Traces}(P_2)$ then no further events will be accepted after this trace

$$(\mathcal{S}_H(P_2)\ \underline{\text{after}}\ s \frown \langle c \rangle)^0 = \emptyset\ .$$

(End of Example)

[1] If process termination is to be taken into full consideration, then it is necessary to define the stopping abstraction in terms of process $Skip$ rather than $Stop$.

The example illustrates a feature of stopping which may often be undesirable: abstracting H can greatly reduce the trace set of the process, or even result in deadlock after particular traces. By inhibiting the occurrence of events, stopping can significantly reduce the number of states the process can reach. Since those states reached through H events may be of particular interest, stopping cannot usually be regarded as a suitable method of abstraction.

However, stopping enjoys a number of algebraic properties which have been described earlier in this section. The following properties follow directly from the definition of the parallel operator, given in the Appendix on page 153.

Proposition 2.2.3 Stopping the empty set in process P has no effect

$$H = \emptyset \quad \Rightarrow \quad \mathcal{S}_H(P) = P \ ,$$

the order of stopping two sets $H1$ and $H2$ of events is irrelevant

$$\mathcal{S}_{H2}(\mathcal{S}_{H1}(P)) \quad = \quad \mathcal{S}_{H1 \cup H2}(P) \ ,$$

the refusals of H in process P are transparent under stopping

$$\mathcal{S}_H(P) \quad = \quad \mathcal{S}_H(P \ |[H]| \ Chaos_H) \ ,$$

and stopping distributes over internal choice

$$\mathcal{S}_H(P \sqcap Q) \quad = \quad \mathcal{S}_H(P) \sqcap \mathcal{S}_H(Q) \ .$$

Any user process U_H whose behaviours are interleaved with P does not affect the abstract process.

Proposition 2.2.4 If process U_H communicates within H only, then

$$\mathcal{S}_H(P \ ||| \ U_H) \quad = \quad \mathcal{S}_H(P) \ .$$

Stopping has only limited applicability as an abstraction function, since it can drastically reduce the set of traces of the process to which it is applied. What is needed is an abstraction function which, rather than prohibiting events from occurring, conceals their occurrence.

Hiding

The primary method of abstraction traditionally used in program development is the hiding (concealment) of events [Hoa85, page 111]. The effect of hiding set H in P is to remove all H events from the traces of P. The abstraction $P \setminus H$ denotes a process behaving like P, except that

- H events occur as soon as they are enabled, and
- only events outside H may appear in a trace.

Definition 2.2.5 Given a process P, the abstraction by hiding is defined

$$\mathcal{H}_H(P) \quad \widehat{=} \quad P \setminus H \ .$$

Example 2.2.6 Let $\Sigma \mathrel{\widehat{=}} \{x, y, a, b\}$ and $H = \{x, y\}$ in

$$\begin{aligned} P \quad \widehat{=} \quad & x \to (a \to P \,\square\, b \to P) \\ & \square\; y \to (a \to P \,\square\, b \to P) \ . \end{aligned}$$

Hiding H allows us to ignore the particular choice of event (x or y), since neither will appear in a trace of the abstraction. $\mathcal{H}_H(P) = P'$ where

$$P' \quad \widehat{=} \quad a \to P' \,\square\, b \to P' \ .$$

(End of Example)

In the example, hiding does not introduce deadlock, because concealed events will occur as soon as possible, according to the underlying model of computation. Process $\mathcal{H}_H(P)$ cannot refuse to engage in activity outside H until it has reached a state in which it cannot engage in further hidden events. The progress of H activity under hiding can thus be regarded as swift or immediate, and for this reason hiding has been termed an $eager$ form of abstraction [RWW94, Ros95].

The following properties follow directly from the definition of the hiding operator, whose semantics is given in the Appendix on page 152.

Proposition 2.2.7 Hiding the empty set has no effect

$$H = \emptyset \;\;\Rightarrow\;\; \mathcal{H}_H(P) = P \;,$$

the order of hiding two sets $H1$ and $H2$ does not matter

$$\mathcal{H}_{H2}(\mathcal{H}_{H1}(P)) \;\;=\;\; \mathcal{H}_{H1 \cup H2}(P) \;,$$

abstraction by hiding distributes over internal choice

$$\mathcal{H}_H(P \sqcap Q) \;\;=\;\; \mathcal{H}_H(P) \sqcap \mathcal{H}_H(Q) \;.$$

There are several drawbacks in using concealment (hiding) in modelling interaction. The first is the introduction of divergence when hiding continuous, unbounded behaviour. The second is closely related to the assumption that concealed behaviour occurs swiftly; it is that hiding does not satisfy transparency of refusals (property $\mathbf{A3}$). Consider the process

$$P \;\;\widehat{=}\;\; x \rightarrow a \rightarrow Stop$$

where $H \mathrel{\widehat{=}} \{x\}$. The property is not satisfied because event a cannot be refused under hiding:

$$\begin{aligned} P \setminus H \;\;&=\;\; a \rightarrow Stop, \;\;\text{but} \\ (P \;|[H]|\; Chaos_H) \setminus H \;\;&=\;\; Stop \sqcap a \rightarrow Stop \;. \end{aligned}$$

Hiding is appropriate for situations where the computational behaviour from which we wish to abstract

- is assumed to occur without delays and
- is either guaranteed not to occur unboundedly, or desired to be detectable should this be the case.

Strict Projection

Strict projection may be regarded as a form of hiding for which the H refusals are transparent, in contrast to the pure hiding operator described in the previous subsection. Transparency of refusals is achieved by ensuring that every subset of H can be refused before the hiding operator is applied.

Definition 2.2.8 Given a process P, abstraction from H by strict projection is defined

$$\mathcal{P}^*_H(P) \quad \widehat{=} \quad (P \,|[H]|\, Chaos_H) \setminus H \ .$$

Abstraction by strict projection is characterised by the following property, which says that if a behaviour $(s, X \cap L)$ of P may be observed through the system alphabet Σ then the restricted behaviour $(s \restriction L, X)$ of the abstract process $\mathcal{P}^*_H(P)$ may be observed through interface L.

Proposition 2.2.9 If P is a process, then

$$(s, X \cap L) \in \textsc{Fails}(P) \quad \Rightarrow \quad (s \restriction L, X) \in \textsc{Fails}(\mathcal{P}^*_H(P)) \ .$$

The following algebraic properties follow directly from the definition of the parallel operator and hiding operators, given in the Appendix.

Proposition 2.2.10 Strict projection of the empty set in P has no effect

$$H = \emptyset \quad \Rightarrow \quad \mathcal{P}^*_H(P) = P \ ,$$

the order of abstracting two sets $H1$ and $H2$ is irrelevant

$$\mathcal{P}^*_{H2}(\mathcal{P}^*_{H1}(P)) \quad = \quad \mathcal{P}^*_{H1 \cup H2}(P) \ ,$$

and strict projection distributes over internal choice

$$\mathcal{P}^*_H(P \sqcap Q) \quad = \quad \mathcal{P}^*_H(P) \sqcap \mathcal{P}^*_H(Q)$$

and thus preserves the refinement order $\sqsubseteq$.

Proposition 2.2.11 If P does not communicate in H, then

$$\mathcal{P}^*_H(P) \quad = \quad P \ .$$

Strict projection satisfies additional properties concerning the parallel composition of processes, which we discuss below. Composing P and Q in parallel and then abstracting will result in a refinement of the process where the abstraction takes place before the composition.

Proposition 2.2.12 Assume that Σ is partitioned by H, L, and that process P communicates in A, and Q communicates in B. Then

$$\mathcal{P}^*_H(P)\,[A\|B]\,\mathcal{P}^*_H(Q) \quad \sqsubseteq \quad \mathcal{P}^*_H(P\,[A\|B]\,Q) \ ,$$

and if additionally $A \cap B \cap H = \emptyset$, then

$$\mathcal{P}^*_H(P)\,[A\|B]\,\mathcal{P}^*_H(Q) \quad = \quad \mathcal{P}^*_H(P\,[A\|B]\,Q) \ .$$

Proof We will prove the second part of the result, under the assumption that $A \cap B \cap H = \emptyset$.

$$\begin{array}{ll}
 & \mathcal{P}^*_H(P)\,[A\|B]\,\mathcal{P}^*_H(Q) \\
= & \qquad\qquad\qquad\qquad [\ \text{Definition 2.2.8}\ ] \\
 & (P\ |[H]|\ Chaos_H) \setminus H\ \ [A\|B] \\
 & \ \ (Q\ |[H]|\ Chaos_H) \setminus H \\
= & \qquad\qquad [\ \text{Assumption on alphabets } A, B\ ] \\
 & (P\ |[H]|\ Chaos_H) \setminus (H \cap A)\ \ [A\|B] \\
 & \ \ (Q\ |[H]|\ Chaos_H) \setminus (H \cap B) \\
= & \qquad\qquad [\ A \cap B \cap H = \emptyset;\ \text{Equation } (A.7)\ ] \\
 & ((P\ |[H]|\ Chaos_H)\,[A\|B]\,(Q\ |[H]|\ Chaos_H)) \setminus H \\
= & \qquad\qquad [\ \text{Property of parallel composition}\ ] \\
 & ((P\,[A\|B]\,Q)\ |[H]|\ Chaos_H)) \setminus H \\
= & \qquad\qquad\qquad\qquad [\ \text{Definition 2.2.8}\ ] \\
 & \mathcal{P}^*_H(P\,[A\|B]\,Q) \ ,
\end{array}$$

as required. □

The second part of Proposition 2.2.12 shows that strict projection distributes over parallel composition, provided that the two processes involved in the composition do not synchronize on any H events. This insight can also be exploited where the synchronized parallel operator $|[S]|$ is used. In the case where no H events are members of the set S of events on whose occurrence both processes must agree, strict projection distributes over parallel.

Proposition 2.2.13 Assume that Σ is partitioned by H, L. Then

$$\mathcal{P}^*_H(P)\ |[S]|\ \mathcal{P}^*_H(Q) \quad \sqsubseteq \quad \mathcal{P}^*_H(P\ |[S]|\ Q) \ ,$$

and if S is a set of events such that $S \cap H = \emptyset$, then

$$\mathcal{P}^*_H(P)\ |[S]|\ \mathcal{P}^*_H(Q) \quad = \quad \mathcal{P}^*_H(P\ |[S]|\ Q) \ .$$

Since interleaving is a form of parallel composition without synchronization, strict projection therefore distributes over $|||$.

Corollary 2.2.14 For any processes P and Q,

$$\mathcal{P}^*_H(P) \mathbin{|||} \mathcal{P}^*_H(Q) \quad = \quad \mathcal{P}^*_H(P \mathbin{|||} Q) \; .$$

A user process running concurrently with process P makes the strict projection more deterministic:

Proposition 2.2.15 If U_H is a user process communicating within H only, then

$$\mathcal{P}^*_H(P) \quad \sqsubseteq \quad \mathcal{P}^*_H(P \mathbin{|[H]|} U_H) \; .$$

Proof

$$\begin{array}{ll}
 & \mathcal{P}^*_H(P) \\
= & \hspace{12em} [\text{ Definition 2.2.8 }] \\
 & (P \mathbin{|[H]|} \mathit{Chaos}_H) \setminus H \\
\sqsubseteq & \hspace{12em} [\text{ Refinement of } \mathit{Chaos}_H \text{ }] \\
 & (P \mathbin{|[H]|} (U_H \mathbin{|[H]|} \mathit{Chaos}_H)) \setminus H \\
= & \hspace{12em} [\text{ Associativity } (A.9) \text{ }] \\
 & ((P \mathbin{|[H]|} U_H) \mathbin{|[H]|} \mathit{Chaos}_H) \setminus H \\
= & \hspace{12em} [\text{ Definition 2.2.8 }] \\
 & \mathcal{P}^*_H(P \mathbin{|[H]|} U_H)
\end{array}$$

□

A special case is given by the user process $U_H \mathrel{\widehat{=}} \mathit{Chaos}_H$. In fact, processes P and $P \mathbin{|[H]|} \mathit{Chaos}_H$ cannot be distinguished under strict projection, i.e. the refusals in H are transparent.

Proposition 2.2.16 The refusals of H in P are transparent under strict projection

$$\mathcal{P}^*_H(P) \quad = \quad \mathcal{P}^*_H(P \mathbin{|[H]|} \mathit{Chaos}_H) \; .$$

Proof Let P be a process. Then

$$
\begin{array}{ll}
 & \mathcal{P}^*_H(P \;|[H]|\; Chaos_H) \\
= & \hfill [\ \text{Definition 2.2.8}\ ] \\
 & ((P \;|[H]|\; Chaos_H) \;|[H]|\; Chaos_H) \setminus H \\
= & \hfill [\ \text{Associativity } (A.9)\ ] \\
 & (P \;|[H]|\; (Chaos_H \;|[H]|\; Chaos_H)) \setminus H \\
= & \hfill [\ \text{Defn of } Chaos_H\ ] \\
 & (P \;|[H]|\; Chaos_H) \setminus H \\
= & \hfill [\ \text{Definition 2.2.8}\ ] \\
 & \mathcal{P}^*_H(P)\ .
\end{array}
$$

□

Strict projection enjoys a number of pleasant properties we would expect an abstraction function to satisfy, such as distribution over internal choice, parallel composition, and interleaving. Furthermore, H-refusals are transparent under strict projection. However, there remain properties which are highly desirable for a general abstraction function but cannot be satisfied by strict projection because of its ability to introduce divergence. For example, if P is a process communicating within interface H only, we would expect the abstraction of H to result in $Stop$, the silent process. In general $\mathcal{P}^*_H(P) = Stop$ does *not* hold, because $\mathcal{P}^*_H(P)$ cannot be guaranteed to be free of divergence (even if P is). A variant of strict projection without these shortcomings is required.

Lazy Abstraction

The motivation for lazy abstraction lies in our desire for a method of abstraction which has essentially the same effect as strict projection, but without the property of introducing divergence in the presence of unbounded H behaviour. Like strict projection, this form of abstraction effectively reduces the alphabet of the process it is applied to. The definition of lazy abstraction is based on the projection operator $P@L$, denoting the process whose behaviour is like that of P but whose alphabet has been reduced to interface L.

Definition 2.2.17 Let P be a process and $L \subseteq \Sigma$. The projection of P onto interface L, written $P@L$, is defined

$$
\begin{aligned}
\textsc{Fails}(P@L) &\;\widehat{=}\; \{\,(s \upharpoonright L, X) \mid (s, X \cap L) \in \textsc{Fails}(P)\,\} \cup \\
&\phantom{\;\widehat{=}\;} \{\,(s, X) \mid s \in \textsc{Divs}(P@L)\,\} \\
\textsc{Divs}(P@L) &\;\widehat{=}\; \{\,(s \upharpoonright L) ^\frown t \mid s \in \textsc{Divs}(P)\,\}\ .
\end{aligned}
$$

It is clear from this definition that $P@L$ cannot engage in events outside interface L, unless it diverges. If Σ is partitioned by sets H and L, then abstraction of H is carried out by projecting onto L. In general, the projection will be on the complement of the abstracted interface within the total set of communications, that is $\Sigma \setminus H$.

Definition 2.2.18 Let P be a process. Abstraction of H by projection is defined

$$
\mathcal{L}_H(P) \quad \widehat{=} \quad P@(\Sigma \setminus H)\ .
$$

Abstraction using projection has been characterised as a "lazy" form of abstraction. Unlike hiding, it does not require the whole of H to be refused in order to generate a refusal in interface L – that is, an observer of interface L is potentially affected by delays in another interface. To emphasize this effect of delays in H on the refusals in L, we will forthwith refer to $\mathcal{L}_H(P)$ simply as "lazy" abstraction.

Example 2.2.19 Let $\Sigma \;\widehat{=}\; \{x, y, a, b\}$ with $H \;\widehat{=}\; \{x, y\}$ and $L \;\widehat{=}\; \{a, b\}$. Process P below offers events x and then a, followed by a choice between events b and y. If y occurs, then a further a event is possible

$$
\begin{aligned}
P \quad \widehat{=} \quad x \rightarrow a \rightarrow (&b \rightarrow \mathit{Stop} \\
\Box\ &y \rightarrow a \rightarrow \mathit{Stop})\ .
\end{aligned}
$$

Abstracting $\{x, y\}$ is carried out by projecting onto the complement interface, that is $\mathcal{L}_H(P) = P@\{a, b\}$ where

$$
\begin{aligned}
P@\{a, b\} \quad \widehat{=} \quad \mathit{Stop} \sqcap a \rightarrow (&b \rightarrow \mathit{Stop} \\
\sqcap\ &a \rightarrow \mathit{Stop})\ .
\end{aligned}
$$

The non-determinism in $\mathcal{L}_H(P)$ reflects the lack of knowledge and control over the activity in H. First, there is the potential delay of event x, which causes the non-deterministic refusal of a. Secondly, interface L is affected by the possible occurrence of event y after a. As long as y is delayed, there is the possibility of engaging in b in L. However if event y proceeds then this option is removed from the interface and the next event can only be a. (End of Example)

Within the failures-divergences model of CSP, it is not conveniently possible to express lazy abstraction in terms of the standard operators on processes, in contrast to the abstraction functions described earlier in this chapter. However, an algebraic characterisation becomes possible in a more detailed model which distinguishes between the finite and the infinite traces of a process. The infinite traces model of CSP is summarized in the Appendix (page 146). In this model, the divergences under hiding are solely derived from the infinite traces of the process. Consequently no divergence will be introduced if this set of infinite traces is empty.

As in strict projection, we add H refusals to P by parallel composition with a chaotic process which can engage any event and refuse any set. However, the use of process $FChaos_H$ which behaves just like $Chaos_H$ but in contrast has no infinite traces, ensures that the process abstraction is free of divergence provided the original process is. Therefore, lazy abstraction $\mathcal{L}_H(P)$ can be equivalently described as

$$(P \mathbin{|[H]|} FChaos_H) \setminus H \ .$$

This representation will yield the correct set of failures and divergences for lazy abstraction, but not necessarily the expected set of infinite traces. A better representation of lazy abstraction within the infinite traces model is discussed by Roscoe [Ros97].

Strict projection has the same effect as lazy abstraction, except when applied to processes which permit an unbounded number of adjacent H events to occur. With lazy abstraction, there is no possibility of introducing divergence through abstraction. For example, let $H \mathrel{\widehat{=}} \{x\}$ in

$$\begin{array}{lcl} Q_1 & \widehat{=} & a \to Q_2 \\ Q_2 & \widehat{=} & a \to b \to Q_2 \\ & & \Box \ b \to a \to Q_2 \\ & & \Box \ x \to Q_2 \end{array}$$

where an a event occurs first, before the a and b events alternate in the sense that the total number of a events in a trace of Q_1 may not exceed the number of b events by more than 2. The x event may occur intermittently without affecting this pattern.

Whereas strict projection will show up the possibility of an unbounded number of x events after the initial a as divergence

$$\langle a \rangle \in \mathrm{DIVS}(\mathcal{P}^*_H(Q_1)) \ ,$$

lazy abstraction will ignore this possibility. The result of lazy abstraction is the non-diverging process $\mathcal{L}_H(Q_1) \mathrel{\widehat{=}} Q_1'$ where

$$\begin{array}{lcl} Q_1' & \widehat{=} & a \to Q_2' \\ Q_2' & \widehat{=} & a \to b \to Q_2' \ \Box \ b \to a \to Q_2' \ . \end{array}$$

Lazy abstraction enjoys a number of pleasant algebraic properties.

Proposition 2.2.20 Lazily abstracting the empty set leaves the process unchanged

$$H = \emptyset \quad \Rightarrow \quad \mathcal{L}_H(P) = P \ ,$$

and the order of projecting does not matter

$$\mathcal{L}_{H_2}(\mathcal{L}_{H_1}(P)) \quad = \quad \mathcal{L}_{H_1 \cup H_2}(P) \ ,$$

lazy abstraction distributes over internal choice

$$\mathcal{L}_H(P \sqcap Q) \quad = \quad \mathcal{L}_H(P) \sqcap \mathcal{L}_H(Q) \ ,$$

and therefore preserves the refinement order $\sqsubseteq$.

If P is a process which can engage in events in interface H only, then abstracting this set will result in a process which is unable to communicate at all.

Proposition 2.2.21 If process P communicates within H only, then

$$\mathcal{L}_H(P) \quad = \quad \mathit{Stop} \ .$$

The proposition above shows the dramatic effects of abstracting all events the process can communicate. In contrast, if lazy abstraction is applied only to events the process cannot engage in, then the abstraction has no effect on process behaviour. Thus $\mathcal{L}_H(P) = P$ whenever H and the total set of P's communications are disjoint.

Proposition 2.2.22 If P does not communicate in H, then

$$\mathcal{L}_H(P) \quad = \quad P \; .$$

Lazy abstraction satisfies the analogous result to Proposition 2.2.12, which shows how strict projection partially distributes over parallel composition. Composing P and Q in parallel first and then abstracting will result in a refinement of the process where the lazy abstraction is carried out before the composition.

Proposition 2.2.23 Assume that Σ is partitioned by H, L, and that P communicates in A, and Q communicates in B. Then

$$\mathcal{L}_H(P) \, [A \| B] \, \mathcal{L}_H(Q) \quad \sqsubseteq \quad \mathcal{L}_H(P \, [A \| B] \, Q) \; ,$$

and if additionally $A \cap B \cap H = \emptyset$, then

$$\mathcal{L}_H(P) \, [A \| B] \, \mathcal{L}_H(Q) \quad = \quad \mathcal{L}_H(P \, [A \| B] \, Q) \; .$$

When the parallel composition is expressed using the synchronized parallel operator $|[S]|$, we have the following result which is analogous to Proposition 2.2.13.

Proposition 2.2.24 Assume that Σ is partitioned by H, L. Then

$$\mathcal{L}_H(P) \; |[S]| \; \mathcal{L}_H(Q) \quad \sqsubseteq \quad \mathcal{L}_H(P \; |[S]| \; Q) \; ,$$

and if S is a set of events such that $S \cap H = \emptyset$, then

$$\mathcal{L}_H(P) \; |[S]| \; \mathcal{L}_H(Q) \quad = \quad \mathcal{L}_H(P \; |[S]| \; Q) \; .$$

A number of crucial properties follow immediately from Proposition 2.2.24. The first is that lazy abstraction distributes over interleaving.

Corollary 2.2.25 For any processes P and Q,

$$\mathcal{L}_H(P) \; ||| \; \mathcal{L}_H(Q) \quad = \quad \mathcal{L}_H(P \; ||| \; Q) \; .$$

Any interleaved user process U_H leaves the abstract process unchanged.

Proposition 2.2.26 If process U_H communicates within H only, then

$$\mathcal{L}_H(P \mid\mid\mid U_H) \quad = \quad \mathcal{L}_H(P) \ .$$

Proof From Corollary 2.2.25 and Proposition 2.2.21. □

A concurrent user process U_H which synchronizes with process P makes the abstract process more deterministic.

Proposition 2.2.27 If U_H is a user process communicating within H only, then

$$\mathcal{L}_H(P) \quad \sqsubseteq \quad \mathcal{L}_H(P \mid[H]\mid U_H) \ .$$

A special case is given by the user process $U_H \mathrel{\widehat{=}} \mathit{Chaos}_H$. In fact, processes P and $P \mid[H]\mid \mathit{Chaos}_H$ cannot be distinguished under lazy abstraction, i.e. the refusals of H events are transparent.

Proposition 2.2.28 The H refusals in P are transparent under lazy abstraction:

$$\mathcal{L}_H(P) \quad = \quad \mathcal{L}_H(P \mid[H]\mid \mathit{Chaos}_H) \ .$$

Proof We have $\mathcal{L}_H(P) \sqsubseteq \mathcal{L}_H(P \mid[H]\mid \mathit{Chaos}_H)$ by Proposition 2.2.27.

Refinement in the other direction also holds, because $P \mid[H]\mid \mathit{Chaos}_H \sqsubseteq P$ in general, and lazy abstraction preserves the refinement order. □

The following proposition shows the effect of removing the events L by lazy abstraction from a process P. The resulting process, $\mathcal{L}_L(P)$, is a refinement of process Chaos_H, the most non-deterministic process which is free of divergence. It follows that $\mathcal{L}_L(P)$ can communicate only events within H and cannot diverge, provided the original process P does not diverge.

Proposition 2.2.29 If Σ is partitioned by H and L, then

$$\mathit{Chaos}_H \quad \sqsubseteq \quad \mathcal{L}_L(P) \ .$$

The following proposition illustrates how lazy abstraction removes detail about the computational behaviour of a process. If a process P has two interfaces H and L, and the abstraction is applied to both interfaces, then the result will be two processes: $\mathcal{L}_H(P)$ which communicates in L, and $\mathcal{L}_L(P)$ which communicates in H. The concurrent interleaving of these processes can serve as a specification for P, in the sense that P is a valid refinement of the interleaving $\mathcal{L}_H(P) \mid\mid\mid \mathcal{L}_L(P)$.

Proposition 2.2.30 If Σ is partitioned by H, L, then

$$\mathcal{L}_H(P) \mid\mid\mid \mathcal{L}_L(P) \quad \sqsubseteq \quad P\ .$$

There is an asymmetric variant of Proposition 2.2.30 which says that abstracting H and synchronization on the complement interface also provides a nondeterministic specification for process P.

Proposition 2.2.31 For any process P,

$$\mathcal{L}_H(P) \mathbin{|[L]|} P \quad \sqsubseteq \quad P\ .$$

Lazy abstraction is in many ways the most suitable method of modelling interaction, as it enjoys a large number of desirable algebraic properties and does not suffer from the possibility of introducing divergence. Its definition, however, is based on projection $@L$, which is not among the standard CSP operators, and there does not appear to be an entirely satisfactory way of expressing lazy abstraction in terms of standard operators. This method of abstraction can therefore not directly be used in verifications with a model checker such as FDR. Fortunately, there is another method of abstraction, based on the standard $\mid\mid\mid$ interleaving operator, which shares many of its properties with lazy abstraction.

Interleaving

The methods of abstraction described above share the feature of removing the events being abstracted from the traces of the process. It is possible to take the opposite approach and insert additional events into the traces, thus obscuring the original source of the event. This is achieved by interleaving the process with Run_H which is itself continually prepared to contribute an event from H.

Definition 2.2.32 Assume that Σ is partitioned by H, L. Given a process P, the interleaving abstraction of P is defined

$$\mathcal{I}_H(P) \quad \widehat{=} \quad P \mid\mid\mid Run_H \ .$$

A trace of the process abstraction $\mathcal{I}_H(P)$ contains a trace of P as a subtrace, together with H events inserted in arbitrary places. If $\mathcal{I}_H(P)$ proceeds with an event from H, then the source is not determined: if the event originated from P then there will be a change of state in this process, but if the event was contributed by Run_H then process P will remain unchanged.

Example 2.2.33 Consider the processes

$$\begin{array}{rcl} R & \widehat{=} & (a \to x \to Stop) \ , \\ S & \widehat{=} & (a \to x \to Stop) \, \Box \, (x \to Stop) \ . \end{array}$$

For process R, interleaving $\{x\}$ will not affect interface $\{a\}$ since any x events contributed by Run_H cannot prohibit the single a event from occurring

$$\begin{array}{rcl} \mathcal{I}_H(R) & = & x \to \mathcal{I}_H(R) \\ & & \Box \ a \to Run_{\{x\}} \ . \end{array}$$

In process S, the occurrence of event x before a will remove the option of engaging this event. This difference results in non-determinism under interleaving: when an x occurs in a trace of $\mathcal{I}_H(S)$, it is not determined whether or not it is still possible to engage a, because the source of x cannot be known.

$$\begin{array}{rcl} \mathcal{I}_H(S) & = & x \to (Run_{\{x\}} \sqcap \mathcal{I}_H(S)) \\ & & \Box \ a \to Run_{\{x\}} \ . \end{array}$$

Process $\mathcal{I}_H(R)$ is a refinement of $\mathcal{I}_H(S)$ and therefore $S \leq_H^{\mathcal{I}} R$, but the reverse does not hold. (End of Example)

Interleaving abstraction is defined in terms of the standard interleaving operator $\mid\mid\mid$ (formally given the Appendix, following page 152), and so we can derive basic properties of this abstraction directly from the operator's definition. Two properties are immediate. Firstly, given a failure (t, X) of a process P, we can insert H events into trace t to produce a sequence s which is a trace of the abstraction $\mathcal{I}_H(P)$. Furthermore, the set $X \cap L$ will be refused after s, so we can deduce that $(s, X \cap L) \in \textsc{Fails}(\mathcal{I}_H(P))$.

$$
\begin{array}{c}
(t, X) \in \textsc{Fails}(P) \wedge u \in H^* \wedge s \in t \mid\mid\mid u \\
\Rightarrow \\
(s, X \cap L) \in \textsc{Fails}(\mathcal{I}_H(P)) \ .
\end{array}
\tag{2.1}
$$

Secondly, given a failure (s, X) of the interleaving abstraction $\mathcal{I}_H(P)$, we can deduce that there must a failure (t, X) of process P that has given rise to the failure of the abstract process. The traces t and s agree in their events apart from some additional H events in trace s.

$$
\begin{array}{c}
(s, X) \in \textsc{Fails}(\mathcal{I}_H(P)) \\
\Rightarrow \\
\exists\, t \in \textsc{Traces}(P), u \in H^* \bullet s \in t \mid\mid\mid u \wedge (t, X) \in \textsc{Fails}(P) \ .
\end{array}
\tag{2.2}
$$

Some additional algebraic properties which the interleaving abstraction enjoys are summarized below.

Proposition 2.2.34 Abstracting the empty set by interleaving has no effect

$$H = \emptyset \quad \Rightarrow \quad \mathcal{I}_H(P) = P \ ,$$

the order of interleaving is irrelevant

$$\mathcal{I}_{H2}(\mathcal{I}_{H1}(P)) \quad = \quad \mathcal{I}_{H1 \cup H2}(P) \ ,$$

and interleaving distributes over internal choice

$$\mathcal{I}_H(P \sqcap Q) \quad = \quad \mathcal{I}_H(P) \sqcap \mathcal{I}_H(Q)$$

and therefore preserves the refinement order $\sqsubseteq$.

Applying interleaving to the whole alphabet of a process has the effect of swamping its behaviour; the resulting process can never refuse to engage H.

Proposition 2.2.35 If process P communicates within H only, then

$$\mathcal{I}_H(P) \quad = \quad Run_H \ .$$

The interleaving abstraction partially distributes over parallel composition.

Proposition 2.2.36 Assume that Σ is partitioned by H, L, and that P communicates in A, and Q communicates in B. Then

$$\mathcal{I}_H(P) \, [A \cup H \| B \cup H] \, \mathcal{I}_H(Q) \quad \sqsubseteq \quad \mathcal{I}_H(P \, [A \| B] \, Q) \ ,$$

and if additionally $A \cap B \cap H = \emptyset$, then

$$\mathcal{I}_H(P) \, [A \cup H \| B \cup H] \, \mathcal{I}_H(Q) \quad = \quad \mathcal{I}_H(P \, [A \| B] \, Q) \ .$$

A corresponding result holds for the shared parallel operator $|[S]|$.

Proposition 2.2.37 For any processes P and Q,

$$\mathcal{I}_H(P) \; |[S]| \; \mathcal{I}_H(Q) \quad \sqsubseteq \quad \mathcal{I}_H(P \; |[S]| \; Q) \ ,$$

and if S is a set of events such that $S \cap H = \emptyset$, then

$$\mathcal{I}_H(P) \; |[S]| \; \mathcal{I}_H(Q) \quad = \quad \mathcal{I}_H(P \; |[S]| \; Q) \ .$$

It is an immediate consequence that the interleaving abstraction distributes over interleaving.

Corollary 2.2.38 For any processes P and Q,

$$\mathcal{I}_H(P) \; ||| \; \mathcal{I}_H(Q) \quad = \quad \mathcal{I}_H(P \; ||| \; Q) \ .$$

Any concurrent user process U_H which does not synchronize with P leaves the abstraction unchanged.

Proposition 2.2.39 If user process U_H communicates within H only, then

$$\mathcal{I}_H(P \; ||| \; U_H) \quad = \quad \mathcal{I}_H(P) \ .$$

Proof From Corollary 2.2.38, and Propositions 2.2.35 and 2.2.34. $\square$

A user process U_H synchronizing with P in H makes the abstract process more deterministic.

Proposition 2.2.40 If U_H is a user process communicating within H only, then

$$\mathcal{I}_H(P) \quad \sqsubseteq \quad \mathcal{I}_H(P \;|[H]|\; U_H) \;.$$

Proof

$$\begin{array}{ll} & \mathcal{I}_H(P) \\ = & \qquad\qquad\qquad\qquad\qquad [\text{ Definition of } |[H]| \text{ and } Run_H \;] \\ & \mathcal{I}_H(P) \;|[H]|\; Run_H \\ = & \qquad\qquad\qquad\qquad\qquad [\text{ Proposition 2.2.35 }] \\ & \mathcal{I}_H(P) \;|[H]|\; \mathcal{I}_H(U_H) \\ \sqsubseteq & \qquad\qquad\qquad\qquad\qquad [\text{ Proposition 2.2.37 }] \\ & \mathcal{I}_H(P \;|[H]|\; U_H) \;. \end{array}$$

$\square$

A special case is given by the user process $U_H \mathrel{\widehat{=}} Chaos_H$. Consequently the refusals in H are transparent under interleaving.

Proposition 2.2.41 The refusals in H in process P are transparent under interleaving

$$\mathcal{I}_H(P) \quad = \quad \mathcal{I}_H(P \;|[H]|\; Chaos_H) \;.$$

Proof We have $\mathcal{I}_H(P) \sqsubseteq \mathcal{I}_H(P \;|[H]|\; Chaos_H)$ by Proposition 2.2.40.

Refinement in the other direction also holds, because $P \;|[H]|\; Chaos_H \sqsubseteq P$ in general and interleaving preserves the refinement order. $\square$

Proposition 2.2.42 If $s \frown \langle a \rangle \in \textsc{Traces}(P)$ and $a \notin H$, then

$$\mathcal{I}_H(P \;\underline{\text{after}}\; s \frown \langle a \rangle) \quad = \quad (\mathcal{I}_H(P \;\underline{\text{after}}\; s)) \;\underline{\text{after}}\; \langle a \rangle \;.$$

2.3 Relating the Abstractions

In this section, we relate the concrete methods of abstraction introduced earlier and comment on their suitability for the modelling of interaction. It appears that in practice, either lazy abstraction, interleaving, or a combination of both are often sufficiently powerful abstractions. This warrants a closer look at the relationship between the two abstractions.

Interleaving versus Lazy Abstraction

On initial consideration, interleaving and lazy abstraction appear to be considerably different forms of abstraction. Interleaving in effect inserts additional events into a process trace. In contrast, lazy abstraction removes events from a process trace in a way similar to the hiding (concealment) of events.

