Safety through Security

Andrew Clive Simpson

St. Hugh's College

Submitted for the degree of Doctor of Philosophy, Trinity Term 1996

Safety through Security

Andrew Clive Simpson

St. Hugh's College

Submitted for the degree of Doctor of Philosophy, Trinity Term 1996

Abstract

In this thesis, we investigate the applicability of the process algebraic formal method Communicating Sequential Processes (CSP) [Hoa85] to the development and analysis of safety-critical systems. We also investigate how these tasks might be aided by mechanical verification, which is provided in the form of the proof tool Failures-Divergences Refinement (FDR) [Ros94].

Initially, we build upon the work of [RWW94, Ros95], in which CSP treatments of the security property of non-interference are described. We use one such formulation to define a property called protection, which unifies our views of safety and security. As well as applying protection to the analysis of safety-critical systems, we develop a proof system for this property, which in conjunction with the opportunity for automated analysis provided by FDR, enables us to apply the approach to problems of a sizable complexity.

We then describe how FDR can be applied to the analysis of mutual exclusion, which is a specific form of non-interference. We investigate a number of well-known solutions to the problem, and illustrate how such mutual exclusion algorithms can be interpreted as CSP processes and verified with FDR. Furthermore, we develop a means of verifying the fault-tolerance of such algorithms in terms of protection.

In turn, mutual exclusion is used to describe safety properties of geographic data associated with Solid State Interlocking (SSI) railway signalling systems. We show how FDR can be used to describe these properties and model interlocking databases. The CSP approach to compositionality allows us to decompose such models, thus reducing the complexity of analysing safety invariants of SSI geographic data. As such, we describe how the mechanical verification of Solid State Interlocking geographic data, which was previously considered to be an intractable problem for the current generation of mechanical verification tools, is computationally feasible using FDR.

Thus, the goals of this thesis are twofold. The first goal is to establish a formal encapsulation of a theory of safety-critical systems based upon the relationship which exists between safety and security. The second goal is to establish that CSP, together with FDR, can be applied to the modelling of Solid State Interlocking geographic databases. Furthermore, we shall attempt to demonstrate that such modelling can scale up to large-scale systems.

Acknowledgements

First and foremost, I would like to thank my supervisor, Jim Woodcock, for his constant encouragement and guidance over the past three years.

Thanks are also due to each of the following for providing support (in whatever shape or form) over that period: Clive Adams, Doug Befroy, Nigel Burke, Graham Cairns, Glyn Carter, Tracy Catlin, Andy Chu, Nigel Cooper, Jim Davies, John Eklund, Ian Frigaard, Birgit Gruber, Don Hayward, Simon Hemsworth, Jon Hill, Ed Holland, Gavin Holt, Mike Ingleby, Richard Jones, Christine Joynes, Matt Kernot, Robin Knight, Hans Koeppel, James Lawrie, Robert Leese, David Lenton, Simba Matondo, Steve McKeever, David Mee, Torsten Meiβner, Aidan Moody, Chris Patmore, Richard Potts, John Redmond, Catherine Robinson, Bill Roscoe, Mark Spivack, Tony Terry, Steve Turner, Florian von Brunn, and Eddie Wilson.

Thanks also to the EPSRC, Smith System Engineering, and St. Hugh's College for financial support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In recent years, software-based systems have been introduced into many new application areas. There are a number of reasons for this, including the fact that such systems are small, cheap, and fast in operation. In addition, it is claimed that they offer easier maintenance, reduced installation costs, and increased potential for reuse. The widespread uptake of software-based systems has led to many such systems being introduced into areas where failure could lead to life-threatening situations. Applications in which failure (of either hardware or software) is intolerable are referred to as being *safety-critical.*

An abundance of stories of such failures[1] has led to questions being raised by a number of groups, including the press, software professionals and safety professionals, regarding the use of software technology in safety-critical systems. For example, the following questions might be asked [Red93].

- Is the technology sufficiently mature for safety-critical applications?
- Is it adequately understood, and are we sure that we can control it?
- Are its engineers appropriately trained, or even, do we yet understand how to train them?
- Are there standards in place for development, for quality, for acceptable risk?