Both forms of abstraction, however, are closely related. One property they share is that neither can introduce divergence into a non-diverging process. Furthermore, it is the case that an application of both abstractions in sequence almost always yields the same result as an application of the second abstraction alone. For instance, an application of lazy abstraction has the same effect on process P as first interleaving and then lazily abstracting.

Theorem 2.3.1 If P is a process, then

$$\mathcal{L}_H(\mathcal{I}_H(P)) \quad = \quad \mathcal{L}_H(P)$$

and thus $\mathcal{I}_H(P) \equiv_H^{\mathcal{L}} P$.

Proof

$$\begin{array}{ll}
 & \mathcal{L}_H(\mathcal{I}_H(P)) \\
= & \hfill [\ \text{Definition 2.2.32}\ ] \\
 & \mathcal{L}_H(P \mid\mid\mid Run_H) \\
= & \hfill [\ \text{Corollary 2.2.25}\ ] \\
 & \mathcal{L}_H(P) \mid\mid\mid \mathcal{L}_H(Run_H) \\
= & \hfill [\ \text{Proposition 2.2.21}\ ] \\
 & \mathcal{L}_H(P) \mid\mid\mid Stop \\
= & \hfill [\ \text{Property of } \mid\mid\mid \text{ interleaving}\ ] \\
 & \mathcal{L}_H(P)\ .
\end{array}$$

□

Lazy abstraction is a stronger form of abstraction than interleaving, in the sense that lazy abstraction removes a greater amount of process behaviour. This is made precise by observing that interleaving always makes at least as fine distinctions between abstract processes as lazy abstraction. Thus, two processes which can be distinguished under lazy abstraction can also be distinguished under interleaving. This result follows immediately from Theorem 2.3.1.

Corollary 2.3.2 For any processes P and Q,

$$P \equiv^{\mathcal{I}}_H Q \quad \Rightarrow \quad P \equiv^{\mathcal{P}}_H Q \ .$$

The converse is not true, as Example 2.3.3 illustrates.

Example 2.3.3 Let $\Sigma \mathrel{\widehat{=}} \{a, x\}$ and $H \mathrel{\widehat{=}} \{x\}$, and consider the processes

$$\begin{array}{lcl} P & \mathrel{\widehat{=}} & (\mathit{Stop} \sqcap a \to P) \,\Box\, (x \to \mathit{Stop}) \\ Q & \mathrel{\widehat{=}} & (a \to Q) \,\Box\, (x \to \mathit{Stop}) \ . \end{array}$$

Both processes are equivalent under lazy abstraction since $\mathcal{L}_H(P) = \mathcal{L}_H(Q)$ but they can be distinguished under interleaving:

$$(\langle\rangle, \{a\}) \in \mathrm{F\small{AILS}}(\mathcal{I}_H(P)) \quad \wedge \quad (\langle\rangle, \{a\}) \notin \mathrm{F\small{AILS}}(\mathcal{I}_H(Q)) \ ,$$

because the refusal of a cannot be observed unless at least one x event appears in the trace of Q. (End of Example)

In Theorem 2.3.1 we have seen that $\mathcal{I}_H(P)$ and P are equivalent under lazy abstraction. Applying the abstractions in the opposite order, it turns out that $\mathcal{L}_H(P)$ and P are not always equivalent under interleaving of H, although we have the following partial result: the double abstraction $\mathcal{I}_H(\mathcal{L}_H(P))$ can serve as a specification for the interleaving abstraction $\mathcal{I}_H(P)$.

Theorem 2.3.4 If P is a process, then

$$\mathcal{I}_H(\mathcal{L}_H(P)) \ \sqsubseteq \ \mathcal{I}_H(P) \ .$$

Proof

$$\begin{array}{ll} & \mathcal{I}_H(\mathcal{L}_H(P)) \\ = & \qquad\qquad [\ \text{Definition 2.2.32}\ ] \\ & \mathcal{L}_H(P) \ ||| \ \mathit{Run}_H \\ = & \qquad\qquad [\ \text{Property of interleaving } |||\ ] \\ & \mathcal{L}_H(P) \ ||| \ \mathit{Chaos}_H \ ||| \ \mathit{Run}_H \\ \sqsubseteq & \qquad\qquad [\ \text{Proposition 2.2.29}\ ] \\ & \mathcal{L}_H(P) \ ||| \ \mathcal{L}_L(P) \ ||| \ \mathit{Run}_H \\ \sqsubseteq & \qquad\qquad [\ \text{Proposition 2.2.30}\ ] \\ & P \ ||| \ \mathit{Run}_H \\ = & \qquad\qquad [\ \text{Definition 2.2.32}\ ] \\ & \mathcal{I}_H(P) \ . \end{array}$$

$\square$

It would be wrong to replace $\sqsubseteq$ by equality $=$ in Theorem 2.3.4, since the double abstraction $\mathcal{I}_H(\mathcal{L}_H(P))$ does not always refine the single interleaving abstraction. An example of this case is given by the process Q of Example 2.3.3. The abstract process is $Q\,@\{a\} = \mathit{Stop} \sqcap (a \rightarrow Q\,@\{a\})$, and so

$$(\langle\rangle, \{a\}) \in \mathcal{I}_H(Q\,@\{a\}) \quad \wedge \quad (\langle\rangle, \{a\}) \notin \mathcal{I}_H(Q)\ .$$

$(\langle\rangle, \{a\})$ is not a failure of $\mathcal{I}_H(Q)$ because at least one x must appear in a trace before event a can be refused. Therefore $\mathcal{I}_H(Q) \not\sqsubseteq \mathcal{I}_H(Q\,@\{a\})$.

Summary

Besides the traditional form of abstraction used in program development – the hiding (concealment) of actions described in Section 2.2 – there are several other functions on processes which permit abstraction of computational behaviour. Although there are no generally agreed criteria on which such functions are adequate abstractions, the operators described in this chapter share a number of algebraic properties. Precisely which of the properties of Section 2.1 are satisfied depends on the computational model in which analysis takes place; in this thesis the failures-divergences model of CSP will serve as the main model of reference.

An important property of all process abstractions is that they, in general, can introduce non-determinism. This non-determinism may either be the consequence of unbounded behaviour (leading to divergence), or the consequence of choices among events being abstracted. Hiding and strict projection are sensitive to the possibility of divergence, and are thus appropriate for situations where it is important to detect when one set of events may continually pre-empt another. The other methods of abstraction are to be preferred where the potential of infinite preemption is of no concern.

The purpose of applying an abstraction function is to disregard computational detail of a process, with the aim of either managing the complexity, or analysing the effect of part of the process's alphabet on its overall behaviour. In particular, a process abstraction may be interpreted as describing how an agent, interacting with a part of the process's interface, perceives this interaction. For instance, an agent able to observe and control events a, b but not x in

$$P \quad = \quad a \rightarrow (b \rightarrow Stop \ \Box \ x \rightarrow a \rightarrow Stop)$$

cannot guarantee the second occurrence of event a because it relies on the prior cooperation of another agent engaging in x. This lack of control can be expected to result in non-determinism under abstraction.

2.4 A Condition for Non-interaction

In this section, we will consider the application of abstraction functions to express a condition for non-interaction. This condition, which we call *robustness*, gives a general characterisation of fault-tolerance or robustness of a process Q with respect to a number of "fault events". The aim of fault modelling is the verification that the occurrence of fault events cannot disrupt the standard expected behaviour of the process. In the terminology developed previously in this chapter, we aim to specify that events H have only limited scope of interaction with what is regarded as correct process behaviour.

Before a verification of robustness can be carried out, it is necessary to define a specification process P which captures the standard behaviour of the system, i.e. the expected sequence of events under the absence of H events. This specification process therefore engages the standard events L only and cannot communicate in H. This process P serves as a specification for Q which *can* communicate H events (whose effects should be limited). However, a refinement check $P \sqsubseteq Q$ would not even be satisfied on the level of trace refinement, because process Q can engage in H events which is impossible for P. It is necessary to abstract the additional H events in the implementation Q, and verify for instance $P \sqsubseteq \mathcal{L}_H(Q)$. The following result shows that there are several ways in which this refinement can be equivalently expressed.

Lemma 2.4.1 Let Σ be partitioned by H and L. If process P does not communicate in H, then the following conditions are equivalent.

$$P \quad \sqsubseteq \quad \mathcal{L}_H(Q) \tag{2.3}$$

$$\mathcal{I}_H(P) \quad \sqsubseteq \quad \mathcal{I}_H(Q) \tag{2.4}$$

$$P \ ||| \ Chaos_H \quad \sqsubseteq \quad Q \ . \tag{2.5}$$

Proof Firstly, we show that (2.3) implies (2.4).

$$
\begin{array}{ll}
 & \mathcal{I}_H(P) \\
= & \qquad\qquad\qquad\qquad [\text{ Definition 2.2.32 }] \\
 & P \mid\mid\mid \mathit{Run}_H \\
\sqsubseteq & \qquad\qquad\qquad\qquad [\text{ by Equation } (2.3) \text{ }] \\
 & \mathcal{L}_H(Q) \mid\mid\mid \mathit{Run}_H \\
= & \qquad\qquad\qquad\qquad [\text{ Definition 2.2.32 }] \\
 & \mathcal{I}_H(\mathcal{L}_H(Q)) \\
\sqsubseteq & \qquad\qquad\qquad\qquad [\text{ Theorem 2.3.4 }] \\
 & \mathcal{I}_H(Q) \ .
\end{array}
$$

We now show that (2.4) implies (2.5).

$$
\begin{array}{ll}
 & P \mid\mid\mid \mathit{Chaos}_H \\
= & \qquad\qquad\qquad\qquad [\text{ Proposition 2.2.22 }] \\
 & \mathcal{L}_H(P) \mid\mid\mid \mathit{Chaos}_H \\
= & \qquad\qquad\qquad\qquad [\text{ Proposition 2.2.26 }] \\
 & \mathcal{L}_H(P \mid\mid\mid \mathit{Run}_H) \mid\mid\mid \mathit{Chaos}_H \\
\sqsubseteq & \qquad\qquad\qquad\qquad [\text{ by Equation } (2.4) \text{ }] \\
 & \mathcal{L}_H(Q \mid\mid\mid \mathit{Run}_H) \mid\mid\mid \mathit{Chaos}_H \\
= & \qquad\qquad\qquad\qquad [\text{ Corollary 2.2.25 }] \\
 & \mathcal{L}_H(Q) \mid\mid\mid \mathcal{L}_H(\mathit{Run}_H) \mid\mid\mid \mathit{Chaos}_H \\
= & \qquad\qquad\qquad\qquad [\text{ Proposition 2.2.21 }] \\
 & \mathcal{L}_H(Q) \mid\mid\mid \mathit{Stop} \mid\mid\mid \mathit{Chaos}_H \\
= & \qquad\qquad\qquad\qquad [\text{ Property of interleaving } \mid\mid\mid \text{ }] \\
 & \mathcal{L}_H(Q) \mid\mid\mid \mathit{Chaos}_H \\
\sqsubseteq & \qquad\qquad\qquad\qquad [\text{ Proposition 2.2.29 }] \\
 & \mathcal{L}_H(Q) \mid\mid\mid \mathcal{L}_L(Q) \\
\sqsubseteq & \qquad\qquad\qquad\qquad [\text{ Proposition 2.2.30 }] \\
 & Q \ .
\end{array}
$$

Finally we show that (2.5) implies (2.3).

$$
\begin{array}{ll}
 & P \\
= & \qquad\qquad\qquad\qquad [\text{ Proposition 2.2.22 }] \\
 & \mathcal{L}_H(P) \\
= & \qquad\qquad\qquad\qquad [\text{ Proposition 2.2.26 }] \\
 & \mathcal{L}_H(P \mid\mid\mid \mathit{Chaos}_H) \\
\sqsubseteq & \qquad\qquad\qquad\qquad [\text{ by Equation } (2.5) \text{ }] \\
 & \mathcal{L}_H(Q) \ .
\end{array}
$$

This completes the proof of Lemma 2.4.1. $\Box$

We now proceed with the definition of robustness, which captures the notion that the occurrence of H events do not interfere with the standard process behaviour. The process abstraction $\mathcal{S}_H(Q)$, in which H events are stopped, describes this standard behaviour. It serves as specification for $\mathcal{L}_H(Q)$ in which H events *can* occur (but do not appear in its traces).

Definition 2.4.2 Let Σ be partitioned by H and L. Interface L is *robust* in process Q if

$$\mathcal{S}_H(Q) \;\; \sqsubseteq \;\; \mathcal{L}_H(Q) \; .$$

Robustness is a weak condition for non-interference, because it does not require the complete absence of interaction from H. In fact, under robustness an individual H step can make the L interface less deterministic or more deterministic, provided the effect is within the original bounds of behaviour given by $\mathcal{S}_H(Q)$, as illustrated by the following example.

Example 2.4.3 In process Q below, the interface $\{a\}$ allows the repeated occurrence of event a, unless the internal choice is made in favour of $Stop$ thus refusing this event. However, once an x event occurs, the refusal of a is no longer possible. A subsequent y event takes the process to its original state, so that again a may be accepted or refused non-deterministically.

$$\begin{array}{rcl} Q & \widehat{=} & (Stop \sqcap a \rightarrow Q) \,\Box\, (x \rightarrow Q') \\ Q' & \widehat{=} & (a \rightarrow Q) \,\Box\, (y \rightarrow Q) \; . \end{array}$$

Interface $\{a\}$ is robust in Q. Note that the occurrence of a x makes interface $\{a\}$ more deterministic, since event a can no longer be refused. However, the occurrence of a y makes $\{a\}$ less deterministic, since the internal choice can now be made in favour of $Stop$. (End of Example)

Although we have defined robustness in terms of stopping and lazy abstraction, it is equally well to express the condition using other forms of abstraction, as the next theorem illustrates.

Theorem 2.4.4 Let Σ be partitioned by H and L. Either of the following conditions are equivalent to robustness of L in process Q.

$$\mathcal{I}_H(\mathcal{S}_H(Q)) \sqsubseteq \mathcal{I}_H(Q) \tag{2.6}$$

$$\mathcal{S}_H(Q) \mid\mid\mid Chaos_H \sqsubseteq Q \ . \tag{2.7}$$

Proof This follows directly from the results of Lemma 2.4.1 since H events cannot occur in $\mathcal{S}_H(Q)$. $\square$

The robustness condition is suitable for verifying whether the behaviours of an implementation process are confined by the behaviours of a specification under blocking of a certain set of events, such as faults or exceptions. The use of stopping in Definition 2.4.2 is suitable where it is desirable to check against a specification where *none* of the fault events may occur. Less stringent fault models useful in this context are discussed by Roscoe [Ros97] thus extending the range of refinement-based fault-tolerance conditions, of which robustness is a special case. The application of robustness for the modelling of authentication and key-exchange protocols is addressed by Gardiner et al [GGJ96]. The relationship between security and safety properties of systems is investigated by Simpson [Sim96] who formulates a notion of safety-critical non-interference called *protection*. This notion provides the basis of a common modelling framework for security as well as fault-tolerance and dependability properties of systems.

Chapter 3

Security

In this chapter, we investigate and discuss conditions which aim to capture the notion of "non-interference" between computational agents. Goguen and Meseguer's original theory of non-interference [GM82] has spawned a number of similar CSP formulations, which we collectively refer to as "invariance", described in Section 3.1. Although invariance can be given an appealing characterisation in terms of a process abstration, we will argue that it is less suitable for application to non-deterministic processes. Invariant interfaces can suffer from the "refinement paradox", i.e. they can be refined to ones that are subject to information flow.

In the presence of non-determinism, it is necessary to insist that interfaces are independent not only of events occurring at other interfaces but also of all possible outcomes of internal choices made by the process. The condition of "independence" is defined in Section 3.2 and also given a characterisation in terms of process abstractions.

The conditions of invariance and independence share the property of being asymmetric: the interface of one agent must not be affected by the actions of another, but not necessarily the reverse. We relate these asymmetric conditions to the symmetric condition of separability in Section 3.3, which requires that a process in which two interfaces cannot affect each other can be equivalently expressed as a parallel composition of components with disjoint alphabets. Insisting on deterministic components results in conditions for non-interference which are preserved under refinement. It is assumed throughout this chapter that only two interfaces are under consideration, namely H (high) and L (low), which partition Σ.

3.1 Invariance

Secure processes are those in which no (unwanted) interaction between interfaces takes place. In particular, it should not be possible, by observation of one interface, to deduce anything about the past or future behaviour in another interface.

The possible observations in interface L consist of a record of those sub-traces of the process traces consisting only of L events, and the initial L events accepted or refused at the interface at any one point in the process history. Since a process could potentially be executed several times, an external observer of interface L is in a position to conduct a variety of similar experiments with the process, based on the choice of initial L events which partially determines the process history.

The condition of invariance, which requires the possible observations in interface L at any point to depend on nothing but the past sequence of L events, was independently introduced by Allen [All91] and Ryan [Rya91]. Most basically, the past record of L events must determine the set of initial events available in L in order for the interface to be invariant. In general, it is not only the initials but also the refusal sets which should be determined by past L events.

Definition 3.1.1 Interface L is trace-invariant in process P if

$$s \restriction L = t \restriction L \quad \Rightarrow \quad (P \underline{\text{ after }} s)^0 \cap L = (P \underline{\text{ after }} t)^0 \cap L \ ,$$

and failures-invariant if

$$\begin{aligned} s \restriction L = t \restriction L \quad \Rightarrow \quad & (P \underline{\text{ after }} s)^0 \cap L = (P \underline{\text{ after }} t)^0 \cap L \ \wedge \\ & \textsc{Refs}(P \underline{\text{ after }} s) \cap \mathbb{P}\, L = \textsc{Refs}(P \underline{\text{ after }} t) \cap \mathbb{P}\, L \ . \end{aligned}$$

Trace invariance may be regarded as a minimum condition that any criterion for non-interaction should satisfy, for its failure would indicate that intermediate events (outside L) can change the traces which can be communicated in interface L. Failures-invariance is a stronger condition, which additionally requires the refusal sets in L to be unaffected by past events outside L. From now on, we will refer to the failures-invariance simply as invariance unless otherwise stated, particularly in the context of non-deterministic processes.

The invariance condition has been given a characterisation in terms of process abstractions [Ros95, RW95]. Informally, a set of events L is invariant in a process P if the occurrence of events outside L in a trace of P do not change the process under interleaving.

Theorem 3.1.2 Let Σ be partitioned by H, L. Interface L is invariant in process P if, and only if, for all $s \in \text{TRACES}(P)$,

$$s \setminus H \in \text{TRACES}(P) \quad \wedge \quad P \,\underline{\text{after}}\, s \equiv_H^{\mathcal{I}} P \,\underline{\text{after}}\, s \setminus H \ .$$

Thus, if L is invariant then the subsequence $s \setminus H$ which can be observed through interface L in a process trace s is also a valid trace, and the process view after either trace is identical modulo interleaving. Since the H events potentially contained in trace s have not affected interface L (under abstraction), it is plausible to regard invariance as a characterisation of non-interference.

Like Goguen and Meseguer's original theory of non-interference, invariance as a condition for non-interference has a strong intuitive appeal. For deterministic processes, trace-invariance is sufficient for a specification of security. For non-deterministic processes, however, the situation is much more complex. We do not regard it as sufficient in this context to take refusals into account as in the failures-invariance condition. The weakness of invariance in this general context is best illustrated with a number of examples.

Examples

In this subsection, we give a series of examples of non-deterministic processes in which interface L is invariant. The greatest weakness of invariance is that it does not prevent what has come to be known as the refinement paradox [Ros95]: even though a (non-deterministic) process may satisfy invariance of L, it can have refinements which violate the condition.

Example 3.1.3 Interface $L = \{a, b\}$ is invariant in

$$P \quad = \quad (x \rightarrow P \ \Box \ y \rightarrow P)$$

$$\Box\ (a \rightarrow P \sqcap b \rightarrow P)\ ,$$

since $(P\ \underline{\text{after}}\ t)^0 \cap L = L$ and $\mathrm{R{\scriptsize EFS}}(P\ \underline{\text{after}}\ t) \cap \mathbb{P}\, L = \{\{a\},\{b\}\}$, for any trace t. However, process P admits for example the deterministic implementation $Q(0)$ where

$$\begin{array}{rcl} Q(c) & = & (x \rightarrow Q(c+1)\ \Box\ y \rightarrow Q(c-1)) \\ & & \Box\ \ \underline{\text{if}}\ c \geq 0\ \underline{\text{then}}\ a \rightarrow Q(c)\ \underline{\text{else}}\ Stop \\ & & \Box\ \ \underline{\text{if}}\ c < 0\ \underline{\text{then}}\ b \rightarrow Q(c)\ \underline{\text{else}}\ Stop\ . \end{array}$$

in which the availability of L events depends on the past occurrence of the x and y events. If a is accepted in L, i.e. $a \in (P\ \underline{\text{after}}\ t)^0$, then $t \downarrow x \geq t \downarrow y$, i.e. the number of x's which had occurred has been at least as large as the number of y's. The opposite is true when b is accepted in L. (End of Example)

Example 3.1.4 Interface $L = \{a\}$ is invariant in

$$\begin{array}{rcl} P & \widehat{=} & a \rightarrow Stop \\ & & \sqcap\ x \rightarrow (Stop \sqcap a \rightarrow Stop) \end{array}$$

because event a may be accepted and refused both before and after the x event occurs. If the initial internal choice is made in favour of x, however, then a delay of the occurrence of x blocks any further activity in interface L. This potential dependency of L becomes most obvious by considering

$$P' \ = \ x \rightarrow a \rightarrow Stop$$

which is a refinement of P. The a cannot occur as long as the x remains postponed, and once the a becomes available it is possible to deduce that x must have happened previously. (End of Example)

Example 3.1.5 Interface $L = \{a\}$ is also invariant in

$$Q \ \widehat{=} \ (a \rightarrow Q) \sqcap (x \rightarrow Q)$$

which gives a repeated internal choice between a and x. The a event may be accepted or refused at any point, irrespective of the number of x events in a trace of Q.

Again the progress of communications in interface L is delayed if the internal choice is taken in favour of x and the occurrence of this event is not immediately forthcoming. In this case the delay of activity in the complement interface will temporarily deadlock the L interface. (End of Example)

The examples above show that an invariant interface can still be dependent on internal (non-deterministic) choices taken by the process. This dependence is clearly illustrated by the lack of ability to define two mutually invariant interfaces using two separate concurrent processes with disjoint interfaces. For instance, process Q of Example 3.1.5 does not equal $Chaos_L \mid\mid\mid Chaos_H$. It is possible to strengthen the invariance condition as described in Theorem 3.1.2 to overcome this weakness, by defining a more powerful "after" operator which takes not only traces but also refusals into account. The resulting condition of *strong* invariance, and the proof that its symmetric formulation corresponds to separability (appearing later in this chapter), is discussed by Roscoe and Wulf [RW95].

An equally strong disadvantage of invariance is that it is not preserved under refinement – if interface L is invariant in process P, and $P \sqsubseteq Q$, then L need not be invariant in Q. This paradoxical situation makes it impractical to use invariance in secure system development, as the system implementations resulting from individual refinement steps cannot be relied upon to satisfy the original security properties of the specification.

The refinement paradox exists because non-deterministic choices may be resolved in a way that makes the L interface dependent on the activity in another interface. This possible dependency is the basis for flows of information which are undesired in a secure system. Since we "don't care" how non-deterministic choices are resolved, it must be assumed that this happens in the worst possible manner (from the security viewpoint), as if the demon making those choices is bound to allow the flow of information to be as large as possible. The way to avoid the refinement paradox is to ensure that there are no non-deterministic choices in the L interface which can result in dependencies and thus introduce information flow.

3.2 Independence

In the previous section, we have seen that invariance as a condition for non-interference suffers from the possibility of non-determinism in interface L. Consequently it is possible to find examples of invariant interfaces that are still dependent on the outcome of choices made in some other interface. Furthermore, invariance is not preserved under refinement, so that a valid refinement of a process with invariant interfaces may no longer guarantee invariance.

In order to find a satisfactory condition for security – the complete absence of interaction – it is necessary to find a stronger condition than invariance. The stronger condition should guarantee that, in a secure process, interface L is independent not only of the activity in other interfaces but also of the outcome of internal choices made by the process.

To formalise this restriction, we introduce a variant of the standard definition of determinism (Appendix, Section A.2), which requires that no a event in L may be simultaneously accepted and refused after any process trace.

Definition 3.2.1 Process P is L-deterministic if, and only if, it is free of divergence, and

$$(s, \{a\}) \in \textsc{Fails}(P) \quad \Leftrightarrow \quad (P \, \underline{\mathsf{after}} \, s)^0 \cap \{a\} = \emptyset$$

for all $a \in L$ and traces s of P.

Strictly speaking the above condition does not imply that an external observer of L would perceive this interface to be deterministic. This is because trace s may contain events accepted at other interfaces, beyond the control of L. To specify that the L interface is deterministic irrespective of activity in other interfaces it is necessary to prevent an event $a \in L$ from being both accepted and refused after traces with identical L behaviour.

Definition 3.2.2 Interface L is independent in process P if, and only if, P does not diverge, and

$$(s, \{a\}) \in \textsc{Fails}(P) \quad \Leftrightarrow \quad (P \, \underline{\mathsf{after}} \, t)^0 \cap \{a\} = \emptyset$$

for all $a \in L$ and traces s, t of P with $s \upharpoonright L = t \upharpoonright L$.

If Σ is partitioned by H and L, independence of interface L from all other possible communications in process P will be denoted

$$H \not\longrightarrow L :: P \ .$$

From the above definition of independence, it is clear under which circumstances the interface L fails to be independent: the dependence becomes evident if it is possible to exhibit two witness traces s and t with identical low sub-traces after which non-deterministic behaviour can be observed in L. That is, interface L cannot be independent if there is an event $a \in L$ and traces s, t such that

$$s \upharpoonright L = t \upharpoonright L \ \wedge$$
$$s \frown \langle a \rangle \in \text{TRACES}(P) \ \wedge \ (t, \{a\}) \in \text{FAILS}(P) \ . \tag{3.1}$$

For deterministic processes, independence and invariance are equivalent conditions. In the context of non-deterministic processes, however, independence is a much stronger condition by requiring the process to present a deterministic interface to an external observer. Although we have defined independence in terms reminiscent of the standard notion of determinism in CSP, it is possible to find alternative characterisations. The most intriguing perhaps is that independence holds under determinism of suitable abstract processes.

Theorem 3.2.3 Let Σ be partitioned by H, L. The following conditions are equivalent.

1. Interface L is independent in process P;
2. Interface L is trace-invariant, and P is L-deterministic;
3. The abstract process $\mathcal{L}_H(P)$ is deterministic;
4. The abstract process $\mathcal{I}_H(P)$ is deterministic.

It is remarkable that lazy abstraction and interleaving identify precisely the same class of processes to be deterministic under abstraction, despite having very different effects on the processes to which these abstractions are applied. For a proof of independence, we can thus use either lazy abstraction or interleaving and exploit their respective algebraic properties.[1]

Independence can also be characterised in terms of other process abstractions. Determinism of the process under strict projection is equivalent to independence, provided that the process remains deterministic under hiding.

[1] Interleaving is the abstraction of choice in automatic verifications of independence. The model checker FDR implements ||| interleaving as one of the standard operators, and allows direct verification of whether or not a process is deterministic.

Proposition 3.2.4 $H \not\longleftrightarrow L :: P$ and the abstract process $\mathcal{H}_H(P)$ is deterministic if, and only if, the abstract process $\mathcal{P}^*_H(P)$ is deterministic.

Proof A proof of this result appears in [RWW96]. (i) Suppose $\mathcal{P}^*_H(P)$ is deterministic. Then process $\mathcal{H}_H(P)$ must equal $\mathcal{P}^*_H(P)$ by elementary properties of CSP refinement, and thus also be deterministic. Furthermore, if $H \not\longleftrightarrow L :: P$ were not to hold, then by (3.1) there are traces s and t of P and an event $a \in L$ such that

$$
\begin{array}{ll}
 & s \upharpoonright L = t \upharpoonright L \ \wedge \\
 & s \frown \langle a \rangle \in \textsc{Traces}(P) \ \wedge \ (t, \{a\}) \in \textsc{Fails}(P) \\
\Rightarrow & \qquad\qquad\qquad\qquad [\ \text{Proposition 2.2.9}\ ] \\
 & s \upharpoonright L = t \upharpoonright L \ \wedge \\
 & (s \upharpoonright L) \frown \langle a \rangle \in \textsc{Traces}(\mathcal{P}^*_H(P)) \ \wedge \\
 & (t \upharpoonright L, \{a\}) \in \textsc{Fails}(\mathcal{P}^*_H(P)) \\
\Rightarrow & \qquad\qquad\qquad\qquad [\ \text{Definition of determinism}\ ] \\
 & \mathcal{P}^*_H(P) \text{ non-deterministic ,}
\end{array}
$$

contradicting the assumption that $\mathcal{P}^*_H(P)$ is deterministic. Therefore interface L must be independent in P.

(ii) The reverse implication can easily be proved with the help of (3.1). □

If interface L is independent in process P, then it is entirely unaffected by concurrent processes communicating in another interface. The following result shows that, given a (non-diverging) user process U_H, the abstract process $\mathcal{I}_H(P \,|[H]|\, U_H)$ is deterministic and guaranteed to be equal to $\mathcal{I}_H(P)$. The corresponding result holds for lazy abstraction.

Proposition 3.2.5 If $H \not\longleftrightarrow L :: P$ and U_H is any user process communicating in H only, then

$$
\begin{array}{rcl}
\mathcal{I}_H(P) & = & \mathcal{I}_H(P \,|[H]|\, U_H) \ \wedge \\
\mathcal{L}_H(P) & = & \mathcal{L}_H(P \,|[H]|\, U_H) \ .
\end{array}
$$

Proof We exploit the assumption that the abstract processes $\mathcal{I}_H(P)$ and $\mathcal{L}_H(P)$ are deterministic, and as such do not have proper refinements. For interleaving, the result follows from Proposition 2.2.40 and the assumption that $\mathcal{I}_H(P)$ is deterministic. For lazy abstraction, the result follows from Proposition 2.2.27 and the assumption that $\mathcal{L}_H(P)$ is deterministic. □

Even though the independence condition requires the abstract process to be deterministic, the H interface being abstracted may still retain non-determinism.

Example 3.2.6 Consider the process P below with interfaces $L = \{a, b, c\}$ and $H = \{x, y\}$.

$$
\begin{array}{rcl}
P & \widehat{=} & a \to (Q \;\Box\; x \to Q) \\
 & & \Box\; b \to Q \\
Q & \widehat{=} & c \to (P \;\Box\; (x \to P \sqcap y \to P)) \\
 & & \Box\; b \to P \;.
\end{array}
$$

Through interface L process P alternately offers the choice between events a and b, and between events b and c. After each a, event x is temporarily available in interface H, and after each c, there is an internal choice between the two high events one of which will be available in H. Interface L is independent from H, and entirely unaffected by the outcome of internal choices in the other interface. (End of Example)

3.3 Separation and Composition

In this section, we consider conditions for secure systems where security is achieved through separation of interfaces. The aim of this approach is to model a secure system as an environment in which individual interfaces are indistinguishable from one given by physically separate or isolated components. Since the components are thought of as isolated from each other, there cannot be any means of interference between them apart from that along known communication channels.

The idea of separability has been introduced by Rushby [Rus81] as a verification technique for kernelised systems, in which security-critical functions have been localised in a single place – the kernel. In particular, Rushby has applied the technique to verify the kernel of a secure operating system. The aim of the proof is to establish that the single, shared system with multiple interfaces cannot be distinguished from a shared environment with physically separate components for each interface. Thus, the system appears dedicated to each user through its interface operating free of interference from any other component. Security is achieved when each user's view of the shared system exactly coincides with that provided by a truly distributed environment.

In the context of communicating processes, the separability condition has been formalised by Burnham [Bur89]: a process is separable if it is equivalent to a parallel composition of sub-processes with disjoint interfaces. This condition is succinctly expressed in CSP (Definition 3.3.1 below) and provides a security criterion which has great intuitive appeal: processes with disjoint interfaces do not synchronize on events when combined in parallel, leaving no scope for direct interference between them. Separability of communicating processes is further investigated by Jacobs [Jac90] who points out a slight flaw in Rushby's original presentation of the technique.

The drawback of separability is that it is inherently a symmetric condition: removing all sources of direct interference between isolated components means that there is no information flow in either direction. Security, however, is generally regarded as an asymmetric condition which prohibits information to flow in one direction, but allows it to flow in the other. In this section we consider both the symmetric condition and its asymmetric variant, called semi-separability, which is somewhat harder to formulate. One conclusion is that (semi-)separability alone should not be regarded as proving the absence of information flow, as long as there is the possibility of non-determinism. The condition is strengthened to remove this possibility, providing further evidence that independence is the most suitable condition of security for general systems.

Separability

A process is said to be separable if it can be equivalently expressed as a parallel composition of disjoint components. As the interfaces of the two component processes are disjoint, the composition can be expressed using $|||$ interleaving.

Definition 3.3.1 A process P is said to be *separable* if there exist processes Q_H communicating in interface H and Q_L communicating in interface L such that

$$P \quad = \quad Q_H \; ||| \; Q_L \; ,$$

and *strongly separable* if both components Q_H and Q_L are deterministic.

The separability condition does not constrain the behaviour of the two component processes, apart from requiring that they capture the complete range of behaviour possible at their respective interfaces, and do not communicate outside of them. Thus it appears that many definitions for Q_H and Q_L are possible. This is, however, not the case. The following result shows that the individual components which we can expect a separable process to have are obtained simply by lazily abstracting the complement interface.

Proposition 3.3.2 If process P is separable, then $\mathcal{L}_H(P)$ describes the behaviour in interface L, and $\mathcal{L}_L(P)$ describes the behaviour in interface H.

Proof Assuming that P is separable as $Q_L \mid\mid\mid Q_H$, we show that $\mathcal{L}_H(P) = Q_L$.

$$
\begin{array}{ll}
 & \mathcal{L}_H(P) \\
= & \qquad\qquad\qquad [\ P \text{ separable wrt } L \text{ and } H\ ] \\
 & \mathcal{L}_H(Q_L \mid\mid\mid Q_H) \\
= & \qquad\qquad\qquad [\ \text{Corollary 2.2.25}\ ] \\
 & \mathcal{L}_H(Q_L) \mid\mid\mid \mathcal{L}_H(Q_H) \\
= & \qquad\qquad\qquad [\ Q_L \text{ communicates within } L\ ] \\
 & Q_L \mid\mid\mid \mathcal{L}_H(Q_H) \\
= & \qquad\qquad\qquad [\ Q_H \text{ communicates within } H\ ] \\
 & Q_L \mid\mid\mid \mathit{Stop} \\
= & \qquad\qquad\qquad [\ \text{Property of } \mid\mid\mid \text{ and } \mathit{Stop}\ ] \\
 & Q_L\ .
\end{array}
$$

The case for interface H is analogous. □

Proposition 2.2.30 in Chapter 2 shows that $\mathcal{L}_H(P) \mid\mid\mid \mathcal{L}_L(P) \sqsubseteq P$ in general. The separability condition requires that the reverse refinement also holds, thus giving a semantically equivalent way of defining the process. Separability provides a very attractive criterion for the specification of non-interference between two interfaces: since the possible behaviours in either interface can be described as a separate process, and interleaving is a form of composition that does not involve synchronization of events, it appears that no information can flow from one interface to the other.