Unfortunately, there are cases in which there is no alternative other than for software to play a major part in the operation of a system. For example, systems in application areas such as missile tracking require calculations to be made at such a speed that they can only be achieved through the use of software. In these cases, the best practice is to attempt to design the system in a way that software cannot lead to catastrophic failures.

There is no doubt that there is a need for both the development and application of methods which are capable of providing assurance for such systems. A number of approaches have been advocated for improving the reliability of safety-critical systems. These include the application of neural networks [Dod93], knowledge-based systems [Fox93], and formal methods, e.g., [Sen89], [McD91], and [BS93b].

In this thesis, we investigate the applicability of formal methods (specifically, the process algebraic notation Communicating Sequential Processes (CSP) [Hoa85] and the associated

[1] See, for example, [Neu95] for a detailed collection of such incidents.

proof tool Failures-Divergences Refinement (FDR) [Ros94][2]) to the development and analysis of safety-critical systems. We start by providing an overview of the three topics which are central to this thesis: safety-critical systems, formal methods, and CSP.

1.1 Safety-critical Systems

What is a Safety-Critical System?

There are many different (and often conflicting) definitions of the term safety-critical in the literature, and a standard meaning has yet to emerge. A typical definition is that of [dLSA92].

> "A safety-critical system is a system for which there is at least one failure that can be adjudged to cause a disaster (e.g., loss of life)."

Despite the fact that most definitions seem to agree that safety-criticality involves a risk to human life, thereafter definitions differ in the following ways [Jac94].

- Some distinguish between single deaths and multiple catastrophic deaths.
- Some include injury.
- Some refer to long-term incapacitating illness.
- Some are concerned with adverse effects on the environment.
- Some include damage to property.
- Some include self-inflicted damage to the system itself.

Many problems surround the development of safety-critical systems. One such problem is that such systems often have unique characteristics and will involve novel factors, either because the technology is new, or because established technologies are being used in a novel fashion [Cla93]. It is in precisely these cases that there is difficulty in establishing the nature and cause of *failures* (q.v.).

Failures

A failure may be due to a system's hardware or software, or to human error. All software failures are due to design faults [Lit93], which may take the form of incorrect or incomplete specifications, software coding errors, or software design errors. Such failures are termed *systematic failures* (q.v.). Alternatively, those failures which occur as consequences of hardware degradation are termed *random failures* (q.v.).

"What happens if a failure does occur?" is a key question which must be asked of every system in which a failure could lead to harm. We refer to how a system behaves in the presence of a failure as its *failure mode* (q.v.). For example, a system may *fail-safe* (q.v.), *fail-soft* (q.v.), or *fail-operational* (q.v.).

Following the occurrence of a failure, the fail-safe approach involves ensuring that the system is left in a safe state, often with no further activity taking place. An example of such an approach is if, after discovering a failure in a signalling system, all signals under that

[2]FDR is a refinement checker, developed by Formal Systems (Europe) Limited, 3 Alfred Street, Oxford OX1 4EH.

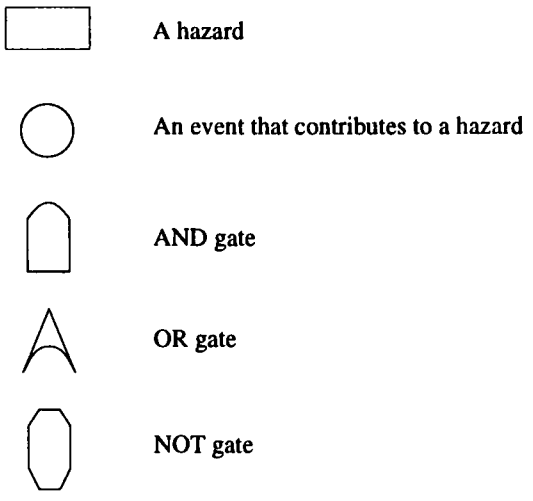

Figure 1.1: Fault tree symbols.

system's control are set to 'stop'. The fail-soft approach involves offering a reduced service which 'works around' the failure. In the case of our faulty signalling system, this might involve reducing the maximum permitted speed for trains on specific sections of track so that part of the responsibility for safety is transferred from the signalling system to the trains and drivers. Finally, a system failing operational would indicate that the system continues to provide a normal service, perhaps via a backup system, with a proviso being that this continuation should not result in a hazardous situation. It should be noted that there is not always an option as to which approach should be adopted.