Despite its intuitive appeal, separability cannot directly be employed in security specifications. The two component processes into which a separable process can be split may both retain a high degree of non-determinism, which could be the source of insecurities. As an example of a separable process specification,

$$Chaos_{H \cup L} \quad \widehat{=} \quad Stop \sqcap (\Box_{e:H \cup L}\, e \rightarrow Chaos_{H \cup L})$$

is the most non-deterministic (but divergence-free) process which communicates in interfaces H and L. It is separable because it semantically equals $Chaos_H \;|||\; Chaos_L$, describing both interfaces separately. Unfortunately it is also equal to process

$$Chaos_{H \cup L} \sqcap Leak \ ,$$

for any non-diverging process $Leak$, however insecure. Consequently the system may, on the outcome of some internal choice, start leaking sensitive information to interface L, even though the original specification was separable. To avoid such insecurities, it is necessary to insist on deterministic interfaces, which is what *strong* separability achieves.

Furthermore, separability is a symmetric condition, with analogous requirements for both interfaces involved. Non-interference and security, however, are generally regarded as asymmetric properties – information can validly flow from L to H, but the reverse should be disallowed in a secure system. It is therefore desirable to develop an asymmetric variation of separability which retains its intuitive appeal, but can be directly applied in security modelling. Before we approach the formal definition of semi-separability, however, we introduce a class of processes which are able to communicate in some interface without affecting other processes which also communicate in that interface.

Monitor Processes

In this subsection, we introduce and discuss a class of processes which, by definition, are intended to communicate within at least two interfaces. These monitor processes are able to observe the activity in one interface, without affecting or restricting the possible behaviours, while also engaging in events in another. Without loss of generality, it is assumed in the following that monitor processes communicate in H and L, where L is the interface to be monitored. It is crucial to ensure that any monitor process does not restrict the possible behaviours in L being observed, since otherwise other processes which also communicate in the same interface will be affected by the monitor.

Definition 3.3.3 A *monitor* of interface L is a process M for which

$$\mathcal{L}_H(M) \quad = \quad Run_L \ .$$

According to this definition, a process monitoring the activity in interface L is one which – apart from possible H actions – behaves like process Run_L. Thus, the monitor process is characterised by the property that it may engage in any L event at any time, and none of these events is ever refused. It is essential to ensure that this lack of refusals is guaranteed; otherwise the monitoring process could potentially restrict the behaviour available through the L interface. This kind of restriction could not only make the low agent aware that monitoring is taking place, but also allow deductions about the high agent, including the implementation of the monitoring.

The above definition of a monitor process is expressed in terms of lazy abstraction. It is equally possible to apply interleaving in order to abstract the interspersed H events. It is straightforward to show that the monitor under interleaving can never refuse any actions at all.

Proposition 3.3.4 Process M is a monitor of interface L if, and only if, $\mathcal{I}_H(R)$ behaves like $Run_{L \cup H}$.

Proof (i) We first show that $\mathcal{I}_H(M) = Run_{L \cup H}$ if process M is a monitor.

$$\begin{array}{ll}
 & Run_{L \cup H} \\
= & \qquad\qquad\qquad [\text{ Property of } ||| \text{ and } Run \text{ }] \\
 & Run_L \ ||| \ Run_H \\
= & \qquad\qquad\qquad [\text{ } M \text{ is a monitor; Definition 3.3.3 }] \\
 & \mathcal{L}_H(M) \ ||| \ Run_H \\
\sqsubseteq & \qquad\qquad\qquad [\text{ Theorem 2.3.4 }] \\
 & M \ ||| \ Run_H \\
= & \qquad\qquad\qquad [\text{ Definition 2.2.32 }] \\
 & \mathcal{I}_H(M) \ .
\end{array}$$

Deterministic processes have no proper refinements. Since $Run_{L \cup H}$ is deterministic it must be equivalent to $\mathcal{I}_H(R)$.

(ii) We now show that M is a monitor of L if $\mathcal{I}_H(M) = Run_{L \cup H}$.

$$
\begin{array}{ll}
 & \mathcal{L}_H(M) \\
= & \hfill [\text{ Theorem 2.3.1 }] \\
 & \mathcal{L}_H(\mathcal{I}_H(M)) \\
= & \hfill [\text{ Assumption about } M \text{ }] \\
 & \mathcal{L}_H(Run_{L \cup H}) \\
= & \hfill [\text{ Property of } ||| \text{ and } Run \text{ }] \\
 & \mathcal{L}_H(Run_L \; ||| \; Run_H) \\
= & \hfill [\text{ Corollary 2.2.25 }] \\
 & \mathcal{L}_H(Run_L) \; ||| \; \mathcal{L}_H(Run_H) \\
= & \hfill [\text{ Proposition 2.2.21 }] \\
 & \mathcal{L}_H(Run_L) \\
= & \hfill [\text{ Proposition 2.2.22 }] \\
 & Run_L \; .
\end{array}
$$

This completes the proof of Proposition 3.3.4. □

Example 3.3.5 Process M below is a monitor of interface L which is also able to communicate in interface $H = \{x\}$.

$$
\begin{array}{lll}
M & \widehat{=} & (\Box_{e_1:L} \, e_1 \to \Box_{e_2:L} \, e_2 \to M) \\
 & & \Box \; (x \to M) \; .
\end{array}
$$

The monitor is always willing to engage in any low event. The occurrence of an x event may be thought of as a special signal recording a specific behaviour in interface L. (End of Example)

The purpose of a monitor is to observe and record the activity in some interface, and thus particularly useful within a concurrent composition with processes which communicate in that interface. Monitor M of Example 3.3.5 could be applied to monitor the behaviour of process

$$
Q \quad \widehat{=} \quad a \to (a \to Q \sqcap b \to Q) \; ,
$$

in the form of a composition $Q \; |[\{a,b\}]| \; M$. The synchronization on $\{a, b\}$ ensures that any occurrence of such an event can be taken into account by M. The monitor may chose to engage in event x whenever there have been two consecutive L events in the trace of Q. Thus, the monitor can signal through the occurrence of x that an even number of low events must have taken place.

The following result shows that a concurrent user process communicating in H with monitor M does not affect the ability of M to monitor interface L.

Proposition 3.3.6 If M is a monitor of interface L then so is $M \mathbin{|[H]|} U_H$, for any user process U_H communicating in H.

Proof From Proposition 2.2.27 and the fact that deterministic processes have no proper refinements. $\square$

We complete this subsection by giving an explicit way of constructing a monitor process from a given process P.

Definition 3.3.7 Let Σ be partitioned by L, H. For a non-diverging process P, we define $Mon_L(P)$ by

$$\begin{array}{rcl} \textsc{Fails}(Mon_L(P)) & \mathrel{\widehat{=}} & \{\, (s, X \cap H) \mid (s, X) \in \textsc{Fails}(P) \,\} \cup \\ & & \{\, (s \frown \langle a \rangle \frown t,\, Y) \mid s \in \textsc{Traces}(P) \wedge \\ & & \quad \langle a \rangle \frown t \in L^* \wedge a \notin (P \;\underline{\mathsf{after}}\; s)^0 \,\} \\ \textsc{Divs}(Mon_L(P)) & \mathrel{\widehat{=}} & \emptyset \end{array}$$

This definition has been chosen to ensure that $Mon_L(P)$ is a monitor of interface L, provided that the original process P does not diverge.

Semi-separability

The notion of semi-separability is based on the idea of splitting a process into two component processes. The first component communicates in L only, and describes the entire behaviour possible at the L interface. The second component process communicates in both H and L. The behaviour of this process includes not only the whole range of high activity, but also the possibility of engaging in low events, which are synchronized with the first process. The purpose of this synchronization is to monitor the behaviour of the low user (for instance, by keeping a record of all L actions which may have occurred).

The monitoring activity has to be completely transparent to the low user – in particular there must be no restrictions on L behaviour that is possible in the original process. This process is said to be semi-separable if and only if such a transparent monitoring process exists.

Definition 3.3.8 Process P is *semi-separable* with respect to interface L if, and only if, there exists a process Q with interface L and a monitor process M of L such that $P = Q \;|[L]|\; M$. In the case where process Q is deterministic, we say that P is *strongly semi-separable* with respect to interface L.

There are two aspects to the definition of (strong) semi-separability. The first concerns the role of M. This process captures the behaviour in interface H completely, while continuously offering to engage in any L events. We insist on M being a monitor for two reasons: (i) to exclude trivial solutions for expressing the parallel composition (such as $Q \mathrel{\widehat{=}} Run_L$); and (ii) to ensure that M cannot affect the behaviour in the L interface. This brings us to the second aspect: all information about interface L in P has been compressed into process Q. As the following result shows, this process is simply given by lazy abstraction.

Proposition 3.3.9 If process P is semi-separable with respect to interface L, then $\mathcal{L}_H(P)$ completely describes the behaviour in L.

Proof Assuming that P is semi-separable as $Q \;|[L]|\; M$ for some monitor M of L, we show that $\mathcal{L}_H(P) = Q$.

$$
\begin{array}{ll}
 & \mathcal{L}_H(P) \\
= & \qquad\qquad\qquad\qquad [\ P \text{ semi-separable wrt } L\ ] \\
 & \mathcal{L}_H(Q \;|[L]|\; M) \\
= & \qquad\qquad\qquad\qquad [\ \text{Proposition 2.2.24}\ ] \\
 & \mathcal{L}_H(Q) \;|[L]|\; \mathcal{L}_H(M) \\
= & \qquad\qquad\qquad\qquad [\ Q \text{ communicates within } L\ ] \\
 & Q \;|[L]|\; \mathcal{L}_H(M) \\
= & \qquad\qquad\qquad\qquad [\ M \text{ is a monitor of } L\ ] \\
 & Q \;|[L]|\; Run_L \\
= & \qquad\qquad\qquad\qquad [\ \text{Property of } |[L]| \text{ and } Run_L\ ] \\
 & Q\ ,
\end{array}
$$

as required. □

Example 3.3.10 Let $\Sigma \mathrel{\widehat{=}} \{a, b, x\}$ and $L \mathrel{\widehat{=}} \{a, b\}$. Consider the process

$$\begin{array}{rcl} P & \widehat{=} & a \to (a \to P \sqcap b \to P) \\ & & \quad \Box \; x \to a \to (a \to P \sqcap b \to P) \; . \end{array}$$

The L interface repeatedly allows an a event followed by an internal choice between a and b. After an even number of low events, a single x can also occur. The L interface can be described as a separate process, namely

$$Q \quad \widehat{=} \quad a \to (a \to Q \sqcap b \to Q) \; ,$$

which is equivalent to $\mathcal{L}_H(P)$. The monitor process M of Example 3.3.5 may be used as a monitor of interface L. Process P is semi-separable with respect to L because it is equivalent to $Q \; |[L]| \; M$. (End of Example)

We proceed by illustrating some key properties of semi-separability. If we can express process P equivalently as a parallel composition $\mathcal{L}_H(P) \; |[L]| \; M$, where M does not need to be a monitor, then the lazy abstraction of interface L in P refines the lazy abstraction of L in M.

Proposition 3.3.11 If $P = \mathcal{L}_H(P) \; |[L]| \; M$ for some process M, then

$$\mathcal{L}_L(M) \quad \sqsubseteq \quad \mathcal{L}_L(P) \; .$$

Proof Assume that $P = \mathcal{L}_H(P) \; |[L]| \; M$.

$$\begin{array}{ll} & \mathcal{L}_L(M) \\ = & \qquad\qquad [\; \mathcal{L}_L(M) \text{ communicates within } H \;] \\ & \mathit{Stop} \; |[L]| \; \mathcal{L}_L(M) \\ = & \qquad\qquad [\; \text{Proposition 2.2.21} \;] \\ & \mathcal{L}_L(\mathcal{L}_H(P)) \; |[L]| \; \mathcal{L}_L(M) \\ \sqsubseteq & \qquad\qquad [\; \text{Proposition 2.2.24} \;] \\ & \mathcal{L}_L(\mathcal{L}_H(P) \; |[L]| \; M) \\ = & \qquad\qquad [\; \text{Assumption} \;] \\ & \mathcal{L}_L(P) \end{array}$$

which completes the proof. $\Box$

Proposition 3.2.5 has shown that the presence of a concurrent user process U_H synchronizing in H does not affect interface L, provided it is independent. A similar result holds if the process concerned is semi-separable. It should be noted, however, that here the abstract processes $\mathcal{I}_H(P)$ and $\mathcal{L}_H(P)$ need not be deterministic.

Proposition 3.3.12 If process P is semi-separable with respect to interface L, and U_H is any user processes communicating in H only, then

$$\begin{array}{lcl}\mathcal{I}_H(P) & = & \mathcal{I}_H(P \;|[H]|\; U_H) \;\wedge \\ \mathcal{L}_H(P) & = & \mathcal{L}_H(P \;|[H]|\; U_H)\;.\end{array}$$

Proof We show the result for interleaving. Since P is semi-separable, it can be equivalently expressed as $Q \;|[L]|\; M$, for some process Q communicating in L and some monitor process M communicating in $H \cup L$.

$$\begin{array}{lll}
 & \mathcal{I}_H(P) & \\
= & & [\;\; P \text{ semi-separable wrt } L \;\;] \\
 & \mathcal{I}_H(Q \;|[L]|\; M) & \\
= & & [\;\; \text{Proposition 2.2.37} \;\;] \\
 & \mathcal{I}_H(Q) \;|[L]|\; \mathcal{I}_H(M) & \\
= & & [\;\; M \text{ is a monitor of } L \;\;] \\
 & \mathcal{I}_H(Q) \;|[L]|\; Run_{H \cup L} & \\
= & & [\;\; M \;|[H]|\; U_H \text{ is a monitor of } L \;\;] \\
 & \mathcal{I}_H(Q) \;|[L]|\; \mathcal{I}_H(M \;|[H]|\; U_H) & \\
= & & [\;\; \text{Proposition 2.2.37} \;\;] \\
 & \mathcal{I}_H(Q \;|[L]|\; (M \;|[H]|\; U_H)) & \\
= & & [\;\; \text{Associativity of parallel} \;\;] \\
 & \mathcal{I}_H((Q \;|[L]|\; M) \;|[H]|\; U_H) & \\
= & & [\;\; P \text{ semi-separable wrt } L \;\;] \\
 & \mathcal{I}_H(P \;|[H]|\; U_H) &
\end{array}$$

as required. The case for lazy abstraction is very similar. □

Relation between Conditions

In this subsection, we show that the relationship between the symmetric and asymmetric conditions of separation is that implied by their names: separability corresponds to semi-separability with respect to both interfaces.

Lemma 3.3.13 If process P is semi-separable with respect to both H and L, then it is separable.

Proof Assume that P is semi-separable, that is

$$P = \mathcal{L}_H(P) \; |[L]| \; M \;\; \wedge \tag{3.2}$$

$$P = \mathcal{L}_L(P) \; |[H]| \; M' \tag{3.3}$$

for suitable monitors M of L and M' of H. To prove that P is separable it is sufficient to show $P \sqsubseteq \mathcal{L}_H(P) \; ||| \; \mathcal{L}_L(P)$, so suppose that $(w, X_L \cup X_H) \in \mathrm{F{\scriptsize AILS}}(\mathcal{L}_H(P) \; ||| \; \mathcal{L}_L(P))$ where $X_H \subseteq H$ and $X_L \subseteq L$.

The failure $(w, X_H \cup X_L)$ must have been caused by failures

$$(s, X_L) \in \mathrm{F{\scriptsize AILS}}(\mathcal{L}_H(P)) \tag{3.4}$$

$$(t, X_H) \in \mathrm{F{\scriptsize AILS}}(\mathcal{L}_L(P)) \tag{3.5}$$

where $s \in L^*$ and $t \in H^*$ and $w \in s \; ||| \; t$. Now w is also a trace of M' (a monitor of H), and therefore $(w, \emptyset) \in \mathrm{F{\scriptsize AILS}}(M')$. Then:

$$\begin{array}{ll}
 & (t, X_H) \in \mathrm{F{\scriptsize AILS}}(\mathcal{L}_L(P)) \;\wedge\; (w, \emptyset) \in \mathrm{F{\scriptsize AILS}}(M') \\
\Rightarrow & \hfill [\ \text{Definition of } |[H]|\ ] \\
 & (w, X_H) \in \mathrm{F{\scriptsize AILS}}(\mathcal{L}_L(P) \; |[H]| \; M') \\
\Rightarrow & \hfill [\ \mathcal{L}_L(P) \; |[H]| \; M' = \mathcal{L}_H(P) \; |[L]| \; M\ ] \\
 & (w, X_H) \in \mathrm{F{\scriptsize AILS}}(\mathcal{L}_H(P) \; |[L]| \; M) \\
\Rightarrow & \hfill [\ \text{Definition of } |[L]|\ ] \\
 & (w, X_H) \in \mathrm{F{\scriptsize AILS}}(M) \; .
\end{array}$$

The failure (w, X_H) of M is needed for the final step

$$\begin{array}{ll}
 & (s, X_L) \in \mathrm{F{\scriptsize AILS}}(\mathcal{L}_H(P)) \;\wedge\; (w, X_H) \in \mathrm{F{\scriptsize AILS}}(M) \\
\Rightarrow & \hfill [\ \text{Definition of } |[L]|\ ] \\
 & (w, X_L \cup X_H) \in \mathrm{F{\scriptsize AILS}}(\mathcal{L}_H(P) \; |[L]| \; M) \\
\Rightarrow & \hfill [\ \text{by } (3.2)\ ] \\
 & (w, X_L \cup X_H) \in \mathrm{F{\scriptsize AILS}}(P)
\end{array}$$

which is what we wanted to prove. □

Theorem 3.3.14 Process P is separable if, and only if, P is semi-separable with respect to both L and H.

Proof (i) We first show that separability implies mutual semi-separability. If P is separable, then it is possible to split P into two disjoint components: $P = Q_L \mid\mid\mid Q_H$, having interfaces L and H respectively. To show semi-separability with respect to these interfaces we can chose monitor processes as follows:

$$\begin{array}{rcl} M & \mathrel{\widehat{=}} & \mathcal{I}_L(Q_H) \ , \\ M' & \mathrel{\widehat{=}} & \mathcal{I}_H(Q_L) \ . \end{array}$$

P is semi-separable with respect to interface L because M is a monitor of L and $P = Q_L \mid[L]\mid M$. An analogous result holds for interface H.

(ii) The converse holds according to Lemma 3.3.13. □

In the special case where the component processes are deterministic, we conclude from Theorem 3.3.14 that strong separability corresponds to mutual strong semi-separability with respect to both interfaces, as expected. It is noteworthy how the determinism of the processes involved can be exploited to lead to much simpler proofs based on the algebraic properties of abstractions, such as the proof of the corresponding result described in Lemma 3.3.13.

To show that mutual strong semi-separability implies strong separability, we recall the algebraic properties of lazy abstraction. If P is strongly semi-separable with respect to both L and H, then the interfaces are given by processes $\mathcal{L}_H(P)$ and $\mathcal{L}_L(P)$ by Proposition 3.3.9, and these processes are deterministic. Therefore the interleaving $\mathcal{L}_H(P) \mid\mid\mid \mathcal{L}_L(P)$ must also be deterministic. Now in general

$$\mathcal{L}_H(P) \mid\mid\mid \mathcal{L}_L(P) \quad \sqsubseteq \quad P$$

by Proposition 2.2.30. Since deterministic processes have no proper refinements, equality must hold, which means that P is strongly separable.

3.4 Discussion

In this chapter, we have advanced reasons for adopting the condition of independence as the most satisfactory criterion for the security of communicating processes with two (or more) interfaces. Originally introduced by Roscoe, Woodcock, and Wulf [RWW94] and developed in later publications [RW95, Ros95, RWW96], independence requires an interface not only to be invariant under events occurring at other interfaces, but also to be deterministic. As a consequence, independence guarantees a consistent and reliable interface to an external agent communicating with the process in that interface, irrespective of simultaneous activity at other interfaces.

The independence condition can be characterised in a variety of ways, the most intriguing being determinism under a suitable abstraction function (Theorem 3.2.3). Although lazy abstraction and interleaving have quite different effects on processes to which they are applied – lazy abstraction removes events from traces, while interleaving adds events – both abstractions agree precisely on which processes are deterministic. The algebraic properties of these abstractions together with the fact that deterministic processes are refined only by themselves can often be exploited in proofs of elegant simplicity, such as those that relate the various notions of separation and composition presented in Section 3.3.

Separability, semi-separability, and their deterministic counterparts do provide strong supporting evidence for independence as a security criterion. We regard the general conditions of separation adequate for systems constructed from physically isolated components, but insufficient for shared systems where the absence of information flow cannot be guaranteed if interfaces are underspecified. The reason for insisting on *strong* separation is that internal (non-deterministic) choices in a specification can in general not be assumed to be implemented in a secure fashion. Several examples of this undesirable phenomenon have been given earlier in this chapter.

A possible critique of independence is that condition can only be applied to processes in which external agents have complete control over the events they communicate with a process. The control is not only over the choice of next event to engage in, but also over the choice of proceeding with or delaying an event. The security of a system should not depend on the assumption that cooperation of other agents is always forthcoming.

In practice, therefore, it is frequently desirable to model events which do not require the cooperation of external agents. A typical example of these are output or signal events [RWW94, Ros95] which – although likely to be occurring as a consequence of some agent's prior activity – cannot be suppressed or delayed. For instance, the display of a message on a computer screen, an instant acknowledgement in reply to a user input, or an audible bell (e.g. to signal the arrival of email) should be regarded as signals. Any of these events may be observed but they occur without participation of their environment and thus do not affect the activity in the complement interface.

Transaction security guarantees the absence of interference for systems in which there are events which "complete" the transactions initiated by external agents communicating with the system. These events are assumed to be outside of the control of the system's environment. Like Foley [Fol87], we partition interfaces into events which an agent *controls*, and events which can be *observed* but cannot be delayed. Security is achieved only if these *signal events* are always guaranteed to occur without delay.

As the signal events typically occur in response to requests of external agents, such as a user's request for system resources, we refer to the set of signals as the "completion set" K. In the formal definition of transaction security, it is necessary to apply a form of "eager" abstraction to this completion set to represent the assumption that signals cannot be delayed.

Definition 3.4.1 Assume that Σ is partitioned by the three sets H, K, and L. Interface L is said to be independent under the completion set K in process P, written $H \setminus\!\!\setminus K \not\longrightarrow L :: P$ if, and only if, the process

$$\mathcal{L}_H(\mathcal{H}_K(P))$$

is deterministic.

In the case where interface L is independent under the above condition, we say that the process is transaction-secure with respect to completions K. Since the order of abstraction does not matter if H and K are disjoint, and lazy abstraction and interleaving agree on which processes are deterministic, any of the following equivalent conditions may be used to verify transaction security.

1. $\mathcal{L}_H(\mathcal{H}_K(P))$ is deterministic;
2. $\mathcal{I}_H(\mathcal{H}_K(P))$ is deterministic;
3. $\mathcal{H}_K(\mathcal{L}_H(P))$ is deterministic;
4. $\mathcal{H}_K(\mathcal{I}_H(P))$ is deterministic.

Neither independence nor transaction independence suffer from the refinement paradox, because lazy abstraction, interleaving, and hiding are all abstractions which preserve the refinement order.

Theorem 3.4.2 Independence of interface L under both standard and transaction security is preserved by refinement

$$\begin{array}{rcl} H \not\longrightarrow L :: P \wedge P \sqsubseteq Q & \Rightarrow & H \not\longrightarrow L :: Q \\ H \setminus\!\setminus K \not\longrightarrow L :: P \wedge P \sqsubseteq Q & \Rightarrow & H \setminus\!\setminus K \not\longrightarrow L :: Q \ . \end{array}$$

As a consequence, the specification of a secure system may be stepwise refined in the knowledge that, at the end of the development process, all possible implementations are guaranteed to be secure.

Chapter 4

Development

In this chapter, we address the issue of developing systems which meet the desired security constraints. The aim of the development process is to derive a deterministic system description, satisfying the independence criteria of Chapter 3, which may then be implemented as a centralised or physically distributed computing system.

In Section 4.1, a number of laws concerning the formal construction of systems satisfying (transaction) independence will be presented. The laws include those for sequential as well as parallel operators. The relevance of these compositional laws lies in the fact that one can build a system model by composition of smaller and simpler components. The compositional laws ensure that security properties will be preserved during system development. The application of these laws can thus make it unnecessary to carry out a potentially difficult verification of security properties on the complete system model.

Section 4.2 presents the technique of unwinding, which has traditionally been used to verify the security of systems. The technique relies on the verification of conditions on individual state transitions, rather than the global system state, and can help to gain additional assurance in the security of the system.

In the initial sections of this chapter we still make the assumption of two interfaces to the system, one high H and one low L. This assumption is relaxed in Section 4.3 where it is shown how our conditions can be applied to multi-level secure systems. The added complexity of multi-level security makes it desirable to start system development with the formulation of a security policy. The policy is used to provide a concise statement of security issues on which later verification can be based.

4.1 Compositional Laws

Compositional laws form an integral part of any theory of secure system development. The relevance of such laws lies in the possibility of compositional verification of security, either during the development process – individual system components are checked for security before being combined into an overall design – or subsequent to the derivation of the deterministic implementation. The compositional approach makes it possible to provide convincing arguments for the security of a system consisting of multiple components whose interaction may be complex. As Meadows [Mea95] has observed,

> "... it is necessary to be able to identify the critical and non-critical parts of the system, have methods of assurance based on compositionality, so that we know what properties are possessed by a system whose components satisfy certain properties, and to be able to develop convincing assurance arguments that support certification and accreditation of secure systems."

Compositional verification is of particular importance where tool support, such as the FDR model checker, is employed to provide assurance of security properties. A model checker relies typically on the exhaustive enumeration of system states. For large systems, this approach will lead to state explosion, making it infeasible to verify the security of a complete system model automatically.

Our formulation of security properties, however, supports such a compositional approach. When the system model is given by composition of smaller (and simpler) components, which often have a much smaller state space, the security properties can be confirmed on these component processes. The compositional laws show exactly how the individual components may be combined into a system, while preserving the desired properties. We start by considering a number of common sequential operators, such as prefixing of events, choice, and hiding. In the subsequent section the behaviour of our security conditions under parallel composition will be detailed.

Sequential Processes

A deterministic process is secure if there are no events in its alphabet which may be a cause of interference. The following law states that the alphabet αP of process P is independent from H provided the two sets are disjoint

Theorem 4.1.1

$$\begin{array}{l|l} \text{a.} & P \text{ communicates within } L \\ \text{b.} & P \text{ deterministic} \\ \hline & H \not\rightarrowtail L :: P \end{array}$$

The next law states that security of process P_a is preserved under the prefix choice operator. The set of events A from which the initial event may be chosen must be a subset of the events independent in P_a

Theorem 4.1.2

$$\begin{array}{l|l} \text{a.} & H \not\rightarrowtail L :: P_a \\ \text{b.} & A \subseteq L \\ \hline & H \not\rightarrowtail L :: \Box_{a:A}\, a \rightarrow P_a \end{array}$$

Prefixing a secure process with a choice of events from a subset of H would not preserve security. The reason for this lies in the nature of prefixing: the chosen initial event has to occur first before the remaining behaviour of P could be observed, thus allowing an observer of interface L to deduce whether or not the initial event has taken place. However, one may get around this situation by offering a choice between a number of prefixed events from $X \subseteq H$ and process P itself, thus prohibiting any deductions about the prefixed high events.

Theorem 4.1.3

$$\begin{array}{l|l} \text{a.} & H \not\rightarrowtail L :: P \\ \text{b.} & X \subseteq H \\ \hline & H \not\rightarrowtail L :: P \,\Box\, (\Box_{x:X}\, x \rightarrow P) \end{array}$$

Suppose that P and Q are secure processes. In general the choice between these processes, $P \,\Box\, Q$, is not secure because an initial H action from process P will remove the option of communicating within L in process Q, and vice versa. Security is assured only when the initial choice between P and Q is solely made within L. This can be achieved by prefixing either process with an event from L.

Theorem 4.1.4

$$\begin{array}{r|l} \text{a.} & H \not\leftrightarrow L :: P_a \\ \text{b.} & H \not\leftrightarrow L :: Q_b \\ \text{c.} & A \subseteq L \\ \text{d.} & B \subseteq L \\ \text{e.} & A \cap B = \emptyset \\ \hline & H \not\leftrightarrow L :: (\Box_{a:A}\, a \to P_a) \Box (\Box_{b:B}\, b \to Q_b) \end{array}$$

Hiding

If interface L is independent of $H \cup C$ in process P, and hiding C does not result in divergence, then L is independent under the completions C. This follows immediately from the definition of transaction security (Definition 3.4.1).

Theorem 4.1.5

$$\begin{array}{r|l} \text{a.} & H \cup C \not\leftrightarrow L :: P \\ \text{b.} & \mathcal{H}_C(P) \text{ non-divergent} \\ \hline & H \setminus\!\!\setminus C \not\leftrightarrow L :: P \end{array}$$

If interface L is independent from H under the completions C in process P, then events C are guaranteed to occur instantaneously, and we can equally well hide C from the external interfaces of P. This allows us to state the standard security condition for $P \setminus C$ rather than the transaction security condition for P.

Theorem 4.1.6

$$\begin{array}{r|l} \text{a.} & H \setminus\!\!\setminus C \not\leftrightarrow L :: P \\ \hline & H \not\leftrightarrow L :: P \setminus C \end{array}$$

A similar result holds for transaction security: if interface L is independent from H under completions $C \cup K$, then we can hide a subset C of the completions to obtain transaction security for process $P \setminus C$.

Theorem 4.1.7

$$\begin{array}{l|l} \text{a.} & H \setminus\!\!\setminus C \cup K \not\leftrightarrow L :: P \\ \hline & H \setminus\!\!\setminus K \not\leftrightarrow L :: P \setminus C \end{array}$$

Parallel Processes

In this subsection, we consider the security of processes under parallel composition. In the following, we assume that P is a process communicating within alphabet A, and Q is a process communicating in alphabet B.

The first law shows that independence is preserved under parallel composition. If L_1 is independent in P and L_2 is independent in Q, then $L_1 \cup L_2$ will be independent when both processes proceed concurrently.

Theorem 4.1.8

$$\begin{array}{l|l} \text{a.} & H \not\leftrightarrow L_1 :: P \\ \text{b.} & H \not\leftrightarrow L_2 :: Q \\ \hline & H \not\leftrightarrow L_1 \cup L_2 :: P \, [A \| B] \, Q \end{array}$$

Proof We exploit distribution of lazy abstraction over parallel composition (Proposition 2.2.23), and the fact that deterministic processes have no proper refinements.

$\mathcal{L}_H(P)$ deterministic $\wedge$ $\mathcal{L}_H(Q)$ deterministic

$\Rightarrow$ [Proposition A.2.1]

$\mathcal{L}_H(P) \, [A \| B] \, \mathcal{L}_H(Q)$ deterministic

$\Rightarrow$ [Proposition 2.2.23]

$\mathcal{L}_H(P \, [A \| B] \, Q)$ deterministic

which is what we wanted to prove. □

The following law shows that it is safe to constrain the high interface H of a secure process P by parallel composition of a process communicating within H only, provided the new component is free of divergence.

Theorem 4.1.9

a. $H \not\rightarrowtail L :: P$

b. Q does not diverge

c. $B \subseteq H$

$H \not\rightarrowtail L :: P\,[A\|B]\,Q$

Proof We exploit that fact that $\mathcal{L}_H(Q) = \mathit{Stop}$ under the assumptions, and that Stop is a deterministic process.

$$
\begin{array}{ll}
 & \mathcal{L}_H(P) \text{ deterministic } \wedge\ \mathit{Stop} \text{ deterministic} \\
\Rightarrow & \qquad\qquad\qquad [\ \text{Proposition 2.2.21}\ ] \\
 & \mathcal{L}_H(P) \text{ deterministic } \wedge\ \mathcal{L}_H(Q) \text{ deterministic} \\
\Rightarrow & \qquad\qquad\qquad [\ \text{Proposition A.2.1}\ ] \\
 & \mathcal{L}_H(P)\,[A\|B]\,\mathcal{L}_H(Q) \text{ deterministic} \\
\Rightarrow & \qquad\qquad\qquad [\ \text{Proposition 2.2.23}\ ] \\
 & \mathcal{L}_H(P\,[A\|B]\,Q) \text{ deterministic}
\end{array}
$$

which is what we wanted to prove. □

The following two laws show that it is safe to add a deterministic process in parallel to a secure process, provided that the process alphabet is disjoint from the events available in the high interface.

Theorem 4.1.10

a. $H \not\rightarrowtail L :: P$

b. $B \cap H = \emptyset$

c. Q deterministic

$H \not\rightarrowtail L :: P\,[A\|B]\,Q$

Proof We exploit that fact that $\mathcal{L}_H(Q) = Q$ under the assumptions.

$$
\begin{array}{ll}
 & \mathcal{L}_H(P) \text{ deterministic } \wedge\ Q \text{ deterministic} \\
\Rightarrow & \qquad\qquad\qquad [\ \text{Proposition 2.2.22}\ ] \\
 & \mathcal{L}_H(P) \text{ deterministic } \wedge\ \mathcal{L}_H(Q) \text{ deterministic} \\
\Rightarrow & \qquad\qquad\qquad [\ \text{Proposition A.2.1}\ ] \\
 & \mathcal{L}_H(P)\,[A\|B]\,\mathcal{L}_H(Q) \text{ deterministic} \\
\Rightarrow & \qquad\qquad\qquad [\ \text{Proposition 2.2.23}\ ] \\
 & \mathcal{L}_H(P\,[A\|B]\,Q) \text{ deterministic}
\end{array}
$$

which is what we wanted to prove. □

The law also holds for processes with transaction security under completions K, provided that the alphabet of the process being added in parallel is disjoint from the (complete) set of high events.

$$\textbf{Theorem 4.1.11} \quad \begin{array}{r|l} \text{a.} & H \mathbin{\backslash\!\backslash} K \not\longrightarrow L :: P \\ \text{b.} & B \cap (H \cup K) = \emptyset \\ \text{c.} & Q \text{ deterministic} \\ \hline & H \mathbin{\backslash\!\backslash} K \not\longrightarrow L :: P \, [A \| B] \, Q \end{array}$$

Proof We exploit that fact that $\mathcal{L}_H(Q) \setminus (K \cap B) = Q$ under the assumptions.

$$\begin{array}{ll} & \mathcal{L}_H(\mathcal{H}_K(P)) \text{ deterministic } \wedge \; Q \text{ deterministic} \\ \Rightarrow & \qquad\qquad [\text{ Definition 3.4.1 and Proposition 2.2.22 }] \\ & \mathcal{L}_H(P) \setminus K \text{ deterministic } \wedge \; \mathcal{L}_H(Q) \text{ deterministic} \\ \Rightarrow & \qquad\qquad [\text{ Assumption on alphabets and (b.) }] \\ & \mathcal{L}_H(P) \setminus K \cap A \text{ deterministic } \wedge \; \mathcal{L}_H(Q) \setminus K \cap B \text{ deterministic} \\ \Rightarrow & \qquad\qquad [\text{ Proposition A.2.1 }] \\ & (\mathcal{L}_H(P) \setminus K \cap A) \; [A \| B] \; (\mathcal{L}_H(Q) \setminus K \cap B) \text{ deterministic} \\ \Rightarrow & \qquad\qquad [\text{ Equation } (A.7) \;] \\ & (\mathcal{L}_H(P) \, [A \| B] \, \mathcal{L}_H(Q)) \setminus K \text{ deterministic} \\ \Rightarrow & \qquad\qquad [\text{ Proposition 2.2.23 }] \\ & \mathcal{L}_H(P \, [A \| B] \, Q) \setminus K \text{ deterministic} \end{array}$$

which is what we wanted to prove. □

The following law shows how to compose two processes which both satisfy transaction security; we need to ensure that the two completion sets K_1 and K_2 are disjoint.

$$\textbf{Theorem 4.1.12} \quad \begin{array}{r|l} \text{a.} & H \mathbin{\backslash\!\backslash} K_1 \not\longrightarrow L :: P \\ \text{b.} & H \mathbin{\backslash\!\backslash} K_2 \not\longrightarrow L :: Q \\ \text{c.} & K_1 \cap K_2 = \emptyset \\ \hline & H \mathbin{\backslash\!\backslash} K_1 \cup K_2 \not\longrightarrow L :: P \, [A \| B] \, Q \end{array}$$

Proof We exploit the assumption that the completion sets are disjoint.

$$
\begin{array}{ll}
& \mathcal{L}_H(\mathcal{H}_{K_1}(P)) \text{ deterministic} \wedge \mathcal{L}_H(\mathcal{H}_{K_2}(Q)) \text{ deterministic} \\
\Rightarrow & \hfill [\text{ Definition 3.4.1 and Proposition A.2.1 }] \\
& \mathcal{L}_H(P) \setminus K_1 \; [A \| B] \; \mathcal{L}_H(Q) \setminus K_2 \text{ deterministic} \\
\Rightarrow & \hfill [\text{ Assumption on alphabets and (c.) }] \\
& \mathcal{L}_H(P) \setminus ((K_1 \cup K_2) \cap A) \; [A \| B] \\
& \mathcal{L}_H(Q) \setminus ((K_1 \cup K_2) \cap B) \text{ deterministic} \\
\Rightarrow & \hfill [\text{ Equation } (A.7) \;] \\
& (\mathcal{L}_H(P) \, [A \| B] \, \mathcal{L}_H(Q)) \setminus K_1 \cup K_2 \text{ deterministic} \\
\Rightarrow & \hfill [\text{ Definition 3.4.1 }] \\
& \mathcal{L}_H(\mathcal{H}_{K_1 \cup K_2}(P \, [A \| B] \, Q)) \text{ deterministic}
\end{array}
$$

as required. □

McCullough [McC88] stresses the importance of being able to "hook-up" secure components by parallel composition in a manner that preserves their non-interference properties. The compositional results above show how to develop hook-up secure systems with independent interfaces.

4.2 Unwinding

The verification of security conditions such as those in Sections 3.1 and 3.2 requires inductive proof over all traces of the process. Unwinding, originally introduced by Goguen and Meseguer [GM84], has been proposed to avoid lengthy inductive proofs to verify security properties. The technique involves finding conditions on individual state transitions, rather than the global state of the process, which are sufficient to guarantee the desired security properties [JT88, Rya91, Mil95].

We will focus attention on the following condition, which requires that a single H step does not change the process interface under the interleaving abstraction.

Definition 4.2.1 A process P satisfies LR, if

$$x \in H \wedge s \frown \langle x \rangle \in \text{TRACES}(P) \quad \Rightarrow \quad P \; \underline{\text{after}} \; s \frown \langle x \rangle \equiv_H^{\mathcal{I}} P \; \underline{\text{after}} \; s \; .$$

The condition has been termed LR with reference to Rushby's condition "local respect" [Rus92]. If individual H steps respect the L interface according to

Definition 4.2.1, then the L events which may be accepted next in any particular state are completely determined.

Lemma 4.2.2 Suppose that Σ is partitioned by H and L. If P satisfies LR, then interface L is trace-invariant in P.

Thus the verification of condition LR ensures that any interspersed x events (outside L) cannot change the set of events accepted at the L interface. It is possible to prove an even stronger result, namely that LR provides an alternative way of characterising invariance.

Theorem 4.2.3 Process P satisfies LR if, and only if, interface L is invariant in P.

Proof (i) The proof of invariance under LR is carried out by induction on the traces of the process.

Base case $t = \langle\rangle$. Invariance of L is immediate since $\langle\rangle \setminus H = \langle\rangle$.

Inductive step $t = s \frown \langle e \rangle$. Suppose Theorem 4.2.3 holds for trace s of P, and consider the extension of the trace by event e.

Case $e \in H$. Then

$$\begin{aligned} t \setminus H &= s \frown \langle e \rangle \setminus H \\ &= s \setminus H \end{aligned}$$

is a trace of P because of the inductive hypothesis applied to the prefix s of t. Furthermore,

$$\begin{aligned} & \mathcal{I}_H(P \,\underline{\text{after}}\, s \frown \langle e \rangle) \\ = & \qquad\qquad\qquad\qquad [\ LR\ ] \\ & \mathcal{I}_H(P \,\underline{\text{after}}\, s) \\ = & \qquad\qquad\qquad\qquad [\ \text{Hypothesis}\ ] \\ & \mathcal{I}_H(P \,\underline{\text{after}}\, s \setminus H) \\ = & \qquad\qquad\qquad\qquad [\ \text{This case}\ ] \\ & \mathcal{I}_H(P \,\underline{\text{after}}\, (s \frown \langle e \rangle) \setminus H) \end{aligned}$$

and thus $P \,\underline{\text{after}}\, t \equiv^{\mathcal{I}}_H P \,\underline{\text{after}}\, t \setminus H$.

Case $e \in L$. Then

$$\begin{array}{rcl} t \setminus H & = & s \frown \langle e \rangle \setminus H \\ & = & (s \setminus H) \frown \langle e \rangle \end{array}$$

is a trace of P because event e is accepted after trace s by assumption, and

$$(P \underline{\text{after}}\, s)^0 \cap L \quad = \quad (P \underline{\text{after}}\, s \setminus H)^0 \cap L$$

by trace-invariance of L (Lemma 4.2.2). Furthermore

$$\begin{array}{ll} & \mathcal{I}_H(P \underline{\text{after}}\, s \frown \langle e \rangle) \\ = & \qquad\qquad [\text{ Proposition 2.2.42 }] \\ & (\mathcal{I}_H(P \underline{\text{after}}\, s))\, \underline{\text{after}}\, \langle e \rangle \\ = & \qquad\qquad [\text{ Hypothesis }] \\ & (\mathcal{I}_H(P \underline{\text{after}}\, s \setminus H))\, \underline{\text{after}}\, \langle e \rangle \\ = & \qquad\qquad [\text{ Proposition 2.2.42 }] \\ & \mathcal{I}_H(P \underline{\text{after}}\, (s \setminus H)\, \underline{\text{after}}\, \langle e \rangle) \\ = & \qquad\qquad [\text{ This case }] \\ & \mathcal{I}_H(P \underline{\text{after}}\, (s \frown \langle e \rangle) \setminus H) \end{array}$$

and thus $P \underline{\text{after}}\, t \equiv_H^{\mathcal{I}} P \underline{\text{after}}\, t \setminus H$. This completes the inductive step.

(ii) If interface L is invariant, then events outside L in a trace of P do not affect the abstract process under interleaving by Theorem 3.1.2. Therefore, if $s \frown \langle x \rangle$ is a trace of P and $x \in H$ then

$$\begin{array}{ll} & \mathcal{I}_H(P \underline{\text{after}}\, s \frown \langle x \rangle) \\ = & \qquad\qquad [\text{ Invariance of } L \text{ }] \\ & \mathcal{I}_H(P \underline{\text{after}}\, (s \frown \langle x \rangle \setminus H)) \\ = & \qquad\qquad [\text{ Assumption } x \in H \text{ }] \\ & \mathcal{I}_H(P \underline{\text{after}}\, (s \setminus H)) \\ = & \qquad\qquad [\text{ Invariance of } L \text{ }] \\ & \mathcal{I}_H(P \underline{\text{after}}\, s)\ , \end{array}$$

which means that LR holds. This completes the proof of Theorem 4.2.3. □

An unwinding result similar to the one above was proved by Ryan [Rya91], albeit in a very different presentation in terms of state machines. Abstraction functions can serve a useful purpose in finding and expressing unwinding results in a concise and elegant manner. However, availability of compositional development techniques together with tool support for automated verification diminishes the practical importance of such results.

4.3 Multi-level Systems

In this section, we extend our approach to secure system specification to systems with multi-level security requirements. We believe that the development of multi-level secure is not significantly different from the development of bi-level secure systems, and argue that the same types of verification conditions should be applied to either. The only additional complexity introduced by multi-level security is introduced by the fact that there a number of security classifications, rather than the levels associated with the interfaces H and L. To deal with this added complexity, a security policy can be used to summarize the permitted and prohibited flows of information between the system interfaces.

The starting point for the definition of a security policy is a clear statement of the system alphabet Σ which is being partitioned into a number of system interfaces. The system alphabet provides the boundary up to which this type of formal modelling can be carried out; security concerns beyond this boundary should be subject to corporate security policies concerning managerial and other aspects within an enterprise [BD97, Pfl97].

For our purposes, we follow a common approach and define a security policy as a relation on the interfaces of a system [Rus92, BY94].

Definition 4.3.1 A *security policy* is a reflexive and transitive relation $\rightsquigarrow$ between the interfaces of a process.

The relation $A \rightsquigarrow B$ indicates that the activity in interface A may interfere with interface B, or that information about interface A can be obtained by observation of B. The complement relation of the communication policy is denoted by $\not\rightsquigarrow$. The relation $A \not\rightsquigarrow B$ indicates the absence of information flow from interface A to B.

The security of a multi-level system is defined with respect to its security policy. To achieve this, we define two sets of "low" and "high" communication events for some interface A.

$$\begin{array}{rcl} \downarrow A & \widehat{=} & \bigcup\{\, B \mid B \rightsquigarrow A \,\}\ , \\ \uparrow A & \widehat{=} & \Sigma \setminus \downarrow A\ . \end{array}$$

A system is multi-level secure if independence holds for all suitable bi-level partitions of its interfaces.

Definition 4.3.2 Let P be a process with policy $\leadsto$. Then P is secure with respect to $\leadsto$ if, and only if,

$$\uparrow A \not\longleftrightarrow \downarrow A :: P \ ,$$

for all interfaces A of process P.

Example 4.3.3 Let P be a process with interfaces A, B, C, and D whose union make up Σ. The security policy for P is given by

$$\begin{gathered} A \leadsto A \\ B \leadsto A \wedge B \leadsto B \\ C \leadsto A \wedge C \leadsto C \\ D \leadsto A \wedge D \leadsto B \wedge D \leadsto C \wedge D \leadsto D \ . \end{gathered}$$

The policy $\leadsto$ determines the conditions which need to be verified for multi-level independence:

$$\begin{aligned} \emptyset \not\longleftrightarrow A \cup B \cup C \cup D :: P \ \wedge \\ A \cup C \not\longleftrightarrow B \cup D :: P \ \wedge \\ A \cup B \not\longleftrightarrow C \cup D :: P \ \wedge \\ A \cup B \cup C \not\longleftrightarrow D :: P \ , \end{aligned}$$

where the first condition may be omitted. (End of Example)

The approach to multi-level security outlined above relies on suitable partitions of the system interfaces into a "high" and a "low interface", and applying the standard independence condition for each partition. This approach to security is essentially transitive: if interface A interferes with B, and B interferes with C, then necessarily A interferes with C. As has been observed [Rya91], it is difficult to find meaningful non-transitive formulations of security within a CSP framework. In our experience, there are few practical systems for which the standard approach described in this section would not be applicable.

Chapter 5

Bi-level Readers and Writers

This chapter presents a larger example of what is known as the readers/writers problem [BL93] in a security context. In this problem, processes may simultaneously attempt to access data objects (or files) subject to the conditions

- several processes may read a data object at the same time, but only while no processes are writing,
- only a single process may write to a data object at any one time, and only while no processes are reading.

The readers-writers problem in a security context has first been described by Reed and Kanodia [RK79] and developed by Gray [Gr91]. In this context, there appears to be a fundamental conflict between the functional requirements above and the security requirement of absence of interference between highly-classified and lowly-classified processes. In particular,

- a high-level process reading a (low-level or high-level) object must not block a low-level process attempting to write to that object. However, if the low-level process proceeds then the other process may well read inconsistent (unexpected) values.

Reed and Kanodia [RK79] conclude that "no algorithm can simultaneously guarantee that readers will be able to complete reading and that readers can never signal writers". Below we present a solution of the secure readers-writers problem where interference between high-level and low-level is guaranteed not to occur.

Our model is general enough to encompass multiple processes, each of which may serve as reader or writer, and multiple data objects (files). Each process and file has an associated security label (classification) attached, which may be high or low.

5.1 Basic Definitions

Data items are stored in files which are distinguished with file identifiers. Readers and writers which access file information are distinguished with process identifiers, given by the set Pid. The given sets are

$$[\,Fid, Pid\,] \tag{5.1}$$

The set $Label$ denotes the security labels attached to both processes and file of the system; it is defined

$$Label \quad \widehat{=} \quad \{\,\mathbf{lo}, \mathbf{hi}\,\} \tag{5.2}$$

which will also be referred to as "low" and "high", respectively. The high label dominates the low label: $\mathbf{lo} < \mathbf{hi}$.

The set Val denotes the set of values which may be stored in each file. We restrict ourselves to the binary values $Val = \{0, 1\}$ for simplicity, and assume that there is a distinguished member $v_0 = 0$ say which will be used to initialise each file.

The following channels are required for system specification. Channel $fopen$ is used to open a file, while channel $flock$ is used to lock a file

$$\underline{\text{channel}} \quad fopen, flock \quad : Fid.Pid.Clear\ .$$

The last field of channels $fopen$ and $flock$ indicates the security level of the process which carries out the operation. Reading and writing of files is achieved through channels

$$\underline{\text{channel}} \quad fread, fwrite \quad : Fid.Pid.Val\ ,$$

where the last field of these channels provides the value being read from or written to the file. Lastly, a file is closed through channel $fclose$ and unlocked through channel $funlock$

$$\underline{\text{channel}} \quad fclose, funlock \quad : Fid.Pid \ .$$

The complete system alphabet Σ is given by

$$\Sigma \quad \widehat{=} \quad \{\!| \ fopen, fclose, flock, funlock, fread, fwrite \ |\!\} \ . \tag{5.3}$$

There are two special functions on the processes and functions, respectively, which return the security classification of their argument. $clear(p)$ returns the security label of process p, whereas $class(f)$ returns the label of file f.

$$\begin{aligned} &clear :: Pid \to Label \ , \\ &class :: Fid \to Label \ . \end{aligned} \tag{5.4}$$

Using these functions, we can partition both the set of file identifiers and process identifiers into a "low" and a "high" set.

$$\begin{aligned} Fid_L &\ \widehat{=}\ \{f \mid f \in Fid \wedge class(f) = \mathbf{lo}\} \\ Fid_H &\ \widehat{=}\ \{f \mid f \in Fid \wedge class(f) = \mathbf{hi}\} \\ Pid_L &\ \widehat{=}\ \{p \mid p \in Pid \wedge clear(p) = \mathbf{lo}\} \\ Pid_H &\ \widehat{=}\ \{p \mid p \in Pid \wedge clear(p) = \mathbf{hi}\} \end{aligned} \tag{5.5}$$

The two interfaces to the system are given by

$$\begin{aligned} L \quad \widehat{=} \quad \{\!| \ & fopen.f.p, flock.f.p, fread.f.p, fwrite.f.p, \\ & \quad fclose.f.p, funlock.f.p \mid f \in Fid, p \in Pid_L \ |\!\} \end{aligned} \tag{5.6}$$

$$\begin{aligned} H \quad \widehat{=} \quad \{\!| \ & fopen.f.p, flock.f.p, fread.f.p, fwrite.f.p, \\ & \quad fclose.f.p, funlock.f.p \mid f \in Fid, p \in Pid_H \ |\!\} \ . \end{aligned} \tag{5.7}$$

The overall security requirement for the system is for the L interface to be unaffected by the concurrent activity in the H interface.

5.2 Compositional Development

The system will be defined as a parallel composition of secure components, which are introduced first.

Components

The requirements for each component will be discussed in turn.

Opening and Reading Files

Firstly, we wish to describe how the read access to a file relates to its opening and closing.

In order to read file f, a process p first has to open the file. Similarly, f cannot be closed without being previously opened. Reading can take place only while the file is open. Once f has been closed, it cannot be read by p until the file is re-opened by p.

$$0 \leq tr \downarrow \{\!|\ fopen.f.p\ |\!\} - tr \downarrow \{\!|\ fclose.f.p\ |\!\} \leq 1 \tag{5.8}$$

$$\wedge$$

$$tr = t \frown \langle fread.f.p.v \rangle \ \Rightarrow$$
$$\underline{\mathsf{last}}\,(t \upharpoonright \{\!|\ fopen.f.p, fclose.f.p\ |\!\}) \in \{\!|\ fopen.f.p\ |\!\}\ . \tag{5.9}$$

This requirement is implemented with process Rd below.

$$\begin{array}{rcl} \alpha Rd & \widehat{=} & \{\!|\ fopen, fclose, fread\ |\!\} \\ Rd & \widehat{=} & |||_{f:Fid}\, Rd'(f) \\ Rd'(f) & \widehat{=} & |||_{p:Pid}\, Rd1(f,p) \\ Rd1(f,p) & \widehat{=} & fopen.f.p?c \rightarrow Rd2(f,p) \\ Rd2(f,p) & \widehat{=} & (fclose.f.p \rightarrow Rd1(f,p)) \\ & & \Box\ (fread.f.p?v \rightarrow Rd2(f,p))\ . \end{array}$$

Interface L is independent in process Rd.

Lemma 5.2.1 $H \not\longrightarrow L :: Rd$.

Locking and Writing Files

A file f can be written by a process p only after being locked by p. No more writes can take place after unlocking the file, but it may be locked again later on.

The requirement for prior locking is captured by the predicate

$$0 \leq tr \downarrow \{\!|\ \mathit{flock}.f.p\ |\!\} - tr \downarrow \{\!|\ \mathit{funlock}.f.p\ |\!\} \leq 1 \tag{5.10}$$

$$\wedge$$

$$tr = t \frown \langle \mathit{fwrite}.f.p.v \rangle \ \Rightarrow$$

$$\underline{\text{last}}\,(t \upharpoonright \{\!|\ \mathit{flock}.f.p, \mathit{funlock}.f.p\ |\!\}) \in \{\!|\ \mathit{flock}.f.p\ |\!\} \tag{5.11}$$

and implemented with process Wr below.

$$\begin{array}{rcl}
\alpha Wr & \widehat{=} & \{\!|\ \mathit{flock}, \mathit{funlock}, \mathit{fwrite}\ |\!\} \\
Wr & \widehat{=} & |||_{f:Fid}\ Wr'(f) \\
Wr'(f) & \widehat{=} & |||_{p:Pid}\ Wr1(f,p) \\
Wr1(f,p) & \widehat{=} & \mathit{flock}.f.p?c \rightarrow Wr2(f,p) \\
Wr2(f,p) & \widehat{=} & (\mathit{funlock}.f.p \rightarrow Wr1(f,p)) \\
 & & \Box\ (\mathit{fwrite}.f.p?v \rightarrow Wr2(f,p))\ .
\end{array}$$

Interface L is independent in process Wr.

Lemma 5.2.2 $H \not\rightarrowtail L :: Wr$.

The BLP Properties

In the following we specify what we call the BLP property of the filesystem.[1] By this we mean the following:

- an attempt at writing file f by process p succeeds only if p has sufficiently low clearance. More precisely, process p can write, lock, or unlock f only if $clear(p) \leq class(f)$. (The *-property.)
- an attempt at reading file f by process p succeeds only if p has sufficiently high clearance. More precisely, process p can read, open, or close f only if $class(f) \leq clear(p)$. (The ss-property.)

[1] The Bell-LaPadula model is an early formalization of a military security policy. The model includes a number of access rules under which subjects (such as processes) may access objects (such as files), the most important of which are the simple-security and $*$-properties. We do not regard these properties as part of a security policy but as useful design principles which help to establish the system's overall independence requirements.

These requirements are captured by the predicates

$$
\begin{aligned}
& fwrite.f.p.v \ \underline{\mathsf{in}}\ tr \vee flock.f.p.c \ \underline{\mathsf{in}}\ tr \vee funlock.f.p \ \underline{\mathsf{in}}\ tr \\
& \qquad \Rightarrow \ class(f) \geq clear(p), \qquad (5.12) \\
& fread.f.p.v \ \underline{\mathsf{in}}\ tr \vee fopen.f.p.c \ \underline{\mathsf{in}}\ tr \vee fclose.f.p \ \underline{\mathsf{in}}\ tr \\
& \qquad \Rightarrow \ class(f) \leq clear(p) \qquad (5.13)
\end{aligned}
$$

and implemented as process BLP.

$$
\begin{aligned}
\alpha BLP &\mathrel{\widehat=} \alpha BLP\_star \cup \alpha BLP\_ss \\
BLP &\mathrel{\widehat=} BLP\_star \ \Box\ BLP\_ss
\end{aligned}
$$

The $*$-property is implemented by

$$
\begin{aligned}
\alpha BLP\_star &\mathrel{\widehat=} \{\!|\ fwrite, flock, funlock\ |\!\} \\
BLP\_star &\mathrel{\widehat=} (\Box_{p:Pid} \Box_{f:Fid} \\
&\quad \underline{\mathsf{if}}\ class(f) < clear(p)\ \underline{\mathsf{then}}\ Stop \\
&\quad \underline{\mathsf{else}}\ ((fwrite.f.p?v \rightarrow BLP) \\
&\qquad \Box\ (flock.f.p?c \rightarrow BLP) \\
&\qquad \Box\ (funlock.f.p \rightarrow BLP)))\ ,
\end{aligned}
$$

while the simple-security property is implemented by

$$
\begin{aligned}
\alpha BLP\_ss &\mathrel{\widehat=} \{\!|\ fread, fopen, fclose\ |\!\} \\
BLP\_ss &\mathrel{\widehat=} (\Box_{p:Pid} \Box_{f:Fid} \\
&\quad \underline{\mathsf{if}}\ clear(p) < class(f)\ \underline{\mathsf{then}}\ Stop \\
&\quad \underline{\mathsf{else}}\ ((fread.f.p?v \rightarrow BLP) \\
&\qquad \Box\ (fopen.f.p?c \rightarrow BLP) \\
&\qquad \Box\ (fclose.f.p \rightarrow BLP)))\ .
\end{aligned}
$$

Interface L is independent in process BLP.

Lemma 5.2.3 $H \not\rightarrowtail L :: BLP$.

Clearance

Processes have to declare their correct security clearances when opening or locking a file:

$$fopen.f.p.c \ \underline{\mathsf{in}}\ tr \vee flock.f.p.c \ \underline{\mathsf{in}}\ tr \quad \Rightarrow \quad clear(p) = c. \qquad (5.14)$$

This is implemented with process $Clear$.

$$
\begin{aligned}
\alpha Clear &\mathrel{\widehat{=}} \{\!|\ fopen, flock\ |\!\} \\
Clear &\mathrel{\widehat{=}} (\Box_{p:Pid}\, fopen?f.p.clear(p) \rightarrow Clear) \\
&\quad \Box\ (\Box_{p:Pid}\, flock?f.p.clear(p) \rightarrow Clear).
\end{aligned}
$$

Interface L is independent in process $Clear$.

Lemma 5.2.4 $H \not\longrightarrow L :: Clear$.

Retrospective Values

Finally, we address the question of how values written to each file are related to the values retrieved upon reading. If there have been no writes to file f, then the initial value v_0 is returned when the file is read. If a process writes value v to file f, and there are no intervening writes, then v is the value obtained when f is read next.

$$
\begin{gathered}
tr = s \frown \langle fread.f.p.v \rangle \quad \Rightarrow \\
((s \upharpoonright \{\!|\ fwrite.f\ |\!\} = \langle\rangle \wedge v = v_0) \vee (s \upharpoonright \{\!|\ fwrite.f\ |\!\} \neq \langle\rangle \wedge \\
\exists\, p' \in Pid \bullet \underline{\mathsf{last}}\,(s \upharpoonright \{\!|\ fwrite.f\ |\!\}) = fwrite.f.p'.v))\ . \qquad (5.15)
\end{gathered}
$$

Process $Retro$ satisfies this requirement.

$$
\begin{aligned}
\alpha Retro &\mathrel{\widehat{=}} \{\!|\ fread, fwrite\ |\!\} \\
Retro &\mathrel{\widehat{=}} |||_{f:Fid}\, Retro1(f, v_0) \\
Retro1(f, v) &\mathrel{\widehat{=}} (fread.f?p!v \rightarrow Retro1(f, v)) \\
&\quad \Box\ (fwrite.f?p?nv \rightarrow Retro1(f, nv)).
\end{aligned}
$$

Unfortunately, process $Retro$ is not secure because $fwrite$ events always change the file value, irrespective of the security levels of the writing process and the file being written. Therefore, information flows from the high to the low interface if a high process writes a file which is subsequently read by a low process. The information flow is apparent by considering why the abstract process $\mathcal{I}_H(Retro)$ (or $\mathcal{L}_H(Retro)$) cannot be deterministic.

Proposition 5.2.5 Interface L is not independent in $Retro$. The abstract process $\mathcal{I}_H(Retro)$ is non-deterministic, since (for example)

$$\begin{array}{rcl} \langle fwrite.fh.ph.1, fread.fh.pl.0\rangle & \in & \text{TRACES}(\mathcal{I}_H(Retro)) \\ \wedge \;\; (\langle fwrite.fh.ph.1\rangle, \{fread.fh.pl.0\}) & \in & \text{FAILS}(\mathcal{I}_H(Retro)) \; , \end{array}$$

where it is assumed that $fh \in Fid_H$, $ph \in Pid_H$, and $pl \in Pid_L$. The event $fread.fh.pl.0$ is accepted after $\langle fwrite.fh.ph.1\rangle$ because event $fwrite.fh.ph.1$ is contributed by Run_H thus leaving the initial file value 0 intact. However, if it is $Retro$ which contributes event $fwrite.fh.ph.1$ then the initial value is overwritten, causing the refusal of $fread.fh.pl.0$ above.

There are two possible approaches to overcoming the insecurity of process $Retro$. The first is to review the components already defined, in order to find one component whose composition with $Retro$ is secure. Since the system will later be defined as a parallel composition, this other component will ensure that the insecurity of $Retro$ cannot propagate to the rest of the system. In this case, we find that BLP removes the insecurities of $Retro$, so that

$$H \not\longrightarrow L :: BLP \, [\alpha BLP \| \alpha Retro] \, Retro \tag{5.16}$$

because BLP prohibits "write-downs" and "read-ups" on which the undesired information flow relied.

If it were not possible to find a component that removes the insecurities, it would be necessary to modify the component in order to make it secure. For instance, process $Retro$ could be secured by separating the file contents depending on the security label of the process accessing the file. We introduce two process parameters, v_L and v_H, which are being read by the low and high processes, respectively. When a high process writes to the file, only v_H becomes updated, whereas a write by a low process will affect both values.

$$\begin{array}{rcl} Retro & = & |||_{f:Fid} \, Retro1(f, v_0, v_0) \\ Retro1(f, v_L, v_H) & = & (\Box_{p:Pid_L} \, fread.f.p!v_L \rightarrow Retro1(f, v_L, v_H)) \\ & & \Box \, (\Box_{p:Pid_H} \, fread.f.p!v_H \rightarrow Retro1(f, v_L, v_H)) \\ & & \Box \, (fwrite.f?p?nv \rightarrow \\ & & \quad \underline{\text{if}} \; clear(p) = \mathbf{hi} \; \underline{\text{then}} \; Retro1(f, v_L, nv) \\ & & \quad \underline{\text{else}} \; \underline{\text{if}} \; clear(p) = \mathbf{lo} \; \underline{\text{then}} \; Retro1(f, nv, nv) \\ & & \quad \underline{\text{else}} \; Stop). \end{array}$$

Interface L is independent in this version of Retro.

Lemma 5.2.6 $H \not\rightarrowtail L :: \mathit{Retro}$.

Exclusive Access

File access is exclusive, in the sense that a file f can be either open or locked, but not both simultaneously. User process p can lock f only if it is neither locked nor open.

$$\begin{array}{ll} & tr = s \frown \langle \mathit{flock}.f.p \rangle \\ \Rightarrow & s \downarrow \{\!|\ \mathit{flock}.f\ |\!\} = s \downarrow \{\!|\ \mathit{funlock}.f\ |\!\} \wedge \\ & s \downarrow \{\!|\ \mathit{fopen}.f\ |\!\} = s \downarrow \{\!|\ \mathit{fclose}.f\ |\!\}\ . \end{array} \tag{5.17}$$

File f may be unlocked only by the process holding the lock.

$$\begin{array}{ll} & tr = s \frown \langle \mathit{funlock}.f.p \rangle \\ \Rightarrow & \underline{\mathsf{last}}\,(s \restriction \{\!|\ \mathit{flock}.f, \mathit{funlock}.f\ |\!\}) = \mathit{flock}.f.p\ . \end{array} \tag{5.18}$$

The requirements for opening a file are somewhat more liberal than for locking, in that several user processes may have f open at the same time.

$$\begin{array}{ll} & tr = s \frown \langle \mathit{fopen}.f.p \rangle \\ \Rightarrow & s \downarrow \{\!|\ \mathit{flock}.f\ |\!\} = s \downarrow \{\!|\ \mathit{funlock}.f\ |\!\} \wedge \\ & s \downarrow \{\!|\ \mathit{fopen}.f.p\ |\!\} = s \downarrow \{\!|\ \mathit{fclose}.f.p\ |\!\}\ . \end{array} \tag{5.19}$$

Closing of file f can be carried out only by processes which have previously opened f.

$$\begin{array}{ll} & tr = s \frown \langle \mathit{fclose}.f.p \rangle \\ \Rightarrow & \underline{\mathsf{last}}\,(s \restriction \{\!|\ \mathit{fopen}.f.p, \mathit{fclose}.f.p\ |\!\}) = \mathit{fopen}.f.p\ . \end{array} \tag{5.20}$$

These requirements are implemented with process XFiles. The parameter f refers to the identifier of the file concerned, and us is the set of user processes which have successfully opened f.

$$\begin{array}{rcl} \alpha \mathit{XFiles} & \widehat{=} & \{\!|\ \mathit{flock}, \mathit{funlock}, \mathit{fopen}, \mathit{fclose}\ |\!\} \\ \mathit{XFiles} & \widehat{=} & |||_{f:\mathit{Fid}}\ \mathit{XFile}(f) \end{array}$$

$$
\begin{array}{rcl}
XFile(f) & \mathrel{\widehat=} & (fopen.f?p?c \rightarrow XOpen(f, \{p\})) \\
 & & \Box\ (flock.f?p?c \rightarrow XLocked(f, p)) \\
XOpen(f, us) & \mathrel{\widehat=} & (\Box_{p:Pid \setminus us}\, fopen.f.p?c \rightarrow XOpen(f, us \cup \{p\})) \\
 & & \Box\ (\Box_{p:us}\, fclose.f.p \rightarrow \\
 & & \qquad \underline{\text{if}}\ \underline{\text{card}}(us)\ = 1 \\
 & & \qquad \underline{\text{then}}\ XFile(f) \\
 & & \qquad \underline{\text{else}}\ XOpen(f, us \setminus \{p\})) \\
XLocked(f, p) & \mathrel{\widehat=} & (funlock.f.p \rightarrow XFile(f))\ .
\end{array}
$$

Unfortunately, interface L fails to be independent in process $XFiles$, because the progress of a low process may be blocked depending on the prior activity of a high process. When a high process successfully locks file fl then no low process is able to open fl. Furthermore, if a high process opens fl then no low process can lock the file.

Proposition 5.2.7 Interface L fails to be independent in $XFiles$. The abstract process $\mathcal{I}_H(XFiles)$ is non-deterministic, since for example

$$
\begin{array}{rcl}
\langle fopen.fl.ph.\mathbf{hi}, flock.fl.pl.\mathbf{lo}\rangle & \in & \textsc{Traces}(\mathcal{I}_H(XFiles)) \\
\wedge\ \ (\langle fopen.fl.ph.\mathbf{hi}\rangle, \{flock.fl.pl.\mathbf{lo}\}) & \in & \textsc{Fails}(\mathcal{I}_H(XFiles))\ ,
\end{array}
$$

where it is assumed that $fl \in Fid_L$, $ph \in Pid_H$, and $pl \in Pid_L$. Event $flock.fl.pl.\mathbf{lo}$ is accepted only if the initial event of the trace was contributed by Run_H. However, if a high process opens the file fl with event $fopen.fl.ph.\mathbf{hi}$, then process pl is unable to lock, resulting in the failure above.