We take, as a further example, a chemical reactor which is powered by two engines. If, following the failure of one engine, the system shuts down completely leaving the reactor in a safe state, then we say that it fails-safe. On the other hand, if a reduced service can be guaranteed with just one engine, again leaving the reactor in a safe state, then we say that it fails-soft. Finally, if the system can offer a normal service, either with a single engine or by means of a backup engine, then we say that it fails-operational.

The Human Factor

It is generally accepted that safety is not just a software or hardware matter: it involves the system as a whole, including human operators. One difficulty in reasoning about humans in a safety-critical context is the sheer complexity of predicting their behaviour. After all, humans are unpredictable and have an astounding ability to act in unusual and unsafe ways.[3] Also, designers have no authority over human behaviour. However, there are some systems where there is no option but to integrate humans into the system design.[4] The interactions are complex, but by considering the human as part of the system, understanding human behaviour, and designing for human capabilities, many problems associated with human behaviour can be overcome.

As complex systems require a careful analysis of *hazards* (q.v.), we must consider humans to be a potential source.

[3] See [Rea79] for a discussion of human behaviour and [Hit92] for a discussion of systems engineering issues.
[4] For example, a pilot in an aeroplane.

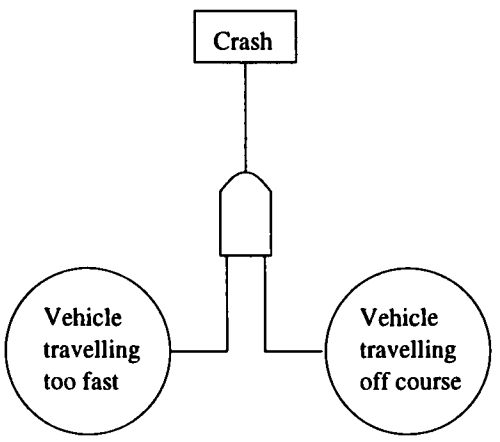

Figure 1.2: A fault tree.

Hazard and Risk Analysis

Judgement upon the implementation of a safety-critical system is often based upon the results of a *hazard analysis* (q.v.) and a *risk analysis* (q.v.).

Hazard analysis can take many forms, examples of which include Fault Tree Analysis (FTA), Event Tree Analysis (ETA), Hazards and Operability Analysis (HAZOP), and Failure Modes and Effects Analysis (FMEA).[5]

Fault trees consist of events combined by AND, OR, and NOT gates. A selection of fault tree symbols is given in Figure 1.1, and a simple example of a fault tree is given by Figure 1.2. Here, if a vehicle travels too fast and off course, then it will crash. Event trees identify the possible outcomes following *initiating events* (q.v.). Simple examples of event trees are given by Figure 1.3 and Figure 1.4.

The process of analysing hazards involves identifying potential hazards and investigating their *risk* (q.v.). Risk can be derived either from historical information about the components of the system and the interfaces between them, or from historical data concerning related systems [Lev91]. Three forms of risk analysis are listed below [Lev86].

- *Single-valued best estimate.* Used when the problem is well enough understood to build models, and there is sufficient data to use best-estimate values for its parameters.
- *Probabilistic.* Used when the problem is well enough understood to build models, but there is limited available data, and so probabilistic values are used as parameters.
- *Bounded.* Used when little is known about a problem, but limited information can be used to bound the answer.

In general, judgement of risk should be carried out by an expert in the field, as determining which aspects of system behaviour can result in danger to humans often requires expert knowledge [McD93].

Following [Lev95], we can think of hazard analysis as being a subset of risk analysis.

> "Hazard analysis ... involves only the identification of hazards and the assessment of hazard level, while risk analysis adds the identification and assessment of the

[5] See [Lev95] for descriptions and evaluations of each of these techniques.