Exclusive Access, Second Attempt

For process $XFiles$, it will be necessary to ensure that low processes can still gain access to a file even in the case when a high process has locked or opened the file.

$$
\begin{array}{rcl}
XFiles & \mathrel{\widehat=} & |||_{f:Fid}\, XFile(f) \\
XFile(f) & \mathrel{\widehat=} & (fopen.f?p?c \rightarrow XOpen(f, \{p\})) \\
 & & \Box\ (flock.f?p?c \rightarrow XLocked(f, p)).
\end{array}
$$

Again, process $XOpen$ deals with the case of f being opened. The parameter us of $XOpen$ refers to the set of processes which have opened f. If this set does not include any low processes, that is $(us \setminus Pid_H) = \emptyset$, then any such process $pl : Pid_L$ which attempts to lock f should not be prevented from doing so.

$$
\begin{aligned}
&XOpen(f, us) \mathrel{\widehat{=}} \\
&\quad (\Box_{p:Pid \setminus us} fopen.f.p?c \rightarrow XOpen(f, us \cup \{p\})) \\
&\quad \Box\ (\Box_{p:us} fclose.f.p \rightarrow \\
&\qquad \underline{\mathsf{if}}\ \underline{\mathsf{card}}(us) = 1\ \underline{\mathsf{then}}\ XFile(f) \\
&\qquad \underline{\mathsf{else}}\ XOpen(f, us \setminus \{p\})) \\
&\quad \Box\ (\underline{\mathsf{if}}\ (us \setminus Pid_H) = \emptyset \\
&\qquad \underline{\mathsf{then}}\ (\Box_{pl:Pid_L} flock.f.pl?c \rightarrow XOpen1(f, us, pl)) \\
&\qquad \underline{\mathsf{else}}\ Stop)
\end{aligned}
$$

Process $XOpen1$ refers to the state when low process pl has obtained the lock on f by engaging in event $flock.f.pl?c$ while processes us still have that file open. Although no other processes may lock the file in this state, the high processes are permitted to close the file they had previously opened for reading.

$$
\begin{aligned}
&XOpen1(f, us, pl) \mathrel{\widehat{=}} \\
&\quad (funlock.f.pl \rightarrow XOpen(f, us)) \\
&\quad \Box\ (\Box_{p:us} fclose.f.p \rightarrow \\
&\qquad \underline{\mathsf{if}}\ \underline{\mathsf{card}}(us) = 1 \\
&\qquad \underline{\mathsf{then}}\ XLocked(f, pl) \\
&\qquad \underline{\mathsf{else}}\ XOpen1(f, us \setminus \{p\}, pl))\ .
\end{aligned}
$$

Process $XLocked$ deals with the case of file f being locked by p. If p is a high process, then no low processes should be prohibited from opening or locking the file.

$$
\begin{aligned}
&XLocked(f, p) \mathrel{\widehat{=}} \\
&\quad (funlock.f.p \rightarrow XFile(f)) \\
&\quad \Box\ (\underline{\mathsf{if}}\ p \in Pid_H \\
&\qquad \underline{\mathsf{then}}\ ((\Box_{pl:Pid_L} flock.f.pl?c \rightarrow XLocked1(f, p, pl)) \\
&\qquad\qquad \Box\ (\Box_{pl:Pid_L} fopen.f.pl?c \rightarrow XLocked2(f, p, \{pl\}))) \\
&\qquad \underline{\mathsf{else}}\ Stop)
\end{aligned}
$$

The case where a low process pl acquires the lock after the high process p is described by $XLocked1$. The case where one or more low processes, identified by parameter us, open f after the high process p has locked that file, is described by $XLocked2$.

$$
\begin{array}{rcl}
XLocked1(f,p,pl) & \widehat{=} & (funlock.f.p \rightarrow XLocked(f,pl)) \\
 & & \Box\ (funlock.f.pl \rightarrow XLocked(f,p)) \\
XLocked2(f,p,us) & \widehat{=} & (\Box_{pl:Pid_L}\, fopen.f.pl?c \rightarrow \\
 & & \qquad XLocked2(f,p,us \cup \{pl\})) \\
 & & \Box\ (\Box_{pl:us}\, fclose.f.pl \rightarrow \\
 & & \qquad \underline{\text{if}}\ \underline{\text{card}}(us)\ = 1 \\
 & & \qquad \underline{\text{then}}\ XLocked(f,p) \\
 & & \qquad \underline{\text{else}}\ XLocked2(f,p,us \setminus \{pl\})) \\
 & & \Box\ (funlock.f.p \rightarrow XOpen(f,us)).
\end{array}
$$

As low processes are never blocked by the opening and locking of files by high processes, interface L is now independent in $XFiles$.

Lemma 5.2.8 $H \not\longrightarrow L :: XFiles$.

System Definition

Our model of readers and writers is a simple parallel composition of the processes described above.

Definition 5.2.9 The system is given by process $System$ below.

$$
\begin{array}{rcl}
System_0 & \widehat{=} & ((BLP\ [\Sigma \| \alpha Rd]\ Rd)\ [\Sigma \| \alpha Wr]\ Wr)\ [\Sigma \| \alpha Clear]\ Clear\ , \\
System & \widehat{=} & (System_0\ [\Sigma \| \alpha Retro]\ Retro)\ [\Sigma \| \alpha XFiles]\ XFiles\ .
\end{array}
$$

The appendix to this chapter contains a detailed version of this process which is provably equivalent to $System$. Having defined it by composition of secure parts, we can conclude that the complete system is also secure.

Theorem 5.2.10 $H \not\longrightarrow L :: System$.

Proof From Lemma 5.2.1, Lemma 5.2.2, Lemma 5.2.3, Lemma 5.2.4, Lemma 5.2.6, and Lemma 5.2.8 by repeated application of Theorem 4.1.8. □

5.3 Discussion

In this chapter, we have studied and presented a solution to the secure readers/writers problem. This problem is interesting because of its inherent conflict between functional requirements (exclusive access to files) and security restrictions (non-interference between processes). Since it does not appear to be possible to entirely resolve this conflict adequately, any concrete implementation must give preference to either functional correctness or security. We have chosen a provably secure implementation, by defining the system as a parallel composition of secure components.

Individual requirements of the system have been described with predicates on traces, often describing the "firing conditions" under which individual events can take place. Each predicate has been implemented using a relatively simple CSP process, whose functional and security properties are often straightforward to verify. The parallel composition of process components corresponds to the conjunction of the predicate(s) they implement; the benefits of this compositional style of development are well-documented by Woodcock [Woo90].

Inevitably there are components which meet the functional requirements of the system but are not secure, such as processes $Retro$ and $XFiles$. The insecurity of $Retro$ concerning file values was not of particular concern, because it did not propagate in combination with BLP, implementing the familiar "no read-up" and "no write-down" principles. The insecurity of $XFiles$, however, could not be resolved that way – one could say that the fundamental conflict of the readers/writers problem has been encapsulated in this process. Giving security the ultimate priority in this case, we have resolved to adapt the functional properties of the process in order to exploit the compositional nature of independence in the proof of security of the complete system.

The price we pay for this approach is the compromise of functional correctness, because processes with differing security clearances may access the same file concurrently. This case can occur for instance when a low process obtains a lock in order to write to a high file while a high process has previously opened the same file for reading. This possibility of inconsistent views of data cannot be avoided without restricting processes to access resources having the same security level only.

Several plausible variations can be introduced to our model of readers and writers, some of which can easily be achieved by changing the individual components of the system. Our model has the property that low processes can change the contents of high-level files – even though those processes cannot themselves read any data stored in high files. If this is of concern, then the following modification of process $Retro1$ will ensure that low write operations to high files will remain without effect.

$$
\begin{array}{lcl}
Retro1(f, v_L, v_H) & \widehat{=} & (\Box_{p:Pid_L}\, fread.f.p!v_L \rightarrow Retro1(f, v_L, v_H)) \\
& & \Box\ (\Box_{p:Pid_H}\, fread.f.p!v_H \rightarrow Retro1(f, v_L, v_H)) \\
& & \Box\ (fwrite.f?p?nv \rightarrow \\
& & \quad \underline{\text{if}}\ clear(p) = \mathbf{hi}\ \underline{\text{then}}\ Retro1(f, v_L, nv) \\
& & \quad \underline{\text{else}}\ \underline{\text{if}}\ clear(p) = \mathbf{lo} \wedge class(f) = \mathbf{lo} \\
& & \qquad \underline{\text{then}}\ Retro1(f, nv, nv) \\
& & \quad \underline{\text{else}}\ \underline{\text{if}}\ clear(p) = \mathbf{lo} \wedge class(f) = \mathbf{hi} \\
& & \qquad \underline{\text{then}}\ Retro1(f, nv, v_H) \\
& & \quad \underline{\text{else}}\ Stop).
\end{array}
$$

Even with the modification above, it is possible for a high process to read "inconsistent" values from a file. This situation occurs when a high process p opens low file f for reading. While p is reading, another low process p' may validly lock f and change the file value.

One approach to overcome this potential problem would be for the file to keep previous values rather than overwriting them, and discarding old values only when p has finished reading. Another approach, suggested in [Gr91], would be to send a special message to p to alert this process that it may be reading inconsistent values.

5.4 Appendix: The Complete System

In this section, we give an alternative version of process $System$ for the benefit of the reader who wishes to see a sequential description of the functionality of each file, rather than the parallel combination of individual components presented previously. This version, given as process $Files$ below, is equivalent to the previous implementation in the sense that

$$System \quad =_{FD} \quad Files\ , \tag{5.21}$$

which has been verified with the FDR model checker.

$$\begin{aligned} \alpha Files \quad &\widehat{=} \quad \{\!|\ flock, funlock, fopen, fclose, fread, fwrite\ |\!\} \\ Files \quad &\widehat{=} \quad |||_{fl:Fid}\, File(fl, v_0)\ . \end{aligned}$$

The structure of each $File$ process closely matches that of the modified $XFiles$ process. Initially, file f with contents v may be opened or locked.

$$\begin{aligned} & File(f, v)\ \widehat{=} \\ & \quad (\Box_{p:Pid}\ \underline{\mathsf{if}}\ clear(p) < class(f)\ \underline{\mathsf{then}}\ Stop \\ & \qquad \underline{\mathsf{else}}\ (fopen.f.p.clear(p) \rightarrow Open(f, v, p))) \\ & \quad \Box\ (\Box_{p:Pid}\ \underline{\mathsf{if}}\ class(f) < clear(p)\ \underline{\mathsf{then}}\ Stop \\ & \qquad \underline{\mathsf{else}}\ (flock.f.p.clear(p) \rightarrow Locked(f, v, p))) \end{aligned}$$

Process $Open$ represents the file state when file f has been opened for reading. If only high processes have opened f then low processes are still permitted to lock the file ($Open0$).

$$\begin{aligned} & Open(f, v, us)\ \widehat{=} \\ & \quad (\Box_{p:us}\, fread.f.p!v \rightarrow Open(f, v, us)) \\ & \quad \Box\ (\Box_{p:Pid \setminus us} \\ & \qquad \underline{\mathsf{if}}\ clear(p) < class(f)\ \underline{\mathsf{then}}\ Stop \\ & \qquad \underline{\mathsf{else}}\ (fopen.f.p.clear(p) \rightarrow Open(f, v, us \cup \{p\}))) \\ & \quad \Box\ (\Box_{p:us}(fclose.f.p \rightarrow \\ & \qquad \underline{\mathsf{if}}\ \underline{\mathsf{card}}(us) = 1\ \underline{\mathsf{then}}\ File(f, v) \\ & \qquad \underline{\mathsf{else}}\ Open(f, v, us \setminus \{p\}))) \end{aligned}$$

$$\begin{array}{l}
\Box\ (\underline{\text{if}}\ (us \setminus Pid_H) = \emptyset \\
\quad \underline{\text{then}}\ (\Box_{p:Pid_L}\ \underline{\text{if}}\ class(f) < clear(p)\ \underline{\text{then}}\ Stop \\
\quad\quad \underline{\text{else}}\ (flock.f.p.clear(p) \rightarrow Open0(f, v, us, p))) \\
\quad \underline{\text{else}}\ Stop)
\end{array}$$

$$\begin{array}{l}
Open0(f, v, us, p)\ \widehat{=} \\
\quad (fwrite.f.p?v \rightarrow Open0(f, v, us, p)) \\
\quad \Box\ (funlock.f.p \rightarrow Open(f, v, us)) \\
\quad \Box\ (\Box_{ph:us}\ fclose.f.ph \rightarrow \\
\quad\quad \underline{\text{if}}\ \underline{\text{card}}(us) = 1\ \underline{\text{then}}\ Locked(f, v, p) \\
\quad\quad \underline{\text{else}}\ Open0(f, v, us \setminus \{ph\}, p)) \\
\quad \Box\ (\Box_{ph:us}\ fread.f.ph!v \rightarrow Open0(f, v, us, p)).
\end{array}$$

Process $Locked$ represents the state of the file f when it has been locked for writing. When the lock was acquired by a high process, then low processes can still lock f ($Locked1$) and open f ($Locked2$).

$$\begin{array}{l}
Locked(f, v, p)\ \widehat{=} \\
\quad (fwrite.f.p?v \rightarrow Locked(f, v, p)) \\
\quad \Box\ (funlock.f.p \rightarrow File(f, v)) \\
\quad \Box\ (\underline{\text{if}}\ p \in Pid_H \\
\quad\quad \underline{\text{then}}\ ((\Box_{pl:Pid_L} \\
\quad\quad\quad \underline{\text{if}}\ class(f) < clear(p)\ \underline{\text{then}}\ Stop \\
\quad\quad\quad \underline{\text{else}}\ (flock.f.pl.clear(pl) \rightarrow Locked1(f, v, p, pl))) \\
\quad\quad\quad \Box\ (\Box_{pl:Pid_L} \\
\quad\quad\quad \underline{\text{if}}\ clear(pl) < class(f)\ \underline{\text{then}}\ Stop \\
\quad\quad\quad \underline{\text{else}}\ (fopen.f.p.clear(pl) \rightarrow Locked2(f, v, p, pl)))) \\
\quad\quad \underline{\text{else}}\ Stop)\ .
\end{array}$$

$$\begin{array}{l}
Locked1(f, v, p, pl)\ \widehat{=} \\
\quad (fwrite.f.p?nv \rightarrow Locked1(f, nv, p, pl)) \\
\quad \Box\ (fwrite.f.pl?nv \rightarrow Locked1(f, nv, p, pl)) \\
\quad \Box\ (funlock.f.p \rightarrow Locked(f, v, pl)) \\
\quad \Box\ (funlock.f.pl \rightarrow Locked(f, v, p))\ .
\end{array}$$

$$
\begin{array}{l}
Locked2(f, v, p, us) \;\widehat{=} \\
\quad (\Box_{pl:us} fread.f.pl!v \rightarrow Locked2(f, v, p, us)) \\
\quad \Box\ (\Box_{pl:Pid_L \setminus us} \\
\qquad \underline{\mathsf{if}}\ clear(pl) < class(f)\ \underline{\mathsf{then}}\ Stop \\
\qquad \underline{\mathsf{else}}\ (fopen.f.pl?c \rightarrow Locked2(f, v, p, us \cup \{pl\}))) \\
\quad \Box\ (\Box_{pl:us} fclose.f.pl \rightarrow \underline{\mathsf{if}}\ \underline{\mathsf{card}}(us) \; = 1 \\
\qquad\qquad \underline{\mathsf{then}}\ Locked(f, v, p) \\
\qquad\qquad \underline{\mathsf{else}}\ Locked2(f, v, p, us \setminus \{pl\})) \\
\quad \Box\ (fwrite.f.p?nv \rightarrow Locked2(f, nv, p, us)) \\
\quad \Box\ (funlock.f.p \rightarrow Open(f, v, us)) \; .
\end{array}
$$

Chapter 6

A Multi-level Filesystem

In this chapter, we present a model of a file system with multi-level security requirements. The model covers all standard operations of shared storage systems: the creation and removal of files, the reading and writing of file values, and the renaming of files. The development is largely compositional; the conjunction of individual requirements is realised by parallel composition of the corresponding CSP processes. We develop the low-level properties of files and buckets (collection of files) first, before making this functionality available at the system interface by exploiting the renaming operator of CSP. The security policy for the file system will be stated, and a formal proof that the implementation satisfies this policy completes the chapter.

6.1 Main Definitions

The filesystem is assumed to be used by a number of external users, whose identifiers are given by the set $User$. File names can be chosen from the set $Fname$, and files created will be stored in buckets with names from set $Bucket$. Files may contain values of type Val. The given sets are thus:

$$[\, User, Bucket, Fname, Val, Label \,]$$

The set $Label$ describes the set of security labels, which is partially ordered by $\leq$. Every user (from set $User$) is assigned a security label by the total function $clear$, and every bucket (from set $Bucket$) is assigned a label by the total function $class$.

$$\begin{aligned} class &:: Bucket \twoheadrightarrow Label \\ clear &:: User \rightarrowtail\!\!\!\!\twoheadrightarrow Label \end{aligned} \tag{6.1}$$

Function $class$ is bijective establishing a one-to-one correspondence between $Bucket$ and $Label$; there is exactly one bucket per security level (and vice versa). Function $clear$ is surjective so that there is at least one user on a particular security level, but there may be several users having the same clearance.

File identifiers are given by the cross product of types $Bucket$ and $Fname$, so that different buckets may contain files of the same name.

$$Fid \quad ::= \quad Bucket.Fname$$

Files can be distinguished by reference to both its name and the bucket which contains the file. For instance, file a in bucket $B2$ can be identified by reference to "$B2.a$".

Security Policy

The security policy which the filesystem is required to satisfy is defined in terms of the relationship between the security labels attached to the external users, which constitute the interface of the system. User u may interfere with user v if, and only if, the security label associated with v dominates that of u. Let $X(u)$ denote the events under control of user u; this set is defined with Equation 6.15 when the external system channels have been introduced.

Definition 6.1.1 The security policy for the filesystem FS is given by the relation $\rightsquigarrow$ defined

$$X(u) \rightsquigarrow X(v) \quad \Leftrightarrow \quad clear(u) \leq clear(v) \ .$$

6.2 Compositional Development

The development starts with a description of the properties of individual files, which are then combined with each other and grouped into "buckets". This low-level description will then be lifted to the level of the external system interface, where further constraints will be added.

File model

Files are created and removed through channels fcr and frm, respectively. Reading and writing of files takes place using channels frd and fwr, respectively. The "file move" channel fmv allows a file name to be changed; the event $fmv.B1.f.B2.g$ moves file f from bucket $B1$ to $B2$ with new name g, without changing the contents of the file.

$$\begin{array}{lll} \underline{\text{channel}} & fcr, frm & : Fid \\ \underline{\text{channel}} & frd, fwr & : Fid.Val \\ \underline{\text{channel}} & fmv & : Fid.Fid \end{array}$$

We consider first the properties of a single file in isolation. File creation and removal must alternate, starting with creation through channel fcr

$$0 \leq tr \downarrow \{\!|\ fcr\ |\!\} - tr \downarrow \{\!|\ frm\ |\!\} \leq 1 \ , \tag{6.2}$$

where there is no reference to the file identifier, because this may change when the file is renamed. The reading and writing of values as well as file moves can occur only while the file is active, that is between a fcr and a frm event.

$$\begin{array}{ll} & tr = s \frown \langle a \rangle \wedge a \in \{\!|\ frd, fwr, fmv\ |\!\} \\ \Rightarrow & s \downarrow \{\!|\ frm\ |\!\} + 1 = s \downarrow \{\!|\ fcr\ |\!\} \ . \end{array} \tag{6.3}$$

These requirements are implemented by

$$\begin{array}{lll} C1File_0 & \widehat{=} & fcr?f \to C1File_1 \\ C1File_1 & \widehat{=} & (frm?f \to C1File_0) \\ & & \Box\ (frd?f?v \to C1File_1) \\ & & \Box\ (fwr?f?v \to C1File_1) \\ & & \Box\ (fmv?f?f' \to C1File_1) \ . \end{array}$$

The file identifier is established upon creation, or when the file is moved or renamed through channel fmv. Whenever the file is read, written, moved, or removed, the reference is to the identifier established either with the last fcr event or with the last fmv event, if there has been one.

$$\begin{array}{ll} & tr = s \frown \langle a \rangle \wedge a \in \{\!| \ frd, fwr, frm, fmv \ |\!\} \wedge \\ & \underline{\mathsf{last}}\,(s \upharpoonright \{\!| \ fcr, fmv \ |\!\}) = fcr.f \\ \Rightarrow & fid(a) = f \ , \end{array} \quad (6.4)$$

$$\begin{array}{ll} & tr = s \frown \langle a \rangle \wedge a \in \{\!| \ frd, fwr, frm, fmv \ |\!\} \wedge \\ & \underline{\mathsf{last}}\,(s \upharpoonright \{\!| \ fcr, fmv \ |\!\}) = fmv.f.f' \\ \Rightarrow & fid(a) = f' \ , \end{array} \quad (6.5)$$

where the function fid is assumed to return the file identifier (i.e. the component of type Fid after the channel name). The requirements concerning file identifiers are implemented by

$$\begin{array}{rcl} C2File_0 & \widehat{=} & (frm?f \rightarrow C2File_0) \\ & & \square\ (frd?f?v \rightarrow C2File_0) \\ & & \square\ (fwr?f?v \rightarrow C2File_0) \\ & & \square\ (fmv?f?f' \rightarrow C2File_1(f')) \\ & & \square\ (fcr?f \rightarrow C2File_1(f)) \\ C2File_1(f) & \widehat{=} & (frm.f \rightarrow C2File_1(f)) \\ & & \square\ (frd.f?v \rightarrow C2File_1(f)) \\ & & \square\ (fwr.f?v \rightarrow C2File_1(f)) \\ & & \square\ (fmv.f?f' \rightarrow C2File_1(f')) \\ & & \square\ (fcr?f' \rightarrow C2File_1(f')) \ . \end{array}$$

Whenever a file is newly created, a value must be written to it before the file can be read

$$\begin{array}{ll} & tr = s \frown \langle frd.f.v \rangle \wedge s \upharpoonright \{\!| \ fcr \ |\!\} \neq \langle\rangle \\ \Rightarrow & \underline{\mathsf{last}}\,(s \upharpoonright \{\!| \ fcr, fwr \ |\!\}) \in \{\!| \ fwr \ |\!\} \ , \end{array} \quad (6.6)$$

implemented by

$$\begin{array}{rcl} C3File_0 & \widehat{=} & (frd?f?v \rightarrow C3File_0) \\ & & \square\ (fwr?f?v \rightarrow C3File_0) \\ & & \square\ (fcr?f \rightarrow C3File_1) \\ C3File_1 & \widehat{=} & (fwr?f?v \rightarrow C3File_0) \\ & & \square\ (fcr?f \rightarrow C3File_1) \ . \end{array}$$

Finally, the value returned through the frd channel reflects the last value written

$$
\begin{array}{rl}
& tr = s \frown \langle frd.f.v \rangle \wedge s \restriction \{\!| \ fwr \ |\!\} \neq \langle\rangle \\
\Rightarrow & \exists f' \bullet \underline{\mathsf{last}}\,(s \restriction \{\!| \ fwr \ |\!\}) = fwr.f'.v \ ,
\end{array}
\tag{6.7}
$$

implemented by

$$
\begin{array}{rcl}
C4File_0 & \widehat{=} & (frd?f?v \rightarrow C4File_0) \\
& & \Box\ (fwr?f?v \rightarrow C4File_1(v)) \\
C4File_1(v) & \widehat{=} & (frd?f!v \rightarrow C4File_1(v)) \\
& & \Box\ (fwr?f?v' \rightarrow C4File_1(v'))\ .
\end{array}
$$

The conjunction of these requirements is implemented by parallel composition of the four components above. The resulting process is given as $File_0$ below. Note that when the file is moved through the fmv channel, the file contents v remains unaltered.

$$
\begin{array}{rcl}
File_0 & \widehat{=} & fcr?f \rightarrow File_1(f) \\
File_1(f) & \widehat{=} & (frm.f \rightarrow File_0) \\
& & \Box\ (fmv.f?f' \rightarrow File_1(f')) \\
& & \Box\ (fwr.f?v \rightarrow File_2(f, v)) \\
File_2(f, v) & \widehat{=} & (frm.f \rightarrow File_0) \\
& & \Box\ (frd.f!v \rightarrow File_2(f, v)) \\
& & \Box\ (fwr.f?v' \rightarrow File_2(f, v')) \\
& & \Box\ (fmv.f?f' \rightarrow File_2(f', v))\ .
\end{array}
$$

Each file acts independently of the others; the indexed form of interleaving chosen for the definition of the file pool $Files$ below ensures that the file pool contains enough resources for the set of available file identifiers.

$$
\begin{array}{rcl}
\alpha Files & \widehat{=} & \{\!| \ fcr, frm, frd, fwr, fmv \ |\!\} \\
Files & \widehat{=} & |||_{i:Fid}\, File_0\ .
\end{array}
$$

Bucket model

A bucket is a collection of files with the same security classification. File names within each bucket are unique. The function $active$ returns the names of files which are active in bucket b after trace s.

$$
\begin{aligned}
active(\langle\rangle, b) &= \emptyset && (6.8)\\
active(s \frown \langle a\rangle, b) &= active(s, b) \cup \{f\},\\
&\quad \underline{\text{if}}\ a = fcr.b.f \ \vee\ (a = fmv.b'.f'.b.f \wedge b' \neq b)\\
&= active(s, b) \setminus \{f\},\\
&\quad \underline{\text{if}}\ a = frm.b.f \ \vee\ (a = fmv.b.f.b'.f' \wedge b' \neq b)\\
&= (active(s, b) \setminus \{f\}) \cup \{f'\},\\
&\quad \underline{\text{if}}\ a = fmv.b.f.b.f'\\
&= active(s, b), \quad \underline{\text{otherwise}}\ .
\end{aligned}
$$

In the function $active$ as well as process $Buck$ below we have to distinguish three cases for the fmv channel: (i) renaming file f within bucket b; (ii) moving f away from bucket b to another bucket; and (iii) moving f from another bucket to b.

$$
\begin{aligned}
tr = s \frown \langle fcr.b.f\rangle &\Rightarrow f \in Fname \setminus active(s, b) && (6.9)\\
tr = s \frown \langle frm.b.f\rangle &\Rightarrow f \in active(s, b) && (6.10)\\
tr = s \frown \langle fmv.b.f.b.f\rangle &\Rightarrow f \in active(s, b) && (6.11)\\
tr = s \frown \langle fmv.b.f.b.f'\rangle \wedge f \neq f' &\Rightarrow f' \in Fname \setminus active(s, b)\\
&\qquad \wedge f \in active(s, b) && (6.12)\\
tr = s \frown \langle fmv.b.f.b'.f'\rangle \wedge b' \neq b &\Rightarrow f \in active(s, b) && (6.13)\\
tr = s \frown \langle fmv.b'.f'.b.f\rangle \wedge b' \neq b &\Rightarrow f \in Fname \setminus active(s, b) && (6.14)
\end{aligned}
$$

The implementation ensures that any two files in bucket bu have distinct names, by keeping track of the set fs of active files.

$$
\begin{aligned}
\alpha Buck(bu) &\mathrel{\widehat=} \{\!|\ fcr.bu, frm.bu, fmv.bu.f.bu, fmv.bu.f.b', fmv.b'.f.bu\\
&\qquad\quad |\ b' \in Bucket \setminus \{bu\}, f \in Fname\ |\!\}\\
Buck(bu, fs) &\mathrel{\widehat=} (\Box_{f:Fname \setminus fs}\, fcr.bu.f \rightarrow Buck(bu, fs \cup \{f\}))\\
&\quad \Box\ (\Box_{f:fs}\, frm.bu.f \rightarrow Buck(bu, fs \setminus \{f\}))\\
&\quad \Box\ (\Box_{f:fs}\ \Box_{f':Fname \setminus fs}\\
&\qquad fmv.bu.f.bu.f' \rightarrow Buck(bu, (fs \setminus \{f\}) \cup \{f'\}))\\
&\quad \Box\ (\Box_{f:fs}\ \Box_{b':Bucket}
\end{aligned}
$$

$$
\begin{array}{l}
\quad \underline{\text{if}}\ bu = b' \ \underline{\text{then}}\ Stop \\
\quad \underline{\text{else}}\ fmv.bu.f.b'?f' \rightarrow Buck(bu, fs \setminus \{f\})) \\
\Box\ (\Box_{b':Bucket} \Box_{f:Fname \setminus fs} \\
\quad \underline{\text{if}}\ bu = b' \ \underline{\text{then}}\ Stop \\
\quad \underline{\text{else}}\ fmv.b'?f'!bu.f \rightarrow Buck(bu, fs \cup \{f\}))
\end{array}
$$

The parallel combination of the bucket processes, which initially contain no active files, must synchronize on the fmv channel when two different buckets are involved in the move:

$$
\begin{array}{rcl}
\alpha Buckets_0(bus) & \widehat{=} & \bigcup\{\alpha Buck(bu) \mid bu \in bus\} \\
Buckets_0(bus) & \widehat{=} & \|_{bu:bus}[\alpha Buck(bu)]Buck(bu, \emptyset)
\end{array}
$$

Buckets and files combined provide the basic low-level file system

$$
\begin{array}{rcl}
\alpha Buckets & \widehat{=} & \alpha Buckets_0(Bucket) \\
Buckets & \widehat{=} & Buckets_0(Bucket) \\
BF_0 & \widehat{=} & Files\ {}_{[\alpha Files \| \alpha Buckets]}\ Buckets\ .
\end{array}
$$

Although process $Files$ is non-deterministic – the non-determinism being caused when data is written to files with the same name and later retrieved – the constraints imposed by $Buckets$ are sufficient to remove this non-determinism.

Proposition 6.2.1 Process BF_0 is deterministic.

Lifting

The basic functionality of the low-level system will now be made available to the users interacting at the system interface. The channels for external users are given by

$$
\begin{array}{lcl}
\underline{\text{channel}} & cr, rm & : User.Fid \\
\underline{\text{channel}} & rd, wr & : User.Fid.Val \\
\underline{\text{channel}} & mv & : User.Fid.Fid
\end{array}
$$

These channels correspond to those for the files, but are extended by an additional field $User$ indicating which external agent communicates on the channel.

The event sets necessary for the verification of security are given by

$$\begin{array}{rcl} X(u) & \mathrel{\widehat{=}} & \{\!|\; rd.u, wr.u, cr.u, rm.u, mv.u \;|\!\} \\ External & \mathrel{\widehat{=}} & \bigcup\{X(u) \mid u \in User\} \;. \end{array} \tag{6.15}$$

The functionality of the system is lifted by renaming events on files to those at the interface, thus making each event available to each external user

$$\begin{array}{rcl} BF_1 & \mathrel{\widehat{=}} & BF_0 \,[\![fcr \leftarrow cr.u, frm \leftarrow rm.u, frd \leftarrow rd.u \mid u \in User]\!] \\ BF & \mathrel{\widehat{=}} & BF_1 \,[\![fwr \leftarrow wr.u, fmv \leftarrow mv.u \mid u \in User]\!] \\ \alpha BF & \mathrel{\widehat{=}} & \{\!|\; cr, rm, rd, wr, mv \;|\!\} \;. \end{array}$$

User interface

Process BF provides a useful filesystem in its own right, but it is not secure since every user – whatever its security clearance – has access to any file. It is thus necessary to add constraints to the interface in such a way as to separate the activity of agents engaging the interfaces which must not interfere. The standard Bell-LaPadula restrictions [BLP73] ensure that users have write access only on levels whose classification matches their own clearance,

$$\begin{array}{l} tr = s \frown \langle a \rangle \wedge a \in \{\!|\; cr.u.bu, rm.u.bu, mv.u.bu, wr.u.bu \;|\!\} \\ \qquad\qquad \Rightarrow class(bu) = clear(u) \;, \end{array} \tag{6.16}$$

and read access only on levels they are cleared to view:

$$tr = s \frown \langle a \rangle \wedge a \in \{\!|\; rd.u.bu \;|\!\} \quad \Rightarrow \quad class(bu) \leq clear(u). \tag{6.17}$$

These conditions are captured by process UIS whose parameter us is the set of users on which the constraints are applied.

$$\begin{array}{rcl} \alpha UIS(us) & \mathrel{\widehat{=}} & \{\!|\; rd.u, wr.u, cr.u, rm.u, mv.u \mid u \in us \;|\!\} \\ UIS(us) & \mathrel{\widehat{=}} & |||_{u:us}\; UI(u) \\ UI(u) & \mathrel{\widehat{=}} & (\Box_{bu:Bucket} \Box_{f:Fname} \\ & & \qquad \underline{\text{if}}\; class(bu) \neq clear(u)\; \underline{\text{then}}\; Stop \\ & & \qquad \underline{\text{else}}\; ((cr.u.bu.f \rightarrow UI(u)) \\ & & \qquad\qquad \Box\; (rm.u.bu.f \rightarrow UI(u)) \end{array}$$

$$
\begin{array}{l}
\qquad\qquad \Box\ (mv.u.bu.f?l'?f' \rightarrow UI(u)) \\
\qquad\qquad \Box\ (wr.u.bu.f?v \rightarrow UI(u)))) \\
\Box\ (\Box_{bu:Bucket} \Box_{f:Fname} \\
\quad \underline{\text{if}}\ clear(u) < class(bu)\ \underline{\text{then}}\ Stop \\
\quad \underline{\text{else}}\ (rd.u.bu.f?v \rightarrow UI(u)))
\end{array}
$$

An additional restriction applies to the moving of files: the source bucket and the target bucket must be identical (and therefore of the same classification).

$$
\begin{array}{rcl}
\alpha Moves(us) & \widehat{=} & \{\!|\ mv.u \mid u \in us\ |\!\} \\
Moves(us) & \widehat{=} & |||_{u:us}\, Move(u) \\
Move(u) & \widehat{=} & (\Box_{l:Bucket} \Box_{f:Fname} \Box_{f':Fname} \\
 & & \quad mv.u.l.f.l.f' \rightarrow Move(u)\ .
\end{array}
$$

Process $Moves$ ensures that users us cannot pass file data simply by downgrading of files, and also prohibits the information flow that would exist if low users could find out which files existed on higher levels by attempting to upgrade their own files.

6.3 Composition of system

The constraints imposed on the interface are applied by parallel composition of processes UIS and $Moves$ to the unconstrained system.

$$
\begin{array}{rcl}
FS_0 & \widehat{=} & BF\ [\alpha BF \| \alpha UIS(User)]\ UIS(User) \\
FS & \widehat{=} & FS_0\ [External \| \alpha Moves(User)]\ Moves(User)\ .
\end{array}
$$

The following example illustrates the functionality of the file system implementation.

Example 6.3.1 Process T_1 below engages in a valid trace of process FS, that is $FS \sqsubseteq_T T_1$. It is assumed that the system has two users $User = \{lisa, mari\}$ with $clear(lisa) \leq clear(mari)$. Furthermore, $B1$ and $B2$ are elements of $Bucket$ which satisfy

$$
class(B1) = clear(lisa) \wedge class(B2) = clear(mari)\ .
$$

User $mari$ initially creates file a in bucket $B2$ and writes value 1 to the file. Then $lisa$ creates file b in the low bucket $B1$ and writes value 0. The current value of the file in bucket $B1$ can always be read by the user with the higher clearance, also when the file has been renamed to a through channel mv.

$$\begin{aligned} T_1 &= cr.mari.B2.a \rightarrow wr.mari.B2.a.1 \rightarrow cr.lisa.B1.b \rightarrow \\ &\quad wr.lisa.B1.b.0 \rightarrow rd.mari.B2.a.0 \rightarrow mv.lisa.B1.b.B1.a \rightarrow \\ &\quad rd.mari.B1.a.0 \rightarrow wr.lisa.B1.a.1 \rightarrow rd.mari.B1.a.1 \rightarrow Stop\ . \end{aligned}$$

(End of Example)

6.4 Verification of Security

In this section, we will show that the filesystem FS is secure in the sense of Definition 4.3.2. This will be carried out be choosing an arbitrary partition of system interfaces, and deriving an alternative version of the system, based on this partition, which is provably equivalent to FS.

The first step is to choose an arbitrary member from the set of security labels, say $p \in Label$. The chosen value naturally partitions the users and buckets of the system as follows

$$\begin{aligned} Us_L &\mathrel{\widehat=} \{u \mid u \in User \wedge clear(u) \leq p\} \\ Us_H &\mathrel{\widehat=} User \setminus Us_L \end{aligned} \tag{6.18}$$

$$\begin{aligned} Buck_L &\mathrel{\widehat=} \{bu \mid bu \in Bucket \wedge clear(u) \leq p\} \\ Buck_H &\mathrel{\widehat=} Bucket \setminus Buck_L\ . \end{aligned} \tag{6.19}$$

Equally, the system alphabet Σ partitions into a "low" and a "high" interface

$$\begin{aligned} L &\mathrel{\widehat=} \{\!|\ cr.u, rm.u, rd.u, wr.u, mv.u \mid u \in Us_L\ |\!\} \\ H &\mathrel{\widehat=} \{\!|\ cr.u, rm.u, rd.u, wr.u, mv.u \mid u \in Us_H\ |\!\}\ . \end{aligned} \tag{6.20}$$

We will now define two processes which represent valid system access by external users to buckets $Buck_L$ and $Buck_H$, respectively. In order to achieve this, a correct number of file resources for each domain must be available. Process $FFiles$ provides a number of independent files whose number is sufficient for buckets bus:

$$FFiles(bus) \mathrel{\widehat=} |||_{i:bus}(|||_{j:Fname}\, File_0)\ .$$

The buckets and files for low users is given by

$$B_L \quad \widehat{=} \quad Buckets_0(Buck_L)\ [_{\alpha Buckets_0(Buck_L)}\|_{\alpha Files}]\ FFiles(Buck_L)$$

and lifted to interface level

$$\begin{array}{lcl} BF'_L & \widehat{=} & B_L\,[\![fcr \leftarrow cr.u, frm \leftarrow rm.u, fmv \leftarrow mv.u \mid u \in Us_L]\!] \\ BF''_L & \widehat{=} & BF'_L\,[\![fwr \leftarrow wr.u, frd \leftarrow rd.u' \mid u \in Us_L \wedge u' \in User]\!] \end{array}$$

as in Section 6.2. Note that access to the low buckets is made available only to low users ($u \in Us_L$), with the exception of the rd channel, in order to cater for high users monitoring the contents of low files.

In a similar manner, buckets and files for high users are given by

$$B_H \quad \widehat{=} \quad Buckets_0(Buck_H)\ [_{\alpha Buckets_0(Buck_H)}\|_{\alpha Files}]\ FFiles(Buck_H)$$

and lifted

$$\begin{array}{lcl} BF'_H & \widehat{=} & B_H\,[\![fcr \leftarrow cr.u, frm \leftarrow rm.u, fmv \leftarrow mv.u \mid u \in Us_H]\!] \\ BF''_H & \widehat{=} & BF'_H\,[\![fwr \leftarrow wr.u, frd \leftarrow rd.u \mid u \in Us_H]\!] \\ BF'''_H & \widehat{=} & BF''_H \mid\mid\mid Run_{RDdown}\ . \end{array}$$

There are two important differences here: (i) the renaming enables all events for high users only ($u \in Us_H$), without exception; and (ii) the events which will permit reading of low files are added by interleaving to the high agent BF''_H. This set is formally defined by

$$RDdown \quad \widehat{=} \quad \{\!|\ rd.u.bu \mid u \in Us_H \wedge bu \in Buck_L\ |\!\}\ .$$

As in Section 6.2, it is necessary to add constraints to the interface without which security cannot be achieved:

$$\begin{array}{rcl} CON(P, us) & \widehat{=} & (P\ |[_{\alpha UIS(us)}]|\ UIS(us))\ |[_{\alpha Moves(us)}]|\ Moves(us) \\ Down & \widehat{=} & CON(BF''_L, Us_L) \\ Up & \widehat{=} & CON(BF'''_H, Us_H)\ . \end{array}$$

The processes just defined are secure

Lemma 6.4.1 $H \not\longrightarrow L :: Down$ and $H \not\longrightarrow L :: Up$.

There are two equivalent ways to compose these two processes, depending on which of the parallel operators is used. Using the interface parallel operator, we define

$$Comp \quad \widehat{=} \quad Down \, [_{L \cup RDdown} \| _{H}] \; Up$$

which can equivalently be defined

$$Comp' \quad \widehat{=} \quad Down \; |[_{RDdown}]| \; Up$$

since both sides synchronize on set $RDdown$ only. Either way, the composition is just an alternative derivation of the original filesystem.

Theorem 6.4.2 Processes FS and $Comp$ are equivalent

$$FS \sqsubseteq_{FD} Comp \; \wedge \; Comp \sqsubseteq_{FD} FS \; .$$

This establishes the desired result:

Theorem 6.4.3 $H \not\longrightarrow L :: FS$, and thus the filesystem is secure in the sense of Definition 4.3.2.

Proof Independence of L in $Comp$ follows from Lemma 6.4.1 and Theorem 4.1.8, and FS is equivalent to $Comp$ by Theorem 6.4.2. These results hold for arbitrary $p \in Label$. □

6.5 Concluding Remarks

In this chapter, a multi-level filesystem with standard operations has been specified and proved secure with respect to its security policy. An alternative verification to the one given above would be to show that the filesystem FS is strongly-separable with respect to the low interface L. Either way, the effort of partitioning the interfaces of the system in a suitable way, and verifying independence on an equivalent version of the system is moderate if a verification tool such as FDR is at hand.

This approach to verification compares favourably to model-based approaches such as those using Z [Col94]. Although the functionality of a shared system such as a file system can be well specified in Z, a proof of security based on a transition system semantics appears to be long and complex. We propose to carry out verifications of security by translating Z models into CSP and apply independence on the corresponding communicating processes. This approach is illustrated with a file system specification in [RWW94, RWW96].

Chapter 7

Secure Concurrency Control

Distributed systems are difficult to build because individual components run concurrently and may require complex synchronizations in order to ensure correct system behaviour. Furthermore, during execution any part of the system may fail and thus prevent the still functioning system components to carry out the desired operations. The concept of concurrently executing transactions has been developed to deal with the dual problems of concurrency and failures.

A transaction is a sequence of operations whose execution (in a well-designed distributed system) appears to be *atomic*. Thus, an atomic transaction either completes successfully or has no effect at all. The effect of atomicity is the appearance of serial execution of transactions even though their individual steps are in fact interleaved. This atomicity property with respect to concurrency of execution is usually referred to as *serializability* [BL93, BHG87, Pap86].

In this chapter, we investigate the properties of one particular algorithm to achieve serializability, timestamp scheduling, applied to multi-level databases. Our aim is to characterize the relationship between scheduling requirements and security for timestamp scheduling. It will be shown that additional constraints imposed on a simple timestamp scheduling algorithm can conflict with security requirements, by introducing interference between transactions. Since we are concerned with the general properties of the scheduling algorithms, rather than a specific implementation, we confine our analysis to successfully executing transactions by not modelling aborts explicitly. As a consequence, the model is simple and elegant but does not take transaction deadlock into account.

In Section 7.1, our model of transactions is introduced, and the task of scheduling their concurrent execution is described. Section 7.2 provides an overview on desirable properties which transaction schedules generated by concurrency control algorithm may satisfy, such as serializability, strictness, and consistency. The algorithm for timestamp scheduling is discussed in Section 7.3 and modified for use on a multi-level database in Section 7.4. An analysis of the potential conflicts between security and scheduling requirements concludes the chapter.

7.1 Concurrent Transactions

In this section we will give a simple model of transactions on untyped databases. It will be assumed that each transaction has a unique identifier, and that data items in the database can be referred to by name.

$$[\, Trans, Data \,]$$

The set of transaction identifiers is given by $Trans$ and the set of names of data items by $Data$. (The values of these data items are of no concern here.) Each transaction t has a distinct starting point, which will be modelled with an event $start.t$, and finishing point, event $commit.t$, which marks the end of a successful execution and termination of t. While the transaction is active, it may read or write any data items in the database. The required channels are declared

$$\begin{array}{lcl} \underline{\text{channel}} & start, commit & : Trans \\ \underline{\text{channel}} & read, write & : Trans.Data \end{array}$$

The following gives the most basic model of a set of successfully executing transactions. An individual transaction i becomes active by engaging event $start.i$ and can carry out arbitrary accesses on the available data items. Once the transaction commits, no further data access can take place.

$$\begin{array}{rcl} \alpha Transactions & \mathrel{\widehat=} & \{\!|\ start, read, write, commit\ |\!\} \\ Transactions & \mathrel{\widehat=} & |||_{t:Trans}\ Trans_0(t) \\ Trans_0(i) & \mathrel{\widehat=} & start.i \rightarrow Active(i) \\ Active(i) & \mathrel{\widehat=} & (read.i?d \rightarrow Active(i)) \\ & & \Box\ (write.i?d \rightarrow Active(i)) \\ & & \Box\ (commit.i \rightarrow Stop)\ . \end{array}$$

According to the definition of process $Transactions$, different transactions can progress independently, which means that their individual steps may be arbitrarily interleaved. In practice, however, the goal of concurrency atomicity requires that some interleavings must not occur. It is the task of the concurrency control algorithm to decide which possible interleavings may be permitted, and reject those deemed to be illegal.

A concrete implementation of a concurrency control algorithm, often referred to as a scheduler, receives as input the sequences of operations which transactions are requesting to carry out on the database. The scheduler has to arrange the individual operations of different transactions in a schedule, i.e. the sequence of operations in which the requests are carried out. A schedule should meet two main requirements.

- Efficiency: a high degree of interleaving between operations, in order to achieve a high degree of concurrency between transactions.
- Correctness: a low degree of interleaving between operations, thus avoiding a potential compromise of atomicity properties.

The two goals of efficiency and correctness are in direct contradiction. It is straightforward to build a correct scheduler, by insisting that transactions are served in sequential order of their arrival, but this would be very inefficient in a distributed system. Below we describe the properties of schedules which are correct, in some well-defined manner, while achieving a high degree of efficiency.

7.2 Properties of Schedules

In this section, we discuss some of the properties which a scheduler might want to impose on its output schedules. The primary property is that of serializabililty in order to meet the requirement of concurrency atomicity. Beyond this concern for serializable schedules, the scheduler usually enforces properties which concern semantic or practical considerations. For example, in the case where a transaction aborts, the concurrency should not be forced to "undo" operations already carried out by a committed transaction, and be able to easily recover to a state where it can restart the aborted transaction.

Serializability

Serializability is the main correctness criterion for concurrently executing transactions. A schedule s is said to be *serializable* if there exists a serial schedule which is in some sense equivalent to s. A schedule is serial if there is no interleaving of operations from different transactions. The purpose of the serializability condition is to ensure that the effect of executing the serializable schedule is indistinguishable from the effect that would result from executing the operations of transactions together in sequence.

There are several variants of serializability depending on which notion of "equivalence" is applied to schedules, including final-state equivalence, view equivalence, and conflict equivalence [Pap86, BHG87]. We will summarize below the concept of conflict equivalence and conflict serializability, which is stronger than the other two conditions and appears to be the criterion on which virtually all concurrency controls are based.

The definition of conflict equivalence is based on the notion of conflict between two operations of different transactions. Two operations conflict if they do not commute, i.e. if the effect on the database would be different when their order of execution is reversed. For an untyped database with read and write access where only single versions of data items are stored, the following operations conflict:

- a $read.t_1.d$ is in conflict with a subsequent $write.t_2.d$, and
- a $write.t_1.d$ is in conflict with a subsequent $read.t_2.d$, and
- a $write.t_1.d$ is in conflict with a subsequent $write.t_2.d$.

These conflicts are referred to as r/w conflicts, w/r conflicts, and w/w conflicts, respectively. Two transactions carrying out conflicting operations on the same item are said to conflict. Transactions accessing only disjoint sets of data items cannot conflict.

Definition 7.2.1 Two schedules s_1 and s_2 are *conflict equivalent* if

1. both are interleavings of the same sequence of operations of the same transactions, and

2. conflicting operations are ordered in the same way in both s_1 and s_2.

Operations which are ordered differently in two conflict equivalent schedules must commute and hence have the same overall effect in either schedule.

A schedule s is said to be conflict serializable if there exists a serial schedule s' which is conflict equivalent to s. This provides a satisfactory condition of correctness since a serializable schedule can always be transformed into a serial schedule by repeatedly swapping the order of operations which commute.

Strictness and Recoverability

Beyond the immediate requirements for correctness, a scheduler usually enforces additional properties of schedules which relate to semantic or practical considerations. The semantics of a $commit.t$ operation is that all previous operations carried out by transaction t are valid and cannot be "undone". Without additional constraints on schedules these semantics cannot be achieved in the presence of aborting transactions.

In a distributed system, an active transaction may abort for several reasons: the transaction itself may request to terminate prematurely, or the concurrency control may insist on terminating the transaction to avoid reaching an incorrect state such as deadlock. To illustrate the difficulties introduced by transaction aborts, consider the schedule $\langle start.1, start.2, write.2.x, read.1.x, commit.1\rangle$. For clarity we rewrite the schedule in form of a table

T_1	T_2
$start.1$	
	$start.2$
	$write.2.x$
$read.1.x$	
$commit.1$	

where operations from the same transaction appear in the same column, and time progresses down the vertical axis. Here two transactions access the same data item x, and transaction T_1 is permitted to commit while T_2 is still active. If transaction T_2 now aborts then the original value of x must be restored. Furthermore, the read operation must be "undone" to reflect that original value, rather than the one written by T_2. This contradicts the semantics of the $commit.1$ operation, signalling the successful termination of T_1's execution.

Even without the final $commit.1$ step, there are practical considerations to disallow the $read.1.x$ while T_2 is still active. If T_1 reads x and T_2 aborts, then also T_1 must be aborted since it has read an illegal value. This problem is known as "cascading aborts", where the abort of one transaction immediately forces the premature termination of one or more other transactions.

A concurrency control should avoid the possibility of cascading aborts. This is achieved by disallowing transactions to read data items which have been previously been written by another still active transaction. The resulting schedules will not exhibit w/r conflicts between operations of different active transactions, and thus be "cascadeless":

$$\begin{array}{rcl} Cascadeless & \widehat{=} & Transactions\ [_{\alpha Transactions} \| _{\alpha NoWRcon}]\ NoWRcon \\ \alpha NoWRcon & \widehat{=} & \{\mid read, write, commit \mid\} \\ NoWRcon & \widehat{=} & \|_{d:Data}\ [\alpha D_1(d)] D_1(d) \end{array}$$

where $\alpha D_1(d) \widehat{=} \{\mid read.t.d, write.t.d, commit.t \mid t \in Trans \mid\}$. The process $D_1(d)$ for each data item d describes the constraints on the $read.t.d$ events, which may be carried out only if they are not in conflict with a previous write on d.

$$\begin{array}{rcl} D_1(d) & \widehat{=} & D_1'(d, \emptyset) \\ D_1'(d, ts) & \widehat{=} & (\ \underline{\text{if}}\ ts = \emptyset\ \underline{\text{then}}\ read?t!d \rightarrow D_1'(d, ts)\ \underline{\text{else}}\ Stop) \\ & & \Box\ (\ \underline{\text{if}}\ \underline{\text{card}}(ts)\ = 1 \\ & & \quad\quad \underline{\text{then}}\ \Box_{t:ts}\ read.t.d \rightarrow D_1'(d, ts)\ \underline{\text{else}}\ Stop) \\ & & \Box\ (write?t!d \rightarrow D_1'(d, ts \cup \{t\})) \\ & & \Box\ (commit?t \rightarrow D_1'(d, ts \setminus \{t\})) \end{array}$$

For each data item d, process $NoWRcon$ maintains a set ts of the uncommitted transactions which have written to the item. If this set is empty, any transaction can read x. If $\underline{\text{card}}(ts)\ = 1$, i.e. the set contains contains exactly one transaction t say, then the only transaction still permitted to read d is t itself – this does not constitute a w/r conflict. When two or more transactions have written to d, then no transaction is allowed to read. There are no constraints on writing: item d can be written by every transaction any time, in which case its identifier is recorded in ts. When a transaction commits, its identifier is removed from ts.

Typically a concurrency control enforces a stronger property called strictness, which solves the problems described above and also allows efficient rollback (restoring) of data values in some situations. Strictness prevents a transaction T_1 from reading or writing data items which have been written by another transaction T_2 while it is still active. Thus, strictness prevents both w/r and w/w conflicts between active transactions.

$$\begin{array}{rcl}
\alpha Strict & \widehat{=} & \{\!|\ start, read, write, commit\ |\!\} \\
Strict & \widehat{=} & Transactions\ [_{\alpha Transactions} \| _{\alpha Strict_0}]\ Strict_0 \\
\alpha Strict_0 & \widehat{=} & \{\!|\ read, write, commit\ |\!\} \\
Strict_0 & \widehat{=} & NoWRcon\ |[_{\{\!|\ read, write, commit\ |\!\}}]|\ NoWWcon \\
NoWWcon & \widehat{=} & \|_{d:Data} [\alpha D_2(d)] D_2(d)
\end{array}$$

where $\alpha D_2(d) \mathrel{\widehat{=}} \{\!|\ read.t.d, write.t.d, commit.t \mid t \in Trans\ |\!\}$.

$$\begin{array}{rcl}
D_2(d) & \widehat{=} & (read?t!d \to D_2(d)) \\
 & & \Box\ (write?t!d \to D_2'(d,t)) \\
 & & \Box\ (commit?t \to D_2(d)) \\
D_2'(d,t) & \widehat{=} & (read?s!d \to D_2'(d,t)) \\
 & & \Box\ (write.t.d \to D_2'(d,t)) \\
 & & \Box\ (commit?t2 \to \underline{\text{if}}\ t = t2\ \underline{\text{then}}\ D_2(d)\ \underline{\text{else}}\ D_2'(d,t))\ .
\end{array}$$

Process $D_2(d)$ prevents w/w conflicts for data item d between uncommitted transactions, by insisting that if transaction t writes d than no other transaction can write d until t commits.

Consistency

Although schedulers typically produce only strict schedules, the strictness condition is insufficient to guarantee external consistency. A schedule is said to be externally consistent if the output schedule is serializable in the order in which transactions have committed. An external user observing the execution of transactions would intuitively expect the transactions to execute in commit order, and thus expect the equivalent serial order of transactions to be consistent with the order implied by the externally observable results. Ideally, a concurrency control should insist on the existence of an equivalent serial trace s' in which the order of transactions is consistent with the order of commit operations in the actual output schedule s.

An example of a schedule without external consistency is given by

$$\langle start.2, read.2.x, start.3, write.3.x, commit.3, start.1,\\ read.1.y, commit.1, write.2.y, commit.2\rangle$$

which may also be written

T_1	T_2	T_3
	$start.2$	
	$read.2.x$	
		$start.3$
		$write.3.x$
		$commit.3$
$start.1$		
$read.1.y$		
$commit.1$		
	$write.2.y$	
	$commit.2$	

This schedule is both strict and conflict serializable; the order of transactions in the serial schedule is $T_1 \to T_2 \to T_3$. However, the schedule is not consistent because there is no equivalent serial schedule which respects the commit order (T_3 before T_1).

Breitbart et al [BGRS91] have shown that the dual requirement of conflict serializability and consistency is equivalent to the condition of strong recoverability. A schedule is strongly recoverable if for any pair of committed transactions T_i and T_j, whenever operation $op.i.x$ precedes operation $op.j.x$ in schedule s and these operations conflict, then $commit.i$ precedes $commit.j$ in s.

A relatively simple way of ensuring external consistency is through the condition of rigorousness [BGRS91]. A schedule s is rigorous if

- s is strict, and
- no data item may be written while a transaction which previously read it is still active.

Thus, a rigorous schedule is strict and prevents r/w conflicts between active transactions.

$$
\begin{array}{rcl}
\alpha Rigorous & \mathrel{\widehat{=}} & \{\!|\ start, commit, read, write\ |\!\} \\
Rigorous & \mathrel{\widehat{=}} & Strict\,[_{\alpha Strict}\|_{\alpha NoRWcon}]\,NoRWcon \\
\alpha NoRWcon & \mathrel{\widehat{=}} & \{\!|\ commit, read, write\ |\!\} \\
NoRWcon & \mathrel{\widehat{=}} & \big\|_{d:Data}\,[\alpha D_3(d)]D_3(d)
\end{array}
$$

where $\alpha D_3(d) \mathrel{\widehat{=}} \{\!|\ read.t.d, write.t.d, commit.t \mid t \in Trans\ |\!\}$. Process $D_3(d)$ describes the constraints on the write events on datum d; these are allowed only when they are not in conflict with a read access carried out by a transaction which is still active.

$$
\begin{array}{rcl}
D_3(d) & \mathrel{\widehat{=}} & D_3'(d, \emptyset) \\
D_3'(d, ts) & \mathrel{\widehat{=}} & (\underline{\text{if}}\ ts = \emptyset\ \underline{\text{then}}\ write?t!d \to D_3'(d, ts)\ \underline{\text{else}}\ Stop) \\
 & & \Box\ (\ \underline{\text{if}}\ \underline{\text{card}}(ts)\ = 1\ \underline{\text{then}}\ \Box_{t:ts}\ write.t.d \to D_3'(d, ts) \\
 & & \qquad \underline{\text{else}}\ Stop) \\
 & & \Box\ (read?t!d \to D_3'(d, ts \cup \{t\})) \\
 & & \Box\ (commit?t \to D_3'(d, ts \setminus \{t\}))\ .
\end{array}
$$

Like $D_1(d)$, process $D_3(d)$ maintains a set of transaction identifiers; those of transactions which have read d and are still active. The definition of the process is just like that of $D_1(d)$, with the roles of read and write events reversed.

The restrictions on schedules imposed by the conditions of "cascadelessness", strictness, and rigorousness are increasingly strong. The stronger the condition on schedules, the smaller is the set of possible schedules which a concurrency control may produce:

Proposition 7.2.2 Processes $Cascadeless$, $Strict$ and $Rigorous$ are trace refinements of each other, that is

$$Cascadeless \sqsubseteq_T Strict \quad \wedge \quad Strict \sqsubseteq_T Rigorous\ .$$

Secure Schedules

In a multi-level database there are items holding more sensitive information. We assume that the set of data items is partitioned into a set of highly classified items $Data_H$, and a set of lowly classified items, $Data_L$. Naturally only privileged users may gain access to the items in $Data_H$. Users with a high security clearance access the database via high transactions, $Trans_H$, and users with low clearance via low transactions, $Trans_L$; these sets are assumed to partition $Trans$.

Different models of security are possible depending on precisely which data items the two sets of transactions may read or write. We will consider a model where

- low transactions can read or write low items only, and
- high transactions can write high items only, but may read both low and high items.

This type of model with read-downs but without write-ups has been termed a "class 1" model [KTS93]. Without this possibility of read-down, there would be no issue of security: the two sets of transactions would then access disjoint sets of items, and there would be no scope for interaction between the interfaces. Such a system could be implemented with two separate schedulers, one for each interface.

There will inevitably be interaction between the H and L interface when high transactions read low data items; such an access could create conflicts which may delay or even abort low transactions. It is the task of the concurrency control to schedule individual steps of transactions in such a manner that a read-down can never cause a conflict, while imposing the overall scheduling discipline on the complete set of transactions. The concurrency control should permit read-downs without giving the agent at L the chance to infer the activity of high transactions, on the basis of which steps may (or may not) be observed in the low interface.

Our model of a "class 1" concurrency control is simply given by the process $Access$ below.

$$\alpha Access \quad \widehat{=} \quad \{\!|\ read, write\ |\!\}$$

$$
\begin{aligned}
Access \quad \widehat{=} \quad & (write?t : Trans_L?d : Data_L \to Access) \\
& \Box\ (read?t : Trans_L?d : Data_L \to Access) \\
& \Box\ (write?t : Trans_H?d : Data_H \to Access) \\
& \Box\ (read?t : Trans_H?d : Data \to Access)\ .
\end{aligned}
$$

According to this definition, transactions access items on their own security level, while high users have the privilege to carry out read accesses on all items.

7.3 Timestamp Scheduling

We will now consider a particular algorithm for concurrency control which is known to produce conflict serializable schedules. In this algorithm, the scheduler distinguishes between transactions according according to their starting order: if a transaction t_1 starts executing before t_2 then t_1 is considered to be *older* than t_2, while t_2 is said to be the *younger* transaction. Thus, the time-order between transactions is determined by the relative order of their *start* events. Due to this linear time-order imposed on transaction, the algorithm is known as timestamp scheduling.

Specification

The basic idea of timestamp scheduling is that conflicting operations in an output schedule must respect the time order between transaction. Thus, two conflicting operations op_1 and op_2 can occur in a trace, in this particular order, only if the transaction issuing op_1 is older than the transaction causing the conflict. Following Papadimitriou [Pap86], the test which the timestamp scheduler applies to possible next operations on the database may be described as follows:

> A step can always proceed provided no conflicting steps of a younger active transaction have already been output.

This algorithm obviously meets the intention behind the time-ordering between transactions: if a younger transaction s engages in an access that may conflict with a possible step of an older transaction t, then t will not be able to carry out this future access. Therefore, conflicts which do not respect the time-order of transactions cannot occur.

We will now proceed to give a CSP description of the algorithm described above. The specification is given by process TS defined as the composition of processes $Transactions$, the basic model of transactions, and $ConflictsTs$ describing the r/w, w/w, and w/r conflicts which occur during scheduling.

$$\begin{array}{rcl}
\alpha TsSpec & \widehat{=} & \{\!|\ start, commit, read, write\ |\!\} \\
TsSpec & \widehat{=} & Transactions\ [_{\alpha Transactions}\|_{\alpha ConflictsTs}]\ ConflictsTs \\
\alpha ConTs(d) & \widehat{=} & \{start.i, commit.i, read.i.d, write.i.d \mid i \in Trans\} \\
\alpha ConflictsTs & \widehat{=} & \{\!|\ start, commit, read, write\ |\!\} \\
ConflictsTs & \widehat{=} & \|_{d:Data}[\alpha ConTs(d)]ConTs_0(d) \\
ConTs_0(d) & \widehat{=} & \|_{t:Trans}[\alpha ConTs(d)]ConTs(d,t)
\end{array}$$

Process $ConTs(d,t)$ describes the conflicts between transaction t and other transactions for access to data item d. Initially, there are no conflicts.

$$\begin{array}{rcl}
ConTs(d,t) & \widehat{=} & (start?s \rightarrow \\
& & \quad \underline{\text{if}}\ s \neq t\ \underline{\text{then}}\ ConTs(d,t) \\
& & \quad \underline{\text{else}}\ ConTs'(d,t,\emptyset,\emptyset,\emptyset)) \\
& & \Box\ (read?s!d \rightarrow ConTs(d,t)) \\
& & \Box\ (write?s!d \rightarrow ConTs(d,t)) \\
& & \Box\ (commit?s \rightarrow ConTs(d,t))
\end{array}$$

Once active, transaction t it keeps track of transactions starting later (set $younger$) that itself. The conflicts created by the access to item d by a younger transaction will be recorded in the sets $rcon$ (for read conflicts) and $wcon$ (for write conflicts). Transaction t can access d only if there are no conflicts (the sets are empty).

$$\begin{array}{l}
ConTs'(d,t,younger,rcon,wcon)\ \widehat{=} \\
\quad (start?s \rightarrow ConTs'(d,t,younger \cup \{s\},rcon,wcon)) \\
\quad \Box\ (\Box_{s:Trans\setminus\{t\}}\ read.s.d \rightarrow \\
\qquad \underline{\text{if}}\ s \in younger \\
\qquad \underline{\text{then}}\ ConTs'(d,t,younger,rcon,wcon \cup \{s\}) \\
\qquad \underline{\text{else}}\ ConTs'(d,t,younger,rcon,wcon))
\end{array}$$

$\Box$ ($\underline{\text{if}}$ $rcon = \emptyset$

$\quad \underline{\text{then}}$ $read.t.d \rightarrow ConTs'(d, t, younger, rcon, wcon)$

$\quad \underline{\text{else}}$ $Stop$)

$\Box$ ($\Box_{s:\,Trans \setminus \{t\}}$ $write.s.d \rightarrow$

$\quad\quad \underline{\text{if}}$ $s \in younger$

$\quad\quad \underline{\text{then}}$ $ConTs'(d, t, younger, rcon \cup \{s\}, wcon \cup \{s\})$

$\quad\quad \underline{\text{else}}$ $ConTs'(d, t, younger, rcon, wcon)$)

$\Box$ ($\underline{\text{if}}$ $rcon = \emptyset \wedge wcon = \emptyset$

$\quad \underline{\text{then}}$ $write.t.d \rightarrow ConTs'(d, t, younger, rcon, wcon)$

$\quad \underline{\text{else}}$ $Stop$)

$\Box$ ($commit?s \rightarrow ConTs'(d, t, younger, rcon, wcon)$)

Implementation

In this subsection we give an alternative description of the basic scheduling algorithm. It is based on the explicit use of timestamps, and thus suggestive of a particular method of implementing the scheduler.

Each transaction is assigned a unique timestamp, which reflects the time at which it starts processing. In the version below, time is measured in units of $start.i$ events: time progresses with each new transaction, so that there is a correspondence between the current "time" and the number of transactions which have been processed. A transaction t_i is older than t_j if and only if t_i's timestamp is smaller than t_j's.

Each item in the database also has two associated timestamps: for each d, the read-timestamp rts_d is the timestamp of the youngest transaction that has read d. The write-timestamp wts_d of d is the timestamp of the youngest transaction that wrote it. The test which the scheduler applies to possible next steps can now be described as follows [Pap86].

> If the timestamp ts of a transaction t_i is greater or equal to the write-timestamp wts_d, then t_i can read d. If timestamp ts is greater or equal to both rts_d and wts_d, then t_i can write d.

In order to implement this algorithm with CSP, we introduce two additional channels: the occurrence of a $tock$ represents the passing of time, and the channel $tstamp$ is used to convey the transactions' timestamps to the data items.

$$\begin{array}{lll} \underline{\text{channel}} & tock & \\ \underline{\text{channel}} & tstamp & : Trans.Time \end{array}$$

The timestamping algorithm is given by process $TsSched1$, defined as the parallel composition of processes $TsItems$ and $TsTstampClock$, with the $timestamp$ channel concealed.

$$\begin{array}{rcl} Ts & \widehat{=} & Transactions \left[{}_{\alpha Transactions} \| {}_{\alpha TsSched1}\right] TsSched1 \\ \alpha TsSched1 & \widehat{=} & \{\!|\ start, commit, read, write\ |\!\} \\ TsSched1 & \widehat{=} & (TsItems \left[{}_{\alpha TsItems} \| {}_{\alpha TsTstampClock}\right] TsTstampClock) \setminus TStamp \end{array}$$

with the management of timestamps achieved by

$$\begin{array}{rcl} \alpha TsTstampClock & \widehat{=} & \{\!|\ start, commit, tstamp\ |\!\} \\ TsTstampClock & \widehat{=} & (TsTstamp \left[{}_{\alpha TsTstamp} \| {}_{\alpha TsClock}\right] TsClock) \setminus Tock \end{array}$$

where $TStamp \widehat{=} \{\!|\ tstamp\ |\!\}$ and $Tock \widehat{=} \{\!|\ tock\ |\!\}$. Process $TsClock$ ensures that time progresses one unit for each starting transaction.

$$\begin{array}{rcl} \alpha TsClock & \widehat{=} & \{\!|\ tock, start\ |\!\} \\ TsClock & \widehat{=} & tock \rightarrow start?i \rightarrow TsClock \end{array}$$

Process $TsTstamp$ assigns a timestamp to each transaction i, and ensures the the transaction communicates its correct value.

$$\begin{array}{rcl} \alpha TsTstamp & \widehat{=} & \{\!|\ tock, start, commit, tstamp\ |\!\} \\ TsTstamp & \widehat{=} & \|_{i:Trans} [\alpha TsTstamp0(i)]\, TsTstamp0(i) \end{array}$$

$$\begin{array}{rcl} \alpha TsTstamp0(i) & \widehat{=} & \{\!|\ tock, start.i, commit.i, tstamp.i.ts \\ & & \qquad \mid ts \in Time\ |\!\} \\ TsTstamp0(i) & \widehat{=} & TsTstamp'(i, 0) \\ TsTstamp'(i, tocks) & \widehat{=} & (\underline{\text{if}}\ tocks + 1 < \underline{\text{card}}(Time) \\ & & \qquad \underline{\text{then}}\ tock \rightarrow TsTstamp'(i, tocks + 1) \end{array}$$

$$
\begin{array}{lcl}
 & & \qquad \underline{\text{else}}\ Stop) \\
 & & \Box\ (start.i \rightarrow TsTstamp''(i, tocks, tocks)) \\
TsTstamp''(i, ts, tocks) & \mathrel{\widehat=} & (\ \underline{\text{if}}\ tocks + 1 < \underline{\text{card}}(Time) \\
 & & \qquad \underline{\text{then}}\ tock \rightarrow TsTstamp''(i, ts, tocks + 1) \\
 & & \qquad \underline{\text{else}}\ Stop) \\
 & & \Box\ (tstamp.i.ts \rightarrow TsTstamp''(i, ts, tocks)) \\
 & & \Box\ (commit.i \rightarrow TsTstamp'''(i, ts, tocks)) \\
TsTstamp'''(i, ts, tocks) & \mathrel{\widehat=} & tock \rightarrow TsTstamp'''(i, ts, tocks)
\end{array}
$$

Process $TsItem$ lies at the heart of the algorithm. It ensures that accesses to data items are only carried out in accordance with timestamping scheme described above.

$$
\begin{array}{rcl}
\alpha TsItems & \mathrel{\widehat=} & \{\!|\ tstamp, read, write\ |\!\} \\
TsItems & \mathrel{\widehat=} & |||_{d:Data}\ TsItem(d) \\
TsItem(d) & \mathrel{\widehat=} & TsItem'(d, 0, 0, \emptyset)
\end{array}
$$

The parameters rts and wts of process $TsItem'$ hold the values of the two timestamps for datum d. The parameter tss is a sequence of pairs of the form (i, ts) where ts is transaction i's timestamp, thus permitting us to keep track of transaction timestamps.

$$
\begin{array}{l}
TsItem'(d, rts, wts, tss)\ \mathrel{\widehat=} \\
\quad (\Box_{i:Trans \setminus \{j \mid (j, ts') \in tss\}} \\
\qquad tstamp.i?ts \rightarrow TsItem'(d, rts, wts, tss \cup \{(i, ts)\})) \\
\quad \Box\ (\Box_{(i,ts):tss}(\ \underline{\text{if}}\ ts < wts\ \underline{\text{then}}\ Stop \\
\qquad\qquad\qquad \underline{\text{else}}\ read.i.d \rightarrow \\
\qquad\qquad\qquad\quad (\ \underline{\text{if}}\ rts < ts \\
\qquad\qquad\qquad\quad \underline{\text{then}}\ TsItem'(d, ts, wts, tss) \\
\qquad\qquad\qquad\quad \underline{\text{else}}\ TsItem'(d, rts, wts, tss)))) \\
\quad \Box\ (\Box_{(i,ts):tss}(\ \underline{\text{if}}\ (ts < rts \vee ts < wts)\ \underline{\text{then}}\ Stop \\
\qquad\qquad\qquad \underline{\text{else}}\ write.i.d \rightarrow \\
\qquad\qquad\qquad\quad (\ \underline{\text{if}}\ wts < ts \\
\qquad\qquad\qquad\quad \underline{\text{then}}\ TsItem'(d, rts, ts, tss) \\
\qquad\qquad\qquad\quad \underline{\text{else}}\ TsItem'(d, rts, wts, tss))))\ .
\end{array}
$$

The specification $TsSpec$ and implementation Ts are just alternative versions of the same algorithm, since they have the same behaviour.