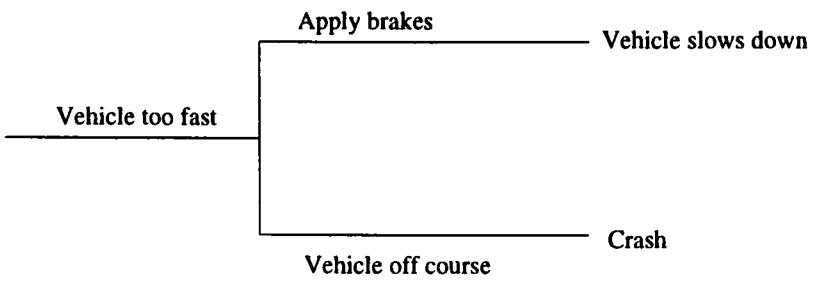

Figure 1.3: An event tree with initiating event 'Vehicle too fast'.

environmental conditions along with exposure or duration."

Proving Safety

Following hazard and risk analyses, one may attempt to illustrate the (relative) safety of the proposed system. This might involve either showing that failures cannot occur, or if they can, that they are not dangerous. Unfortunately, *absolute safety* (q.v.) can never be proved for most (if not all) systems [McD93]. This is because, in general, safety-critical systems are so complex that we can never be sure that all possible circumstances have been taken into account. In practice, the best we can do is define an acceptable level of safety and reduce the risks to this level [Red93]. Unacceptable risks may be eliminated in a number of ways, such as redesign of requirements, incorporation of safety features, or incorporation of warning devices [Rus93].

An oft-quoted method for measuring software reliability is that of probability of failure per hour. Historically, the reliability required for safety-critical applications is extremely high (in the order of between 10^{-7} and 10^{-9} failures per hour). However, it is now appreciated that such an evaluation of safety is not practical, as the length of time required to provide assurance that such levels of reliability hold of a system is usually far too long [McD91]. Furthermore, one might consider the quantification of software reliability to be meaningless — software is either correct or incorrect with respect to its requirements [BF93].

Safety-critical Development

In summary, safety-critical systems are extremely complex, and a great deal of care should be taken during the course of their development. The following describes the safety aspects of the development process [McD93].

1. Identify potential hazards.
2. Design and verify the system to show that the hazards will not (are sufficiently unlikely to) arise.
3. Analyse possible failure modes and show that safety is maintained, even in the presence of failures.
4. Control the development process and produce documentary evidence to prove that steps 1, 2, and 3 have been accomplished sufficiently.

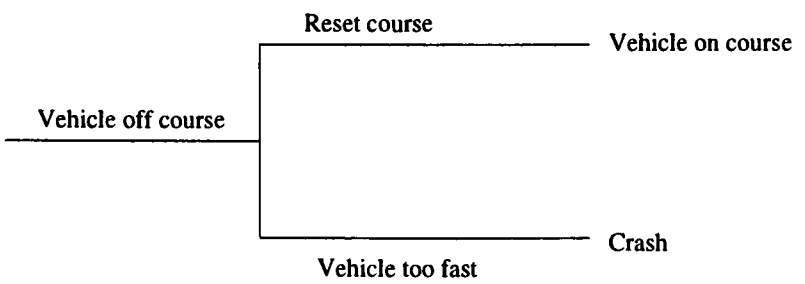

Figure 1.4: An event tree with initiating event 'Vehicle off course'.

1.2 Formal Methods

One approach to improving the confidence in critical systems is the application of formal methods. This can be done either in isolation, or in conjunction with other approaches such as testing and simulation.

What are Formal Methods?

Although often written about, the term 'formal methods' has been given many different definitions. One of the clearest is that of [Rus93].

> " 'Formal methods' are the use of mathematical techniques in the design and analysis of computer hardware and software; in particular, formal methods allow properties of a computer system to be predicted from a mathematical model of the system by a process akin to calculation."

Further, formal methods are described in [Win90] in the following way.

> "[Formal methods are] mathematically based techniques for describing system properties. Such formal methods provide frameworks within which people can specify, develop, and verify systems in a systematic, rather than ad hoc manner."