Proposition 7.3.1 Specification and implementation are equivalent

$$TsSpec \sqsubseteq_{FD} Ts \quad \wedge \quad Ts \sqsubseteq_{FD} TsSpec \ .$$

Since the implementation details make process Ts considerably more complex, we will confine our attention to the description of the secure timestamping algorithm in terms of conflicts only.

7.4 Secure Timestamp Scheduling

We will now present the secure scheduling algorithm, which is a variation of the basic timestamping algorithm described in the previous section. The concurrency control has to ensure that read-downs carried out by high transactions cannot be the source of conflicts with steps taken by low transactions. The basic test which the secure scheduler applies to possible next transaction steps is as follows.

> A step can proceed provided no conflicting steps of a younger active transaction have already been output. A read-down step accessing item d can proceed only if
>
> 1. there are no older active low transactions, and
> 2. none of the younger active low transactions has written d.

The motivation for these restrictions on read-downs is straightforward. Firstly if there was an active low transaction which is older than the high transaction reading down, then this step would create the possibility of conflict should the older transaction request to write to the same data item. Secondly, were the high transaction to read data previously written by a younger transaction, then the scheduling discipline would be violated since there would be a w/r conflict between active transactions. Since the standard timestamping algorithm already prevents such conflicts, only the first constraint on read-downs needs to be explicitly enforced.

The additional constraint on the read-down steps will be captured with process $RdDown1$.

$$
\begin{array}{rcl}
\alpha RdDown1 & \mathrel{\widehat=} & \bigcup(\{\alpha RdDown1_0(t) \mid t \in \mathit{Trans}\}) \\
RdDown1 & \mathrel{\widehat=} & \|_{t:\mathit{Trans}} [\alpha RdDown1_0(t)] RdDown1_0(t) \\
RdDown1_0(t) & \mathrel{\widehat=} & \{\!|\ \mathit{start}, \mathit{commit}, \mathit{read}.t\ |\!\} \\
RdDown1_0(t) & \mathrel{\widehat=} & RdDn1(t, \emptyset)
\end{array}
$$

The parameter older in process $RdDn1(t, \mathit{older})$ keeps track of those transactions which started before transaction t.

$$
\begin{array}{rcl}
RdDn1(t, \mathit{older}) & \mathrel{\widehat=} & (\mathit{start}?t2 \rightarrow \\
 & & \qquad \underline{\mathsf{if}}\ t = t2\ \underline{\mathsf{then}}\ RdDn1'(t, \mathit{older}) \\
 & & \qquad \underline{\mathsf{else}}\ RdDn1(t, \mathit{older} \cup \{t2\})) \\
 & & \Box\ (\mathit{commit}?t2 \rightarrow RdDn1(t, \mathit{older} \setminus \{t2\})) \\
 & & \Box\ (\mathit{read}.t?d \rightarrow RdDn1(t, \mathit{older})) \\
RdDn1'(t, \mathit{older}) & \mathrel{\widehat=} & (\mathit{start}?t2 \rightarrow RdDn1'(t, \mathit{older})) \\
 & & \Box\ (\mathit{commit}?t2 \rightarrow \\
 & & \qquad \underline{\mathsf{if}}\ t = t2\ \underline{\mathsf{then}}\ RdDn1(t, \mathit{older}) \\
 & & \qquad \underline{\mathsf{else}}\ RdDn1'(t, \mathit{older} \setminus \{t2\})) \\
 & & \Box\ (\ \underline{\mathsf{if}}\ t \in \mathit{Trans}_L \vee \mathit{older} = \emptyset \\
 & & \qquad \underline{\mathsf{then}}\ \mathit{read}.t?d \rightarrow RdDn1'(t, \mathit{older}) \\
 & & \qquad \underline{\mathsf{else}}\ (\Box_{d:\mathit{Data}_H}\ \mathit{read}.t.d \rightarrow RdDn1'(t, \mathit{older})))
\end{array}
$$

The constraints on the concurrency control imposed by multi-level security will now be added to the basic timestamping algorithm above.

$$
\begin{array}{rcl}
\alpha \mathit{Secure} & \mathrel{\widehat=} & \alpha \mathit{Access} \cup \alpha RdDown1 \\
\mathit{Secure} & \mathrel{\widehat=} & \mathit{Access}\ [{\scriptstyle \alpha \mathit{Access} \| \alpha RdDown1}]\ RdDown1\ .
\end{array}
$$

The basic secure timestamping scheduler is then given by

$$
\begin{array}{rcl}
\alpha \mathit{SecTsSpec} & \mathrel{\widehat=} & \{\!|\ \mathit{start}, \mathit{commit}, \mathit{read}, \mathit{write}\ |\!\} \\
\mathit{SecTsSpec} & \mathrel{\widehat=} & \mathit{TsSpec}\ [{\scriptstyle \alpha \mathit{TsSpec} \| \alpha \mathit{Secure}}]\ \mathit{Secure}
\end{array}
$$

Since we have simply constrained the original algorithm TsSpec through a parallel composition, we can be sure that process $\mathit{SecTsSpec}$ produces serializable schedules:

Proposition 7.4.1 We have $TsSpec \sqsubseteq_T SecTsSpec$, and thus each trace of $SecTsSpec$ is serializable.

Furthermore, the concurrency control is secure

Theorem 7.4.2 $H \not\longrightarrow L :: SecTsSpec$.

Adding further Constraints

As we have shown in Section 7.2, considerations beyond correctness make it desirable to impose additional constraints on the basic concurrency control algorithm. For instance, we can add the requirements for strict or rigorous schedules as follows

$$\begin{array}{lcl} SecTsSpec1 & \widehat{=} & Strict\,[_{\alpha Strict}\|_{\alpha SecTsSpec}]\,SecTsSpec \\ SecTsSpec2 & \widehat{=} & Rigorous\,[_{\alpha Rigorous}\|_{\alpha SecTsSpec}]\,SecTsSpec\ . \end{array}$$

It turns out that only the strict version of the algorithm is secure.

Theorem 7.4.3 $H \not\longrightarrow L :: SecTsSpec1$.

The rigorous version fails to be secure. To show a concrete example, assume that $x \in Data_L$, and that t_1 and t_2 are a low and a high transaction, respectively. According to the secure algorithm, t_2 may read x before t_1 becomes active in order to write. That is,

$$\langle start.t_2, read.t_2.x, start.t_1, write.t_1.x\rangle \in \text{TRACES}(SecTsSpec)\ .$$

With the additional requirement of rigorous schedules, however, this is not a legal trace, because rigorousness prevents r/w conflicts between active transactions. Thus, what had been a legal interleaving of steps from two transactions must now be prohibited. The r/w conflict (and the prior step taken by t_1) will show up in the L interface by t_2's inability to write x, even though this could not have been caused by other low transactions (there are none).

The failure of security will be shown up by our condition as follows. The process $\mathcal{I}_H(SecTsSpec2)$ is non-deterministic because

$$\langle start.t_2, read.t_2.x, start.t_1, write.t_1.x\rangle \in \text{TRACES}(\mathcal{I}_H(SecTsSpec2))\ \wedge$$
$$(\langle start.t_2, read.t_2.x, start.t_1\rangle, \{write.t_1.x\}) \in \text{FAILS}(\mathcal{I}_H(SecTsSpec2))\ .$$

The event $write.t_1.x$ can be refused after the trace when its initial two events have derived from process $SecTsSpec2$, rather than having been inserted by Run_H.

7.5 Discussion

In this chapter, we have presented an algorithm for concurrency control in multi-level secure database systems. Our algorithm is based on timestamps which are associated with transactions when they begin execution. A multitude of alternative algorithms to timestamping have been proposed, including locking, conflict graph scheduling, and multi-version scheduling which increases the degree of concurrency between transactions by maintaining multiple versions of data items [Pap86, BHG87]. Our model of transactions is grossly simplified; we have chosen to consider only successful execution of transactions. A more realistic model including explicit transaction aborts would be necessary to avoid deadlock between transactions with conflicting access requests.

Sandhu has pointed out that integrity (i.e. correctness) and confidentiality (security) are often in direct conflict in multi-level database systems. *"In order to fully understand this conflict, and its possible resolutions, we need security models which unify these objectives in a common framework. So far, all that researchers have done is approach this issue in an ad hoc manner"* [San92]. Indeed a large proportion of published research appears to focus on the possible implementation of individual algorithms for particular architectures, such as kernelized, replicated, trusted subject, and integrity lock systems. A contrasting approach is to identify simple transaction processing primitives which can be combined in parallel and provide general models of database concurrency controls [Woo90]. Here we have attempted to capture the essence of timestamp concurrency control in terms of the conflicts between access requests; other algorithms can be characterized in a similar way. Furthermore, we have specified additional constraints which a typical concurrency control imposes on transaction schedules for semantic or practical reasons, and shown that rigorousness can introduce information flows which do not exist in a strict, secure timestamp scheduler.

Keefe et al [KTS93] discuss a number of criteria for secure transaction schedulers, including data-conflict security, strong-value security, schedule and coschedule security etcetera. In contrast, we apply the standard independence criterion to the proposed scheduling algorithms, removing the need for security conditions which cannot be applied outside the database arena. This approach makes it possible to reconcile correctness and security concerns within a common modelling framework applicable to a wide range of computing systems.

Chapter 8

Conclusions

In this final chapter, we will survey the publications of a number of authors whose work relates to that described in this thesis. Section 8.1 is divided into two parts, covering first standard non-deterministic and then a probabilistic model.

In Section 8.2 we summarize the contributions of this thesis and highlight possible areas of further research.

8.1 Comparisons

The first part of this section covers frameworks for the development of standard deterministic or non-deterministic secure systems. In the second part, we investigate the modelling of security and independence in a framework with probabilistic choice. We proceed in roughly chronological order.

Standard Systems

Sutherland [Sut86] proposes a model of deducibility for abstract state machines. A definition of information flow is given which is phrased in terms of "information functions". Such functions project from the complete set of behaviours of a state machine.

"Given a set of possible worlds W and two functions f_1 and f_2 with domain W, we say that *information flows* from f_1 to f_2 if and only if there exists some possible world w and some element z in the range of f_2 such that z is achieved by f_2 in some possible world but in every possible world w' such that $f_1(w') = f_1(w)$, $f_2(w')$ is not equal to z."

The possible worlds are the execution sequences (traces) of a computing system, and the information functions are the functions that extract the individual agents' system view from the traces. In the concrete CSP interpretation which we may apply to this definition, the functions are the trace restrictions $\upharpoonright H$ and $\upharpoonright L$. A system in which no information flows (according to Sutherland) is said to be deducibility-secure.

Definition 8.1.1 Let Σ be partitioned by H, L. Process S is *deducibility-secure* if, and only if, for all $t \in \textsc{Traces}(S)$ and $z \in \textsc{Traces}(S @ L)$ we have

$$\exists\, t' \in \textsc{Traces}(S) \quad \bullet \quad t' \upharpoonright H = t \upharpoonright H \;\wedge\; t' \upharpoonright L = z \ .$$

Thus for every trace t and every possible subtrace z that may be observed through the L interface, there must be a trace t' with the same H behaviour as trace t that could have resulted in the observation of z in interface L.

Example 8.1.2 Let $\Sigma \mathrel{\widehat{=}} \{a, x\}$ and $L \mathrel{\widehat{=}} \{a\}$. Consider

$$R \quad \widehat{=} \quad (a \to x \to \mathit{Stop}) \;\Box\; (x \to \mathit{Stop}) \ ,$$

which is deducibility-secure according to Definition 8.1.1. However, process

$$S \quad \widehat{=} \quad a \to x \to \mathit{Stop} \ .$$

would not be regarded as secure, because it is possible to deduce that no x event will occur as long as there is no activity in interface L. Formally, the trace $t = \langle a, x \rangle$ and the subtrace $z = \langle\rangle$ of S provide evidence of this flow of information from H to L. The only trace t' with the same H behaviour as t is $t' = \langle a, x \rangle$ itself, but $\langle a, x \rangle \upharpoonright L \neq \langle\rangle$. Therefore it is not possible to engage in event x without the prior occurrence of event a in interface L. (End of Example)

Example 8.1.2 illustrates that the above interpretation of deducibility-security and the condition of independence are incomparable. Interface L is not independent in R although this process is deducibility-secure. However, interface L is independent in S.

Jacob [Jac87, Jac90] develops a theory of confidentiality in terms of the flow of information between users in a shared computer system. The core aim of this theory is the definition of a relation $\preceq_L$ between processes which describes the condition under which a process provides at least the same amount of confidentiality as another.

Jacob's approach is based on the use of *inference functions* which provide information on what can be inferred about process behaviour from a particular observation in a single process interface. If the low agent observes the subtrace t in interface L, then the function $\mathsf{infer}(S, L, t)$ returns the traces of process S which could have given rise to t. If $t \in \textsc{Traces}(S\,@L)$ then the inference function is defined

$$\mathsf{infer}(S, L, t) \quad \widehat{=} \quad \{r \mid r \in \textsc{Traces}(S) \wedge r \upharpoonright L = t\}\ . \qquad (8.1)$$

The greater the set returned by $\mathsf{infer}(S, L, t)$, the wider is the range of possible behaviours of S which enabled the low agent to observe t in interface L. The wider this range, the less certain one can be of which process behaviour has actually taken place. A large amount of uncertainty is identified with a high degree of confidentiality: process S is at least as secure as R through interface L, written $R \preceq_L S$ if, and only if

$$\begin{aligned} \forall\, t \in \textsc{Traces}(S\,@L) \quad \bullet \quad & t \in \textsc{Traces}(R\,@L) \wedge \\ & \mathsf{infer}(R, L, t) \subseteq \mathsf{infer}(S, L, t)\ . \end{aligned} \qquad (8.2)$$

The more that can be inferred from a particular observation of process S, the less can be deduced about the H actions which may or may not have occurred in the H interface. This uncertainty provides the basis for a secure system. According to Jacob [Jac90], the ordering $\preceq_L$ serves a purpose similar to the standard refinement order $\sqsubseteq$ for safety properties, apart from measuring the degree of security rather than functional correctness and safety. However, this security order can sometimes yield results which do not coincide with our intuitive understanding of non-interference, as the following example illustrates.

Example 8.1.3 Assume that Σ is partitioned by $H \mathrel{\widehat{=}} \{x\}$ and $L \mathrel{\widehat{=}} \{a\}$. Consider the following two processes (which also appear in Example 8.1.2).

$$\begin{aligned} R &\mathrel{\widehat{=}} (a \to x \to STOP) \ , \\ S &\mathrel{\widehat{=}} (a \to x \to STOP) \,\square\, (x \to STOP) \ . \end{aligned}$$

Intuitively we would regard process R as secure, because the x event cannot delay or affect the prior occurrence of a. In fact interface L is independent in R, but this is not the case for S: if the high agent is quick to engage in x before the a event can occur, then a is removed from the L interface and further progress is impossible. Thus the L interface depends on H activity and is not trace-invariant (Definition 3.1.1).

According to (8.2), however, process S is at least as secure as R; in fact S is strictly more secure. This seems to contravene our intuitive understanding of the nature of non-interference, since S appears to be inherently insecure for the reasons outlined above. (End of Example)

Jacob also defines a non-interference property called *restriction of information flow* which is given in terms of inference functions.

Definition 8.1.4 [Jac90, page 231] Suppose that Σ is partitioned by H, L. Process S satisfies *restriction of information flow* if, and only if,

$$t \in \textsc{Traces}(S\,@L) \quad \Rightarrow \quad L^* \cap \mathsf{infer}(S, L, t) \neq \emptyset \ .$$

The intention behind this definition is as follows. Whatever trace t is observed through interface L, it must be possible that the system has not engaged in any events outside L. That is, although the low agent may know to which behaviours the H interface is restricted, it is impossible to deduce which of those possible behaviours have actually occurred [Jac90].

As examples of restricted information flow, consider again the processes R and S of Example 8.1.3. With $L \mathrel{\widehat{=}} \{a\}$, both processes pass the condition and are therefore regarded as secure. The availability of event a in S however depends on whether or not the high agent engages in x. This lack of trace invariance for the L interface indicates that S can hardly be regarded as secure.

We must conclude that it is difficult to relate processes we would intuitively regard as secure to those that satisfy Jacob's condition for restricted information flow. Although the condition works well for some processes, it is too weak in general, since there are deterministic processes which are clearly insecure but are not rejected by restriction of information flow. In contrast to Jacob [Jac90, Jac91] we do not believe that the introduction of a security ordering $\preceq_L$ in addition to the standard refinement order is either desirable or necessary. We have shown in this thesis that standard development methods and refinement techniques can equally well be applied to secure systems, provided that the deterministic implementation satisfies the criterion of (transaction) independence.

Graham-Cumming [Gra92, Gra93] develops a theory for the formal development of secure systems within a CSP framework. His criterion for non-interference is equivalent to independence when applied to deterministic systems, but more liberal for non-deterministic systems some of which we would not regard as secure. Graham-Cumming's definition of non-interference is based on the following process equivalence, which describes under which condition two processes appear to be the same from the viewpoint of an agent interacting through interface L. Processes S and T are considered equivalent (from L's viewpoint) if they allow the same continuations of process behaviour when H actions are disallowed:

$$S \equiv_L T \quad \Leftrightarrow \quad \mathcal{S}_H(S) = \mathcal{S}_H(T) \ .$$

Although Graham-Cumming does not use the notion of process abstraction, stopping is in effect used to abstract from H behaviour. Processes which yield the same abstractions under stopping are regarded as equivalent. This equivalence is the basis of the definition of non-interference: interface H does not interfere with interface L if the process behaviour after trace t and after trace $t \upharpoonright L$ cannot be distinguished.

Definition 8.1.5 [Gra92, page 22] Let Σ be partitioned by H, L. Process P satisfies *non-interference* with respect to interface L if, and only if,

$$\forall\, t \in \text{TRACES}(P) \quad \bullet \quad P \ \underline{\mathsf{after}}\ t \equiv_L P \ \underline{\mathsf{after}}\ t \upharpoonright L \ ,$$

with the assumption that this condition is only applied when $t \upharpoonright L$ is indeed a trace of P (see [Gra92, page 24]).

Interface H of process P does not interfere with interface L according to Definition 8.1.5 precisely when the future L behaviour is the same irrespective of what the past H events have been. The agent with interface L cannot distinguish the trace t, which may include H events, from the purged trace $t \upharpoonright L$

because the above condition requires the process behaviour under abstraction to be identical.

Graham-Cumming's condition for non-interference is essentially equivalent to trace-invariance (Definition 3.1.1), since it requires the initial events available in the L interface at some point to depend on nothing but the sequence of L events which have occurred. As a consequence, the conditions of non-interference and independence agree whether or not a particular deterministic process is secure. The difference between these conditions becomes apparent when the L interface is non-deterministic. We illustrate this with an example from Graham-Cumming's thesis.

Example 8.1.6 [Gra92, pages 27–28] Let $B \mathrel{\widehat{=}} \{0, 1\}$ be the set of binary values. The example system consists of two binary variables which may be updated by the system's users, user U and user V. The update takes place through communication on channels

$$\underline{\text{channel}} \quad uUpdate, vUpdate \quad : B.B.B$$

The first component on each channel is to return the current value of the variable, the second another possible output (see below), and the third the input value used to update the variable. Either user can update only her respective variable. The overall security requirement is that either user must be non-interfering with the other user.

One possible implementation of the shared system is with process A_0 below. The process parameters u and v represent the current value of the binary variables. When a communication on the $uUpdate$ channel occurs, the old value u is output and the new value u' takes its place. The second component of the channel is underspecified; the value of b is chosen non-deterministically.

$$\begin{array}{rcl} A_0 & \widehat{=} & A(0, 0) \\ A(u, v) & \widehat{=} & \bigsqcap_{b:B} uUpdate.u.b?u' \to A(u', v) \\ & & \Box \\ & & \bigsqcap_{b:B} vUpdate.v.b?v' \to A(u, v') \ . \end{array}$$

In order to give a deterministic implementation it is necessary to remove the underspecification in the second component of the channels. Process C_0 below is a possible refinement, $A_0 \sqsubseteq C_0$, and is deterministic.

$$C_0 \quad \widehat{=} \quad C(0, 0)$$

$$
\begin{aligned}
C(u,v) \quad \widehat{=} \quad & uUpdate.u.v?u' \rightarrow C(u',v) \\
& \Box\ vUpdate.v.u?v' \rightarrow C(u,v')\ .
\end{aligned}
$$

In C_0 the value chosen for the second component is the value of the other variable, i.e. the value which the user is not supposed to know. Clearly, C_0 is an insecure process, and it satisfies neither Graham-Cumming's non-interference condition nor independence.

The difference appears when considering A_0. This process satisfies the non-interference condition, but we would not regard the process as secure: if A_0 were the system specification, we may obtain the insecure C_0 as its implementation, and there is no way to distinguish them! Processes with insecure refinements do not pass the independence condition, which is stronger than Graham-Cumming's. (End of Example)

In recent work, Schneider et al [Sch96, SS96] have applied the notion of process abstraction to model security properties beyond those concerned with non-interference and confidentiality in multi-level systems, in particular anonymity and authenticity. The property of anonymity refers to the secrecy of identity of an agent associated with particular actions or messages. In the context of a communications protocol, the originator or recipient of a message might wish to hide his association with the message in order to protect his identity.

Schneider and Sidiropoulos [SS96] define anonymity in terms of a renaming function f_A, which may also be regarded as a form of process abstraction. Assuming that a is some arbitrary event in set A of events, the function f_A is defined

$$f_A(x) \quad \widehat{=} \quad a\ , \quad \underline{\text{if}}\ x \in A \tag{8.3}$$

$$f_A(x) \quad \widehat{=} \quad x\ , \quad \underline{\text{if}}\ x \notin A \tag{8.4}$$

Given a process P, every occurrence of an event from A is being renamed to the single event a in process $f_A(P)$. The inverse function f_A^{-1} reverses the renaming: the process $f_A^{-1}(Q)$ makes every event from A available whenever a is possible in Q. Anonymity is guaranteed if the renaming using f_A followed by the inverse renaming produces a process whose traces are the same as those of the original P.

Definition 8.1.7 [SS96] Process P is said to be strongly anonymous on interface A if

$$f_A^{-1}(f_A(P)) \quad = \quad P \ ,$$

where equality is with respect to traces.

According to this definition, anonymity requires that process P makes all events available in interface A whenever any event is available. Anonymity on interface A means that all events from that set are indistinguishable; if one of them occurs then so could any, and there is no way to distinguish them from another interface.

In the context of a communication protocol, message authentication requires a high degree of assurance about the origin of a message. A message is regarded as "authentic" only if we can be certain that the message purporting to come from a particular source really did come from that source. In particular, it should not be possible to receive a message m from sender s without s having previously submitted m for transmission.

Schneider [Sch96] formalises the notion of authenticity using stopping as process abstraction on interfaces. Interface B authenticates interface A in process P if the stopping of $A \cup B$ has the same effect as the stopping of A alone.

Definition 8.1.8 [Sch96] Assume that A and B are disjoint subsets of Σ. Interface B authenticates A if, and only if,

$$\mathcal{S}_{A \cup B}(P) \quad = \quad \mathcal{S}_A(P) \ ,$$

where equality is with respect to traces.

According to this definition, it is not possible to observe an event in interface B without some A event having previously taken place. Thus, the observation of a B event provides assurance about some prior activity in interface A. This notion of authenticity appears useful in the context of message communication, where we require $receive$ events to authenticate the $send$ events of "honest" nodes in order to accept the message as coming from a reliable source. The approach is illustrated with a verification of a confidentiality property in an example communication network.

The definitions for anonymity and authenticity proposed by Schneider et al [Sch96, SS96] are formulated within the traces model of CSP, and thus apply primarily to deterministic systems. Despite this limitation, these definitions appear to capture notions of security beyond those discussed in this thesis, and illustrate how process abstractions can be usefully applied to describe a wider range of security properties. We regard their analysis as complementary to our framework which may open avenues for further research.

Probabilistic Systems

In seminal papers published in the 1940s, Shannon proposed a theory of information as an effective tool to model communication channels. Shannon's theory made it possible to model the effects of chance (probability) on the amount of information that can be conveyed over a channel. A large part of research in computer security has subsequently been based on or formulated in terms of Shannon-style information theory, such as the work of Sutherland [Sut86], Millen [Mil87], McLean [McL90], Wittbold and Johnson [WJ90], and Gray [Gr91, Gr92].

The key feature of information-theoretic and probabilistic modelling is the explicit representation of the effects of random choices which occur in many natural phenomena. For instance, the repeated flipping of an unbiased coin may be represented as

$$Flips \quad \widehat{=} \quad head \to Flips \;{}_{1/2}\!\oplus\; tail \to Flips \ .$$

The probabilistic choice operator ${}_{p}\!\oplus$ selects the left-hand argument with probability p, and the right-hand argument with probability $1-p$. Each individual choice is taken at random in a way which respects the indicated probability over a large number of experiments. For instance an observer of the behaviour of $Flips$ would observe roughly the same number of $head$ and $tail$ events over a long period of time.

Probabilistic choice should be distinguished from the standard choices $\Box$ and $\sqcap$ traditionally used in system specification and development. Internal choice $\sqcap$ represents the effects of some non-deterministic demon which can make the choice demonically either way, and each such possibility must be taken into account to ensure program correctness. For instance, in process

$$Flips' \quad \widehat{=} \quad head \to Flips' \sqcap tail \to Flips' \ .$$

either event is possible every time; we do not care how the choice is taken or what its outcome is. This form of underspecification is a weaker restriction on the choice than a truly random decision, so that internal choices may legitimately be implemented as probabilistic choices ${}_p\oplus$. Consequently we would expect $\mathit{Flips}' \sqsubseteq \mathit{Flips}$ in most models. A behavioural model may contain standard underspecified choices only, probabilistic choices only, or both types of choices. In the following we investigate the specification of non-interference and independence in a model which include both types of non-determinism.

Morgan et al [MMSS94, MM95] construct a probabilistic powerdomain over the failures-divergences model of CSP. The resulting model supports all standard CSP operators plus a probabilistic choice ${}_P\oplus$ operator. The distinguishing feature of the model is that all standard operators distribute over probabilistic choice, which means that ${}_P\oplus$ can generally be moved "to the outside". This distributivity property can be interpreted as meaning that probabilistic choices are always resolved prior to the occurrence of any events, and the results of these choices are known and may be exploited by the demon controlling the remaining non-deterministic choices. As a consequence, many intuitively secure processes fail to satisfy the independence condition, because the process abstraction is not maximal in the model.

Example 8.1.9 Assume that Σ is partitioned by $H \mathrel{\widehat{=}} \{x\}$ and $L \mathrel{\widehat{=}} \{a, b\}$. Consider the process

$$\begin{array}{rcl} P & \widehat{=} & (a \rightarrow \mathit{Stop}\ {}_{1/2}\oplus\ b \rightarrow \mathit{Stop}) \\ & & \Box\ x \rightarrow (a \rightarrow \mathit{Stop}\ {}_{1/2}\oplus\ b \rightarrow \mathit{Stop}) \end{array}$$

which offers a probabilistic choice between a and b in the low interface, irrespective of whether an x event occurs first. Since the ${}_{1/2}\oplus$ choices are resolved first, there is a 50% chance that the same low event will be picked (in which case L is independent), and a 50% chance that two different low events are selected (in which case L is not even trace-invariant). For the same reason, P is not semi-separable with respect to interface L. (End of Example)

A process should be regarded as secure in Morgan's model [MMSS94] only if it satisfies independence for all possible outcomes of its probabilistic choices. This requirement is very strict, thus identifying many intuitively secure processes as insecure. However, the model always errs on the safe side: there are no probabilistic processes satisfying independence which we would regard as insecure.

8.2 Summary and Conclusion

Security violations in a shared computer system take the form of unauthorized disclosure or corruption of data or information. It is the responsibility of software engineers and designers to be able to detect and prevent such violations, both during the initial stages of system development and in cases of fully operational systems where security concerns are raised. Security violations can be prevented by placing suitable constraints on the possible interactions between the system interfaces. The external agents communicating with the system through these interfaces may be the system's legitimate users, hostile intruders, or other computational processes which affect the way users perceive their communications with their interface.

In this thesis, we have employed the theory of Communicating Sequential Processes to develop a formal framework for describing and restricting the interaction between system interfaces. This framework is most suitably applied during the initial specification and design stages of secure system development. Refinement of the abstract specification may be carried out in the knowledge that the more concrete model will retain the original security properties, since our notions of independence and transaction independence are preserved under refinement.

At the heart of the formalism lies the use of abstraction functions, which make it possible to abstract parts of the computational behaviour of a process. There are several functions which can reasonably be regarded as abstraction functions, each having its own set of properties and serving a distinct purpose. Hiding is the abstraction function traditionally used in system development since it models the invisible occurrence of internal events. In a security specification it should be employed to abstract from instantaneous actions whose occurrence cannot be delayed, such as signal (output) events on a computer screen. Lazy abstraction and interleaving are appropriate for events under complete control of the system agent: the agent not only has power over the choice of next event, but also over the choice of engaging or delaying events. Although lazy abstraction and interleaving have generally quite different effects on processes, they share a remarkable number of algebraic properties and either of them may be used to verify independence. Since interleaving is one of standard CSP operators, a model checker such as FDR can be employed for security verification.

Abstraction functions share the property of introducing or increasing the amount of non-determinism when applied to a process. Our primary security criterion (independence) requires the absence of non-determinism after abstraction. This will ensure that the choices taken by high-level agents cannot perceivably affect the interface of other agents. A consequence of determinism under abstraction is that all possible refinements of a secure system are also guaranteed to be secure.

Apart from preservation of security under refinement, our development framework is fully compositional. This means that, given two secure components P and Q, the parallel composition $P \; |[A]| \; Q$ will also be secure. Compositionality is a crucial property of any framework of system development for several reasons. Firstly, it allows a complex distributed system model to be built up by composition of smaller components. The overall security properties of the complete system follow immediately from the security of its components. Secondly, compositionality opens the door for automated verification of security (and other) properties, because we can base assurance arguments on the checking of individual components. This is crucial in cases where the total number of states in the system is so large that an automated verification with a model checker such as FDR is not feasible.

There are several strategies a software engineer may follow during system development. One attractive strategy is to separate functional requirements from the security properties of the system. Suppose that the functional specification is given by $Func$ (which may itself be built up from smaller components) and the security properties by Sec, which imposes the required constraints on interfaces. The secure system is then given by a parallel composition

$$System \quad \widehat{=} \quad Func \; |[A]| \; Sec \; .$$

This approach is attractive from the viewpoint of separation of concerns: the functional and security requirements are described separately. The disadvantage lies in the fact that $Func$ is normally an insecure process (since it does not include the security constraints) and so one cannot rely on compositionality for the assurance argument. A possible alternative is to build the system from components all of which are secure

$$System \quad \widehat{=} \quad Sec_1 \; |[A_1]| \; (Sec_2 \; |[A_1]| \; Sec_3) \; .$$

This approach is illustrated in the study of the reader/writers problem (Chapter 5). In practice, it can be difficult to reconcile the functional requirements and security constraints within component processes. In this case it is most pragmatic to focus on the functional requirements first and consider the assurance argument for security later, as has been illustrated with the case studies in Chapters 6 and 7.

Non-determinism can serve as a powerful tool in system development by leaving unnecessary details unspecified, resulting in highly abstract specifications. It is essential to distinguish this type of non-determinism, also called underspecification, from that which arises from sources of random data or behaviour. A computational process may contain either or both types of non-determinism, and it is important to be able to distinguish between them in the modelling framework.

Independence as a security criterion is defined as the absence of underspecification under suitable abstraction functions. Being expressed in this way allows the condition to be applied in modelling frameworks which allow the observation of much more detailed types of process behaviour. In particular it could potentially be applied to systems whose behaviour includes random choices and/or real-time timing constraints.

It may be argued that independence is too strict a condition, excluding many non-deterministic systems of interest which we may "intuitively" regard as being secure. However, as long as we cannot make precise how non-deterministic choices are resolved in a particular implementation, we should not assume that these choices are resolved in a fair manner, with equal consideration to each alternative on a statistical basis. Underspecified choices are open to exploitation by an ingenious intruder as well as to the carelessness of a system implementor. Therefore it is best to assume that underspecified choices are resolved in the worst possible manner, to maximum effect within the range of possible interaction.

Thus our condition is conservative; erring on the safe side. If this is regarded as being too restrictive, then computational models need to be developed which distinguish between different types of non-determinism, and make precise which types are present in a particular system model. This is a challenging and fascinating area for further research.

Bibliography

[AL93] M. Abadi, L. Lamport (1993). "Composing Specifications", *ACM Transactions on Programming Languages and Systems*, vol 15, no 1.

[All91] P.G. Allen (1991). "A Comparison of Non-interference and Non-deducibility using CSP", *Proc. 1991 IEEE Computer Security Foundations Workshop*, pp 43–54, IEEE Computer Society Press.

[BLP73] D. Bell, L. LaPadula (1973). "Secure Computer Systems: Mathematical Foundations and Model", *MITRE Technical Report 2547*. Also available in *Journal of Computer Security*, vol 4, nos 2,3, pp 239–263.

[BY94] W.R. Bevier, W.D. Young (1994). "A State-based Approach to Non-interference", *Journal of Computer Security 3(1)*, pp 55–70, IOS Press.

[BHG87] P. Bernstein, V. Hadzilacos, N. Goodman (1987). *Concurrency Control and Recovery in Database Systems*, Addison-Wesley.

[BL93] A.J. Bernstein, P.M. Lewis (1993). *Concurrency in Programming and Database Systems*, Jones and Bartlett Publishers, Inc.

[BGRS91] Y. Breitbart, D. Georgakopoulos, M. Rusinkiewicz, A. Silberschatz (1991). "On Rigorous Transaction Scheduling", *IEEE Transactions on Software Engineering*, pp 954–960, vol 17, no 9.

[Bro91] R. Brown (1991). "The Turing Test and Non-Information Flow", *Proc. 1991 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, pp 373–385.