However, as is pointed out in [BM92], a general definition of formal methods is all very well, but one should avoid making general observations about formal methods due to the large number of different techniques, which, in general, can be divided into the following broad categories.[6]

- Model-based approaches, e.g., Z [BN92, Spi92, WD96] and VDM [Jon90].
- Algebraic approaches, e.g., OBJ [GW88] and ACT ONE [EFH83].
- Process algebras, e.g., CSP [Hoa85] and CCS [Mil80, Mil89].
- Logic-based approaches, e.g., Temporal Logic [Gal87] and HOL [GM93].
- Net-based approaches, e.g., Petri nets [Rei85] and Statecharts [Har84, HPSS87, Har92].

[6] In reality, making this distinction is not always easy, as a number of 'hybrid' methods exist, e.g., LOTOS [BB87, ISO88].

Benefits

Formal methods can provide clear and unambiguous specifications. They can also make the task of verification easier and more effective. Together, these benefits provide a means of proving that a (formally developed) system implementation is correct with respect to its (formally stated) requirements.

Clear and unambiguous specifications are possible because the meaning of a formal specification is mathematical. This also provides formal methods with the rigour of mathematical expression. Furthermore, mathematics brings with it many general concepts which can be applied to a number of different systems that might be special cases of a general problem. This is another benefit of formal methods: they possess the power of mathematical abstraction, which enables us to provide descriptions of large, complex systems. In addition, modularity allows us to concentrate on the behaviour of particular system components without concerning ourselves with the behaviour of others.

A further benefit of the use of certain formal methods is the opportunity for refinement: a relationship between two descriptions of the same system, with one being in some sense 'better than' the other. Thus, an abstract description of a system can be replaced by a concrete one with a guarantee of consistency with respect to these conditions.[7]

It is well documented that a major cost of software production is the introduction of errors during its development, with approximately two-thirds of errors introduced during requirements specification and design [Cha86]. In this respect, formal methods can (depending on the cost of their application) reduce the cost of software development, with formal verification providing a means of reducing the possibility of errors, and formal modelling moving the possibility of error detection to earlier in the development process [Pla93]. By eliminating errors at the requirements stage, total cost and time spent on testing and maintenance can be reduced.

Arguably, formal methods are most effective when used as a concrete mechanism for documenting design decisions — a task which is aided by the mathematical precision afforded to them. Also, when used as a basis of verifying a system's correctness, not only are there the benefits of clarity, conciseness, etc., but there is the opportunity of proving that a system does what it is supposed to. Finally, when accompanied by reliable support tools, the possibilities of proof checking and animation are also afforded to formal methods.

Drawbacks

It must be remembered that formal methods do have their limitations. Despite the fact that formal methods can help in proving that an implementation is correct with respect to its requirements, how do we know that the requirements themselves are correct? For example, how do we know that they are correct or consistent? A proof of correctness proves only that an implementation meets its specification. But are there possible events and situations not anticipated in the specification? In addition, there may be unarticulated requirements that the specification is assumed to possess [Bor87]. In this respect, [Pla93] identifies

- *explicit requirements*: the requirements that are explicitly stated; and
- *implicit requirements*: the unstated qualities that the system is assumed to possess.

[7] See [McD89] for an introduction to the topic of refinement.

Although there are many cases in which the use of formal methods can greatly increase the confidence in computer systems, it should not be supposed that they can guarantee failure-free systems. This is simply because neither humans nor verification techniques are perfect. In addition, a formal proof that a program satisfies a desired property is usually larger than the program itself [Lev95]. Other drawbacks include the facts that the cost of failure is expensive and, in some cases, the cost of training is high [BS93a]. In addition, a number of misconceptions or *myths* still surround the use of formal methods [Hal90]. Ultimately, the best way to dispell such myths is to apply formal methods sensibly, i.e., selectively and not just for the sake of it [BS93a].

The Application of Formal Methods

"How are formal methods best applied?" is a topic of ongoing research. In particular, the following issues need to be addressed [Pla93].

- How should formal methods be used in conjunction with other techniques?
- What are the benefits of applying formal methods throughout the whole of a system's development?
- If they should not be applied throughout the whole development process, at which stages should they be applied?