[BR84] S.D. Brookes, A.W. Roscoe (1984). "An Improved Failures Model for Communicating Sequential Processes", *Proceedings NSF-SERC Seminar on Concurrency*, LNCS 124, Springer Verlag.

[BD97] G. Bruce, R. Dempsey (1997). *Security in Distributed Computing*, Hewlett-Packard Professional Books, Prentice-Hall.

[Bur89] R. Burnham (1989). *The Specification of Security in Distributed Computing Systems*, MSc Thesis, University of Oxford.

[Col94] R. Collinson (1994). "A Critical Look at Functional Specifications", *Proc. FME '94 Symposium*, Springer Verlag LNCS 873.

[Den76] D.E. Denning (1976). "A Lattice Model of Secure Information Flow", *Comm. ACM 19(5)*, pp 236–242.

[FLR77] R.J. Feiertag, K.N. Levitt, L. Robinson (1977). "Proving Multilevel Security of a System Design", *Proc. 6th ACM Symp. Operating System Principles*, pp 57–65.

[FG95] R. Focardi, R. Gorrieri (1995). "A Classification of Security Properties for Process Algebras", *Journal of Computer Security*, vol 3, no 1, pp 5–33.

[Fol87] S. Foley (1987). "A Universal Theory of Information Flow", *Proc. 1987 IEEE Symposium on Security and Privacy*, pp 116–122, IEEE Computer Society Press.

[Fol92] S. Foley (1992). "Aggregation and Separation as Non-interference Properties", *Journal of Computer Security*, vol 1, no 2.

[FSE94] Formal Systems Europe (1994). *Failures-Divergences Refinement*, Manual.

[GGJ96] P. Gardiner, M. Goldsmith, D. Jackson (1996). "Security Modelling in CSP and FDR", *Deliverable under DRA contract CSM/106*, Formal Systems Europe.

[Gas88] M. Gasser (1988). *Building a Secure Computer System*, Van Nostrand Reinhold.

[Gra92] J. Graham-Cumming (1992). *The Formal Development of Secure Systems*, DPhil Thesis, University of Oxford.

[Gra93] J. Graham-Cumming (1993). "Laws of Non-interference in CSP", *Journal of Computer Security*, vol 2, no 1, pp 37–52.

[GM82] J. Goguen, J. Meseguer (1982). "Security policy and security models", *Proc. 1982 IEEE Symposium on Security and Privacy*, pp 11–20, IEEE Computer Society Press.

[GM84] J. Goguen, J. Meseguer (1984). "Unwinding and inference control", *Proc. 1984 IEEE Symposium on Security and Privacy*, pp 75–86, IEEE Computer Society Press.

[Gr91] J.W. Gray III (1991). "Probabilistic Interference", *Proc. 1991 IEEE Symposium on Security and Privacy*, pp 170–179, IEEE Computer Society Press.

[Gr92] J.W. Gray III (1992). "Towards a Mathematical Foundation of Information Flow Security", *Proc. 1992 IEEE Symposium on Security and Privacy*, pp 21–35, IEEE Computer Society Press.

[GS92] J.W. Gray III, P.F. Syverson (1992). "A Logical Approach to Multilevel Security of Probabilistic Systems", *Proc. 1992 IEEE Symposium on Security and Privacy*, pp 164–176, IEEE Computer Society Press.

[HKMY87] J.T. Haigh, R.A. Kemmerer, J. McHugh, W.D. Young (1987). "An Experience using Two Covert Channel Analysis Techniques on a Real System Design", *IEEE Transactions on Software Engineering*, vol 13, no 2.

[HY86] J.T. Haigh, W.D. Young (1986). "Extending the non-interference version of MLS for SAT", *Proc. 1986 Symposium on Security and Privacy*, pp 232–239, IEEE Computer Society Press.

[Hoa85] C.A.R. Hoare (1985). *Communicating Sequential Processes*, Prentice Hall.

[Jac87] J.L. Jacob (1987). *On Shared Systems*, DPhil Thesis, University of Oxford.

[Jac90] J.L. Jacob (1990). "Specifying Security Properties", in C. A. R. Hoare (ed), *Developments in Concurrency and Communication*, pp 221–237, ACM Press.

[Jac90b] J.L. Jacob (1990). "Separability and the detection of hidden channels", *Information Processing Letters 34(1)*, pp 27–29.

[Jac91] J.L. Jacob (1991). "The Varieties of Refinement", *Proc. 4th Refinement Workshop*, J.M. Morris and R.C. Shaw (editors), Springer Verlag.

[JT88] D.M. Johnson, F.J. Thayer (1988). "Security and the Composition of Machines", *Proc. 1988 Security Foundations Workshop*.

[Lam73] B. Lampson (1973). "A Note on the Confinement Problem", *Communications of the ACM*, vol 16, no 10, pp 613–615.

[Lap85] J.-C. Laprie (1985). "Dependable Computing and Fault-tolerance: Concepts and Terminology", *Fault-Tolerant Computing Symposium 15*, pp 2–11, IEEE Computer Society.

[Low96] G. Lowe (1996). "Representing Non-determinism and Probabilistic Behaviour in Reactive Processes", draft paper. A preliminary version has been published as Technical Report PRG-TR-11-93, Oxford University Computing Laboratory.

[KTS93] T.F. Keefe, W.T. Tsai, J. Srivastava (1993). "Database Concurrency Control in Multilevel Secure Database Managment Systems", *IEEE Transactions on Knowledge and Data Engineering*, vol 5, no 6, pp 1039–1055.

[McC88] D. McCullough (1988). "Non-interference and the Composability of Security Properties", *Proc. 1988 Symposium on Security and Privacy*, pp 177–186, IEEE Computer Society Press.

[McL90] J. McLean (1990). "Security Models and Information Flow", *Proc. 1990 Symposium on Security and Privacy*, pp 180–187, IEEE Computer Society Press.

[McL94] J. McLean (1994). "A General Theory of Composition for Trace Sets closed under selective Interleaving Functions", *Proc. 1994 Symposium on Security and Privacy*, IEEE Computer Society Press.

[McL94b] J. McLean (1994). "Security Models", *Encyclopedia of Software Engineering*, John Marciniak (ed.), Wiley Press.

[Mea95] C. Meadows (1995). "Report on Computer Security Track for ONR Workshop on High Assurance Computing", Naval Research Laboratory, Washington DC, USA.

[Mil76] J.K. Millen (1976). "Security Kernel Validation in Practice", *Communications of the ACM*, vol 19, no 5, pp 243–250.

[Mil87] J.K. Millen (1987). "Covert Channel Capacity", *Proc. 1987 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, pp 60–66.

[Mil95] J.K. Millen (1995). "Unwinding Forward Correctability", *Journal of Computer Security*, vol 3, no 1, pp 35–54.

[MM95] A. McIver, C. Morgan (1995). "Probabilistic Determinism", Technical Report PRG-TR-13-95, Oxford University Computing Laboratory.

[MMSS94] C. Morgan, A. McIver, K. Seidel and J.W. Sanders (1994). "Refinement-oriented probability for CSP", Technical Report PRG-TR-12-94, Oxford University Computing Laboratory.

[Mor94] C. Morgan (1994). *Programming from Specifications*, 2nd edition, Prentice-Hall International.

[MC92] I.S. Moskowitz, O.L. Costich (1992). *Proc. 1995 IEEE Computer Security Foundations Workshop*, pp. 2–8, IEEE Computer Society Press.

[O'H93] C. O'Halloran (1993). *Category Theory and Information Flow Applied to Computer Security*, DPhil Thesis, University of Oxford.

[Pap86] C. Papadimitriou (1986). *The Theory of Concurrency Control*, Computer Science Press, 1986.

[Per84] C. Perrow (1984). *Normal Accidents: Living with High Risk Technologies*, Basic Books, Ney York.

[PW] B. Pfitzmann, M. Waidner. "A general framework for formal notions of secure systems". Draft paper.

[Pfl97] C.P. Pfleeger (1997). *Security in Computing*, second edition, Prentice-Hall.

[RK79] D.P. Reed, R.K. Kanodia (1993). "Syncronization with event counts and sequencers", *Communications of the ACM*, volume 22, no. 2, February 1979.

[Ros93] A.W. Roscoe (1993). "Unbounded Non-determinism in CSP", *Journal of Logic and Computation* **3**.

[Ros94] A.W. Roscoe (1994). "Model Checking CSP", in A. W. Roscoe (ed) *A Classical Mind*, Prentice Hall.

[Ros95] A.W. Roscoe (1995). "CSP and Determinism in Security Modelling", *Proc. 1995 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press.

[RW95] A.W. Roscoe, L. Wulf (1995). "Composing and Decomposing Systems under Security Properties", *Proc. 1995 IEEE Computer Security Foundations Workshop*, IEEE Computer Society Press.

[RWW94] A.W. Roscoe, J.C.P. Woodcock, L. Wulf (1994). "Non-interference through Determinism", *Proc. Third European Symposium on Research in Computer Security* (ESORICS-94), Springer LNCS 875.

[RWW96] A.W. Roscoe, J.C.P. Woodcock, L. Wulf (1996). "Non-interference through Determinism", *Journal of Computer Security*, vol 4, no 1, pp 27–53.

[Ros97] A.W. Roscoe (1997). *The Theory and Practice of Concurrency*, draft chapters of book in preparation, Prentice-Hall International.

[Rus81] J. Rushby (1981). "Design and Verification of Secure Systems", from the *8th ACM Symposium on Operating System Principles*, Pacific Grove, California, 14–16, December 1981.

[Rus85] J. Rushby (1985). *The SRI Security Model*, Computer Science Laboratory, SRI International.

[Rus92] J. Rushby (1992). *Non-interference, transitivity, and channel-control security policies*, Technical Report SRI-CSL-92-02, SRI Computer Science Laboratory.

[Rus94] J. Rushby (1994). "Critical System Properties: Survey and Taxonomy", *Reliability Engineering and System Safety*, Vol. 43, No. 2, pp 189–219.

[Rya91] P.Y.A. Ryan (1991). "A CSP Formulation of Non-interference and Unwinding", *Cipher* Winter 1991, pp 19–27, IEEE Computer Society Press.

[San92] R.S. Sandhu (1992). "On some Research Issues in Multi-level Database Systems", *5th RADC Workshop on Multi-level Database Security*, Fredonia, PA.

[Sch96] S. Schneider (1996). "Security Properties and CSP" *Proc. 1996 Symposium on Security and Privacy*, IEEE Computer Society Press.

[SS96] S. Schneider, A. Sidiropoulos (1996). "CSP and Anonymity", draft paper.

[Sha48] C.E. Shannon (1948). "A Mathematical Theory of Communication", *Bell Sys. Tech. Journal* 27, pp 379–423 and 623–656.

[Sha49] C.E. Shannon (1949). "Communication Theory and Secrecy Systems", *Bell Sys. Tech. Journal* 28, pp 657–715.

[Sim96] A.C. Simpson (1996). *Safety through Security*, DPhil Thesis, University of Oxford.

[Smi92] G. Smith (1992). "Formal Methods", *Computer Security Reference Book*, edited by K.M. Jackson and J. Hruska, Butterworth Heinemann.

[Sut86] D. Sutherland (1986). "A Model of Information", *Proceedings of the 9th National Computer Security Conference*, pp 175–183.

[Wel88] D. Welsh (1988). *Codes and Cryptography*, Oxford Science Publications, Clarendon Press.

[WJ90] J.T. Wittbold, D.M. Johnson (1990). "Information Flow in Nondeterministic Systems", *Proc. 1990 Symposium on Security and Privacy*, pp 144–161, IEEE Computer Society Press.

[Woo90] J.C.P Woodcock (1990). "Transaction Processing Primitives and CSP", in C. A. R. Hoare (ed), *Developments in Concurrency and Communication*, pp 239–265, ACM Press.

Appendix A

Reference

This chapter provides a reference for the syntax and semantics of Communicating Sequential Processes. A brief overview on the syntax of CSP and our notational conventions is given in the first section. Sections A.2 and A.3 provide a summary of the failures-divergences model and the infinite traces model, respectively. The primary references for the notation and definitions in this Appendix are Brookes and Roscoe's paper [BR84] and Hoare's book [Hoa85] based on the failures-divergences model, Formal System's FDR manual [FSE94], and Roscoe's forthcoming book on the theory and practice of concurrency [Ros97]. Concerning process alphabets, we have adopted the modern "untyped" style (as used in FDR) which does not require each definition of a process to include its alphabet. We use the term "alphabet" informally to refer to the communications a process can engage in, and define αP when it is required in an interface parallel composition of process P with other processes.

A.1 Notation

In this section, a summary of CSP's syntax and notational conventions as used in this thesis will be given.

Channels and Events

<u>channel</u> $c : A.B$	Declaration of channel c of (composite) type $A.B$
$c.i.j$	Event $c.i.j$ on channel c
$c?i?j$	Input event, with an external choice of values for i and j
$c?i : A$	Input event, with an external choice from A for i
$\{\| c, d \|\}$	Set of events on channels c and d

Sets

$\emptyset$	Empty set
$\{1, 2, 3\}$	Set comprehension
$a \in A$	Membership of a in set A
$A \cup B$	Union of sets A and B
$A \cap B$	Intersection of sets A and B
$A \setminus B$	Difference of sets A and B
<u>card</u>(A)	Cardinality of set A
$\mathbb{P}\, A$	Powerset (set of subsets) of set A

Sequences (Traces)

$\langle\rangle$	Empty trace
<u>last</u> s	Last item in (non-empty) trace s
$s \frown t$	Concatenation of traces s and t
$s \downarrow A$	Count of events A in trace s
$s \upharpoonright A$	Restriction (projection) of trace s to events A
$s \setminus H$	Hiding (removal) of events H from trace s
$s \mid\mid\mid t$	Set of interleavings from traces s and t
$s \mid[A]\mid t$	Set of interleavings from traces s and t with synchronization on events A

Sequential Operators

$a \rightarrow P$	Prefixing of event a to process P
$P \,\square\, Q$	External choice between processes P and Q
$P \sqcap Q$	Internal choice between processes P and Q
$\square_{i:S}\, P$	Indexed external choice of i from set S in process P
$\sqcap_{i:S}\, P$	Indexed internal choice of i from set S in process P
$P\,[\![a \leftarrow b]\!]$	Renaming, with event a becoming b
$P \setminus C$	Hiding (concealment) of set C
<u>if</u> C <u>then</u> P <u>else</u> Q	Conditional choice between P and Q on boolean C

The formal semantics of standard sequential operators is given under subsection "Sequential Processes" in Section A.2.

Parallel Operators

$P \mid[S]\mid Q$	Parallel composition of P and Q with synchronization on events S
$P\,[A \| B]\,Q$	Interface parallel composition of P with interface A and Q with interface B
$\|_{i:S}\,[A_i]P_i$	Indexed parallel composition of processes P_i with interfaces A_i
$P \mid\mid\mid Q$	Interleaving (no synchronization of events)

The formal semantics of standard parallel operators is given under subsection "Parallel Processes" in Section A.2.

Process Abstractions

$\mathcal{S}_H(P)$	Stopping; Definition 2.2.1
$\mathcal{H}_H(P)$	Hiding; Definition 2.2.5
$\mathcal{P}^*_H(P)$	Strict projection; Definition 2.2.8
$\mathcal{L}_H(P)$	Lazy abstraction; Definition 2.2.18
$\mathcal{I}_H(P)$	Interleaving; Definition 2.2.32

Security

$H \not\rightarrowtail L :: P$	Independence of L from H in process P
$H \setminus\!\setminus K \not\rightarrowtail L :: P$	Independence of L from H under completions K in P (transaction security)

A.2 Failures-Divergences Model of CSP

The failures-divergences model [BR84, Hoa85] is defined with respect to some non-empty, finite alphabet Σ of communication events. Given such a (generally understood) set Σ, the model consists of a all pairs (F, D) such that

$$F \subseteq \Sigma^* \times \mathbb{P}\Sigma \quad \wedge \quad D \subseteq \Sigma^*$$

which satisfy the following axioms.

F1. The set of traces is non-empty and prefix-closed

$$F \neq \varnothing \quad \wedge \quad (s \frown t, X) \in F \Rightarrow (s, \varnothing) \in F \ .$$

F2. Any subset of a refusal can also be refused

$$(t, X) \in F \wedge Y \subseteq X \quad \Rightarrow \quad (t, Y) \in F \ .$$

F3. If a trace t cannot be continued with any event in set Y, then Y can also be refused after t.

$$(t, X) \in F \wedge \forall\, a \in Y \bullet (t \frown \langle a \rangle, \varnothing) \notin F \quad \Rightarrow \quad (t, X \cup Y) \in F \ .$$

D1. The set of divergences is suffix-closed

$$s \in D \quad \Rightarrow \quad s \frown t \in D \ .$$

D2. Every divergent trace contributes arbitrary failures

$$s \in D \quad \Rightarrow \quad (s \frown t, X) \in F \ .$$

Let $P \mathrel{\hat{=}} (F, D)$ be a process as defined above. Then let

$$\begin{aligned}
\textsc{Fails}(P) &\mathrel{\hat{=}} F && \text{(A.1)}\\
\textsc{Divs}(P) &\mathrel{\hat{=}} D && \text{(A.2)}\\
\textsc{Traces}(P) &\mathrel{\hat{=}} \{\, t \mid (t, \emptyset) \in F \,\} && \text{(A.3)}\\
(P)^0 &\mathrel{\hat{=}} \{\, a \mid \langle a \rangle \in \textsc{Traces}(P) \,\} && \text{(A.4)}\\
\textsc{Refs}(P) &\mathrel{\hat{=}} \{\, X \mid (\langle\rangle, X) \in F \,\} && \text{(A.5)}
\end{aligned}$$

For $s \in \textsc{Traces}(P)$, the behaviour of P after engaging in trace s is given by

$$\begin{aligned}
\textsc{Fails}(P \ \underline{\mathsf{after}}\ s) &\mathrel{\hat{=}} \{\, (t, X) \mid (s \frown t, X) \in \textsc{Fails}(P) \,\}\\
\textsc{Divs}(P \ \underline{\mathsf{after}}\ s) &\mathrel{\hat{=}} \{\, t \mid s \frown t \in \textsc{Divs}(P) \,\}
\end{aligned}$$

When the behaviours of a process Q are a subset of the behaviours of another process P we say that P is refined by Q. This relation is written $P \sqsubseteq Q$ and formally defined by

$$P \sqsubseteq Q \quad \Leftrightarrow \quad \textsc{Fails}(P) \supseteq \textsc{Fails}(Q) \wedge \textsc{Divs}(P) \supseteq \textsc{Divs}(Q) \ .$$

Two processes are equal if either refines the other, which means that the processes agree on their behaviours:

$$P = Q \quad \Leftrightarrow \quad \textsc{Fails}(P) = \textsc{Fails}(Q) \wedge \textsc{Divs}(P) = \textsc{Divs}(Q) \ .$$

A subscript may be added to emphasize at which level of modelling a refinement holds: $P \sqsubseteq_T Q$ with regard to trace refinement, $P \sqsubseteq_{FD} Q$ with regard to failures-divergences refinement, and $P =_{FD} Q$ with regard to failures-divergences equality.

A process P is regarded as *deterministic* if, and only if, (i) it is free of divergence; and (ii) it satisfies

$$(t, \{a\}) \in \textsc{Fails}(P) \quad \Leftrightarrow \quad (P \ \underline{\mathsf{after}}\ t)^0 \cap \{a\} = \emptyset \ ,$$

for all traces t, that is, there never is a choice between accepting and refusing an event.

Commonly used special processes include $Stop$, Run_H and $Chaos_H$, all of which are free of divergence. Process $Stop$ never engages any event, and thus its trace is empty

$$\textsc{Fails}(Stop) \quad \mathrel{\hat{=}} \quad \{(\langle\rangle, X) \mid X \subseteq \Sigma\} \ .$$

Run_A is the process which can engage any event from A at any time; its traces are equal to A^*. Since Run_A offers an external choice of next event, the process can only refuse events outside A.

$$Run_A \quad \widehat{=} \quad \Box_{a:A}\, a \rightarrow Run_A \ .$$

$Chaos_A$ can also engage any trace of events from A and therefore has the same traces as Run_A, namely A^*.

$$Chaos_A \quad \widehat{=} \quad Stop \sqcap (\Box_{a:A}\, a \rightarrow Chaos_A) \ .$$

However, $Chaos_A$ can also deadlock at any time (when the internal choice is taken in favour of $Stop$), and thus refuse to make progress. Consequently the process can refuse any subset of events from Σ.

Sequential Processes

The most commonly used operators for the definition of sequential processes include event prefixing, choice, and hiding.

The expression $a \rightarrow P$ denotes a process which engages event a and then behaves like process P. Since only event a is initially accepted, all sets excluding a may be refused

$$\begin{aligned}
\textsc{Fails}(a \rightarrow P) \quad &\widehat{=} \quad \{(\langle\rangle, X) \mid a \notin X\} \cup \\
&\qquad \{(\langle a\rangle \frown s, X) \mid s \in \textsc{Fails}(P)\} \\
\textsc{Divs}(a \rightarrow P) \quad &\widehat{=} \quad \{\langle a\rangle \frown s \mid s \in \textsc{Divs}(P)\} \ .
\end{aligned}$$

The expression $P \,\Box\, Q$ denotes the external (deterministic) choice between processes P and Q. Since it is the environment which makes the decision between the alternatives, process $P \,\Box\, Q$ can refuse a set of events only if both components do.

$$\begin{aligned}
\textsc{Fails}(P \,\Box\, Q) \quad &\widehat{=} \quad \{\, (\langle\rangle, X) \mid (\langle\rangle, X) \in \textsc{Fails}(P) \cap \textsc{Fails}(Q) \,\} \cup \\
&\qquad \{\, (s, X) \mid s \neq \langle\rangle \wedge (s, X) \in \textsc{Fails}(P) \cup \textsc{Fails}(Q) \,\} \cup \\
&\qquad \{\, (s, X) \mid s \in \textsc{Divs}(P \,\Box\, Q) \,\} \\
\textsc{Divs}(P \,\Box\, Q) \quad &\widehat{=} \quad \textsc{Divs}(P) \cup \textsc{Divs}(Q) \ .
\end{aligned}$$

The expression $P \sqcap Q$ denotes the internal choice between processes P and Q. The choice between those alternatives is non-deterministic and cannot be controlled by the environment. Therefore, a set of events initially refused by either process may also be refused by the choice, and any subsequent observation could be that of either process.

$$\begin{array}{rcl} \textsc{Fails}(P \sqcap Q) & \mathrel{\widehat{=}} & \textsc{Fails}(P) \cup \textsc{Fails}(Q) \\ \textsc{Divs}(P \sqcap Q) & \mathrel{\widehat{=}} & \textsc{Divs}(P) \cup \textsc{Divs}(Q) \ . \end{array}$$

The expression $P \setminus C$ denotes the process with the set C of events concealed, i.e. removed from the process trace. The semantics of the hiding operator in the failures-divergences model is given by

$$\begin{array}{rcl} \textsc{Fails}(P \setminus C) & \mathrel{\widehat{=}} & \{\, (s \setminus C, X) \mid (s, X \cup C) \in \textsc{Fails}(P) \,\} \ \cup \\ & & \{\, (s, X) \mid s \in \textsc{Divs}(P \setminus C) \,\} \\ \textsc{Divs}(P \setminus C) & \mathrel{\widehat{=}} & \{\, (s \setminus C) \frown t \mid s \in \textsc{Divs}(P) \,\} \ \cup \\ & & \{\, (s \setminus C) \frown t \mid (\forall\, n \in \mathbb{N} \bullet (\exists\, u \in C^* \bullet \\ & & \qquad\qquad \#u > n \wedge s \frown u \in \textsc{Traces}(P))) \,\} \ . \end{array}$$

This last clause shows that hiding C in P results in divergence whenever P can engage an unbounded number of consecutive events from C. Hiding distributes over parallel composition, provided that none of the events being concealed need to be synchronized between the two processes involved in the composition. If $S \cap C = \emptyset$ then

$$(P \,|[S]|\, Q) \setminus C \;=\; (P \setminus C) \,|[S]|\, (Q \setminus C) \ , \tag{A.6}$$

and if $A \cap B \cap C = \emptyset$, then

$$(P \,[A \| B]\, Q) \setminus C \;=\; P \setminus (C \cap A) \,[A \| B]\, Q \setminus (C \cap B) \ . \tag{A.7}$$

Parallel Processes

The set of all possible interleavings of traces s and t is $s \mid\mid\mid t$, defined as

$$\begin{array}{rcl} s \mid\mid\mid t & \mathrel{\widehat{=}} & t \mid\mid\mid s \\ s \mid\mid\mid \langle\rangle & \mathrel{\widehat{=}} & \{s\} \\ \langle a \rangle \frown s \mid\mid\mid \langle b \rangle \frown t & \mathrel{\widehat{=}} & \{\, \langle a \rangle \frown r \mid r \in s \mid\mid\mid \langle b \rangle \frown t \,\} \cup \\ & & \{\, \langle b \rangle \frown r \mid r \in \langle a \rangle \frown s \mid\mid\mid t \,\} \end{array}$$

The expression $P \mid\mid\mid Q$ denotes the composition of processes P and Q without synchronization. Each component may progress independently, and thus events from either process may occur in a trace of $P \mid\mid\mid Q$ arbitrarily interleaved. The semantics of $\mid\mid\mid$ interleaving is given by

$$\begin{array}{rcl}
\mathrm{F\scriptsize{AILS}}(P \mid\mid\mid Q) & \widehat{=} & \{ (r, X) \mid \exists\, s, t \bullet r \in s \mid\mid\mid t \,\wedge \\
& & \qquad (s, X) \in \mathrm{F\scriptsize{AILS}}(P) \wedge (t, X) \in \mathrm{F\scriptsize{AILS}}(Q) \} \,\cup \\
& & \{ (u, X) \mid u \in \mathrm{D\scriptsize{IVS}}(P \mid\mid\mid Q) \wedge X \subseteq \Sigma \} \\
\mathrm{D\scriptsize{IVS}}(P \mid\mid\mid Q) & \widehat{=} & \{ r \frown u \mid \exists\, s, t \bullet r \in s \mid\mid\mid t \,\wedge \\
& & \qquad (s \in \mathrm{D\scriptsize{IVS}}(P) \wedge t \in \mathrm{T\scriptsize{RACES}}(Q)) \,\vee \\
& & \qquad (s \in \mathrm{T\scriptsize{RACES}}(P) \wedge t \in \mathrm{D\scriptsize{IVS}}(Q)) \} \;.
\end{array}$$

The set of all possible interleavings of traces s and t which agree on events S is $s \mid[S]\mid t$, defined as

$$\begin{array}{rcl}
s \mid[S]\mid t & \widehat{=} & t \mid[S]\mid s \\
s \mid[S]\mid \langle\rangle & \widehat{=} & \{s\} \;,\; \underline{\text{if}}\; s \upharpoonright S = \langle\rangle \\
\langle a\rangle \frown s \mid[S]\mid \langle a\rangle \frown t & \widehat{=} & \{ \langle a\rangle \frown r \mid r \in s \mid[S]\mid t \} \;,\; \underline{\text{if}}\; a \in S \\
\langle a\rangle \frown s \mid[S]\mid \langle b\rangle \frown t & \widehat{=} & \{ \langle b\rangle \frown r \mid r \in \langle a\rangle \frown s \mid[S]\mid t \} \;,\; \underline{\text{if}}\; a \in S \wedge b \notin S \\
\langle a\rangle \frown s \mid[S]\mid \langle b\rangle \frown t & \widehat{=} & \{ \langle a\rangle \frown r \mid r \in s \mid[S]\mid \langle b\rangle \frown t \} \,\cup \\
& & \{ \langle b\rangle \frown r \mid r \in \langle a\rangle \frown s \mid[S]\mid t \} \;,\; \underline{\text{if}}\; a, b \notin S
\end{array}$$

The expression $P \mid[S]\mid Q$ denotes the parallel composition of processes P and Q with synchronization on events in set S. Events in the synchronization set cannot occur without agreement of both component processes

$$\begin{array}{rcl}
\mathrm{F\scriptsize{AILS}}(P \mid[S]\mid Q) & \widehat{=} & \{ (r, (X \cap Y) \cup (X \cap S) \cup (Y \cap S)) \mid \\
& & \qquad \exists\, s, t \bullet r \in s \mid[S]\mid t \,\wedge \\
& & \qquad (s, X) \in \mathrm{F\scriptsize{AILS}}(P) \wedge (t, Y) \in \mathrm{F\scriptsize{AILS}}(Q) \} \,\cup \\
& & \{ (u, X) \mid u \in \mathrm{D\scriptsize{IVS}}(P \mid[S]\mid Q) \wedge X \subseteq \Sigma \} \\
\mathrm{D\scriptsize{IVS}}(P \mid[S]\mid Q) & \widehat{=} & \{ r \frown u \mid \exists\, s, t \bullet r \in s \mid[S]\mid t \,\wedge \\
& & \qquad (s \in \mathrm{D\scriptsize{IVS}}(P) \wedge t \in \mathrm{T\scriptsize{RACES}}(Q)) \,\vee \\
& & \qquad (s \in \mathrm{T\scriptsize{RACES}}(P) \wedge t \in \mathrm{D\scriptsize{IVS}}(Q)) \} \;.
\end{array}$$

The shared parallel operator $\mid[S]\mid$ distributes through $\sqcap$ choice and is associative.

$$P \mid[S]\mid (Q \sqcap R) \;=\; (P \mid[S]\mid Q) \sqcap (P \mid[S]\mid R) \qquad \text{(A.8)}$$
$$P \mid[S]\mid (Q \mid[S]\mid R) \;=\; (P \mid[S]\mid Q) \mid[S]\mid R \qquad \text{(A.9)}$$

The expression $P\,[{\scriptstyle A}\|{\scriptstyle B}]\,Q$ denotes the parallel composition of processes P and Q with explicitly given interfaces. In this composition, process P cannot communicate outside interface A whereas Q cannot communicate outside interface B, and events $A \cap B$ common to both interfaces are synchronized. Although the semantics of $P\,[{\scriptstyle A}\|{\scriptstyle B}]\,Q$ is determined by that of the shared parallel operator $|[{\scriptstyle S}]|$, we will state the operator's definition here nevertheless for matters of completeness.

$$\begin{array}{rcl}
\textsc{Fails}(P\,[{\scriptstyle A}\|{\scriptstyle B}]\,Q) & \widehat{=} & \{\,(s, X \cup Y \cup Z) \mid s \in (A \cup B)^* \wedge \\
& & \quad X \subseteq A \wedge Y \subseteq B \wedge Z \cap (A \cup B) = \emptyset \wedge \\
& & \quad (s \upharpoonright A, X) \in \textsc{Fails}(P) \wedge \\
& & \quad (s \upharpoonright B, Y) \in \textsc{Traces}(Q))\,\} \\
& & \cup \{\,(u, X) \mid u \in \textsc{Divs}(P\,[{\scriptstyle A}\|{\scriptstyle B}]\,Q)\,\} \\
\textsc{Divs}(P\,[{\scriptstyle A}\|{\scriptstyle B}]\,Q) & \widehat{=} & \{\,r \frown u \mid r \in (A \cup B)^* \wedge \\
& & \quad (r \upharpoonright A \in \textsc{Divs}(P) \wedge r \upharpoonright B \in \textsc{Traces}(Q)) \vee \\
& & \quad (r \upharpoonright A \in \textsc{Traces}(P) \wedge r \upharpoonright B \in \textsc{Divs}(Q))\,\}
\end{array}$$

Thus a trace of $P\,[{\scriptstyle A}\|{\scriptstyle B}]\,Q$ is made up of events from A and B; events outside $A \cup B$ cannot occur. Restricting such a trace s to A produces a trace $s \upharpoonright A$ of P, while a similar restriction produces a trace $s \upharpoonright B$ of Q. A subset of A is refused if process P refuses it while Q may choose whether or not to perform events in B. Events in the shared interface $A \cap B$ can be refused by either component process.

Determinism is preserved under parallel composition with the interface parallel operator. That is, if processes P and Q are both deterministic processes, then the interface parallel composition will also be deterministic:

Proposition A.2.1 If P and Q are both deterministic, then so is $P\,[{\scriptstyle A}\|{\scriptstyle B}]\,Q$.

The relationship between the parallel operators can be summarized as follows. Interleaving has the same effect has composition without synchronization

$$P \,|||\, Q \quad = \quad P \,|[\emptyset]|\, Q \;, \tag{A.10}$$

and interface parallel has the same effect as shared parallel provided both component processes communicate within their interfaces

$$P\,[{\scriptstyle A}\|{\scriptstyle B}]\,Q \quad = \quad P \,|[{\scriptstyle A \cap B}]|\, Q \tag{A.11}$$

if P communicates within A and Q communicates within B.

A.3 Infinite Traces

In the infinite traces model of CSP, introduced by Roscoe [Ros93], processes are represented by triples (F, D, I). The F and D components identify the failures and divergences of the process, and satisfy Axioms **F1**to **D2** of Section A.2. The third component I gives the set of infinite traces the process can perform. The infinite traces satisfy the following additional axioms.

I1. The prefix of an infinite trace is a itself a (finite) trace

$$s \frown t \in I \quad \Rightarrow \quad (s, \emptyset) \in F \ .$$

I2. A divergent trace s generates arbitrary infinite traces

$$s \in D \quad \Rightarrow \quad s \frown t \in I \ .$$

I3. Every process is identical to the non-deterministic composition of its pre-deterministic (i.e. deterministic until divergent) implementations. This axiom ensures that the set of infinite traces includes all those that, on the basis of its failure set, the process could be forced to engage.

The infinite traces I of a process P will also be referred to as $\textsc{Infs}(P)$. Since this set is an explicit representation of the process' infinite traces, it will determine the set of divergent traces under hiding:

$$\begin{array}{rcl}
\textsc{Fails}(P \setminus A) & \widehat{=} & \{ (s \setminus A, X) \mid (s, X \cup A) \in \textsc{Fails}(P) \} \cup \\
& & \{ (s, X) \mid s \in \textsc{Divs}(P \setminus A) \} \\
\textsc{Divs}(P \setminus A) & \widehat{=} & \{ (s \setminus A) \frown t \mid s \in \textsc{Infs}(P) \wedge s \setminus A \text{ finite} \} \cup \\
& & \{ (s \setminus A) \frown t \mid s \in \textsc{Divs}(P) \}
\end{array}$$

This contrasts with the definition given on page 152, where the divergent traces are inferred from those finite traces of P having unbounded A continuations.

The special process $FChaos_H$ has the same finite behaviour as $Chaos_H$, but it does not have any infinite traces:

$$\begin{array}{rcl}
\textsc{Fails}(FChaos_H) & = & \textsc{Fails}(Chaos_H) \\
\textsc{Divs}(FChaos_H) & = & \emptyset \\
\textsc{Infs}(FChaos_H) & = & \emptyset \ .
\end{array}$$