In addressing the first of these questions, it is our belief that formal methods are best used in conjunction with techniques which are already established in the area of application. Indeed, it is essential that they are capable of integration with current techniques [WZ91], and should be seen as being complementary to, rather than a replacement for, existing technologies [BS93a]. Furthermore, whenever formal methods are applied to any large-scale system, it is desirable that a 'partnership' approach is adopted. This involves the collaboration of experts from the area from which the procured system is to be applied and from the formal methods world. Working in conjunction, the former can identify what needs to be proved in terms of functionality, safety, etc., while the latter can determine appropriate means of proof and development.

Rather than advocating the use of one particular method throughout the whole of a system's development, we believe that it is often beneficial to apply different techniques at different stages. In addition, different methods might be used for reasoning about different types of system property. For example, [CDSW95] describes the use of Z and CSP in developing a formal description of a token ring protocol. The former is used to provide a clear statement of the system's requirements, while the latter builds upon this specification and is used to describe how the stations communicate. A further example is [Web96], which integrates statecharts and Z.

The Industrial Uptake of Formal Methods

Industry has already made gains from the use of formal methods, as when formal methods have been applied in industry, they have usually been applied successfully [Tho93]. Two well-cited examples are the IBM CICS project [Hay85], where a saving of 9% in development costs is claimed [HK91], and the Inmos T800 transputer floating-point unit, where a saving

of 12 months' development time is claimed [May90]. These examples of applications are somewhat typical in that safety-critical systems form only a small percentage of all industrial applications of formal methods [BSC92]. Industry has been "wisely cautious" [BS93b] in applying formal methods in this area [Nee92].

It is generally accepted that the country with the largest uptake of formal methods has been the United Kingdom. But even here, the industrial usage has been relatively small. There have been a number of reasons suggested for this resistance. These include a reluctance from managers because of the cost, and the lack of mathematical expertise of many practitioners within industry. Also, it is claimed in [Hal90][8] that a number of widespread *myths* have surrounded formal methods which have provided ammunition for their detractors. One such myth cited in [Hal90] is that "formal methods are all about program proving." In fact, it can be argued that there are three increasingly rigorous levels of specification [WKC91].

1. Formal specification with no proofs.
2. Formal specification with manual proofs.
3. Formal specification with mechanical proofs.

It is our belief that even when applied at the lowest level, where there has been much industrial use [BSC92, BS93a], formal methods can help in reducing errors.

Other factors have limited the industrial uptake of formal methods — not all of them myths. One of the biggest barriers has been the lack of reliable support tools. In addition there is a user-friendliness, or readability, problem — without greater education in discrete mathematics within industry, and greater understanding of the importance of readable system descriptions within the academic community, it will be extremely difficult for a widespread industrial acceptance of formal methods to materialise. In addition, it is suggested in [Fin96] that those championing the cause of formal methods should not underestimate the level of mathematical expertise required for their application.

Many other reasons have been suggested for the slow uptake of formal methods in industry. For example, [Qui93] suggests that

- there is little assistance available in developing an informally stated requirement into a formally stated one;
- the specification stated in logic is often perceived as being less useful than the informal, natural language version; and
- the process of formalising a specification is not seen as being particularly valuable.

In addition, [BS93a] postulates

- it is claimed that the methods are infeasible for any realistically sized problem;
- there is an understandable reluctance to use formal methods since they have been largely untried in practice;
- many engineers, programmers and managers, although possessing the ability, do not currently have the skills to apply the techniques beneficially; and
- there is scope for improvement, especially in the area of requirements capture.

[8] See also the follow-up article, [BH94].

Taking these and other factors into account,[9] appropriate action has been taken in some quarters and, slowly but surely, the transfer of technology from academia to industry is starting to occur.

One area in which formal methods have been successfully applied is that of railway signalling — a subject to which we shall return in later chapters. For example, [DM94] describes the use of B [Abr96] in the development of three safety-critical systems in the railway industry. In addition, [CGR93] is a collection of reports detailing the use of formal methods in a number of industries (including railway signalling), while [Pla93] provides an in-depth study of the industrial uptake of formal methods. References such as these are invaluable in gauging the current uptake of formal methods in an industrial context, and developing ideas for their future exploitation.

A major step towards the acceptance of formal methods in the U.K. came with the draft Interim Defence Standards 00-55 [MoD91a] and 00-56 [MoD91b].[10] The former is concerned with the procurement of safety-critical software in defence equipment, while the latter is concerned with hazard analysis and safety classification. As is pointed out in [BFM91], formal methods have been used extensively for security-critical systems for several years, and

> "it is interesting to speculate whether the 00-55 will impact [sic] the formal methods market in the way that the DoD Orange Book [DoD85] focussed the U.S. verification industry."

One of the aims of this thesis is to establish a correspondence between the areas of safety and security, by investigating how a concept usually associated with security-critical systems — non-interference — might be used to describe properties of safety-critical systems. Before detailing this approach, we provide a brief introduction to the notation and semantics of the technique which we employ, Communicating Sequential Processes (CSP) [Hoa85], and to the mechanical proof tool, Failures-Divergences Refinement (FDR) [Ros94], which has been used to verify the methods, examples, and case studies presented in this thesis.

1.3 Communicating Sequential Processes

Communicating Sequential Processes (CSP) is a process algebraic formal description technique which is commonly used for describing concurrent properties of distributed systems. In this section, we provide an introduction to both the notation and the semantic models of CSP which we employ. We also define the concepts of refinement and determinism, both of which are central to the approaches to safety advocated in this thesis.

Process Construction

In CSP, atomic actions are referred to as *events*. We let Σ denote the (infinite) set of all events.

Processes are behaviour patterns of objects, as viewed through the availability, occurrence, and refusal of events. We term the set of observable events associated with a process its *alphabet*. Given a process P, we let $\alpha P \subseteq \Sigma$ denote the alphabet of P.

[9] See [WW93] for a 'ten-point plan' which should be satisfied by any 'industrial strength' formal method.
[10] See also [WGH94], which provides a formal specification of 00-56.

Channels are constructs which allow us to pass values via communications. For example, $c!v$ represents the output of value v on the channel c, while $c?x$ represents input on c — it offers the environment the choice of all communications of the form $c.e$, where e is an expression passed along channel c. We let $c.*$ denote all possible communications over channel c. In addition, the function *events* defines the set of all possible communications over a particular channel. For example,

$$events(c) = \{n : \mathbb{N} \bullet c.n\}$$

is the set of all possible communications over a channel c which passes natural numbers.

This notation can be extended to describe a communication such as $a.i!e$, which transmits the expression e on channel i of an array of channels a.

We construct processes according to the following rules.

Stop *Stop* represents the deadlocked process which will never engage in any events.

Skip *Skip* represents a process which does nothing except terminate immediately and successfully.

Event Prefixing The process $e \rightarrow P$ performs the event e, and subsequently behaves as process P. In addition, the process $a : A \rightarrow P$ performs some a from the set of events A, again subsequently behaving as process P.

Internal Choice The process $P \sqcap Q$ represents the internal choice between processes P and Q. The observed behaviour might be that of either process, with the environment having no influence over which is chosen — the choice is made nondeterministically.

External Choice The process $P \Box Q$ represents the external choice between processes P and Q. The environment may choose between P and Q by selecting an appropriate event e. If e is possible initially in P but not in Q, then the behaviour is described by P. If e is possible initially in Q but not in P, then the behaviour is described by Q. Alternatively, if e is possible initially both in P and in Q, then the behaviour is described by $P \sqcap Q$. Indexed external choice is denoted

$$\Box_{a \in A} P(a)$$

This represents the external choice between processes from the set $\{a : A \bullet P(a)\}$. Note that

$$a : A \rightarrow P \equiv \Box_{a \in A} a \rightarrow P$$

Parallel Composition The process $P \,[\![\, A \,]\!]\, Q$ represents the synchronised parallel composition (partial interleaving) of processes P and Q. Here, P and Q must co-operate on the occurrence of events contained in the set A. Events outside A may occur independently in either process. We denote the full parallel composition of P and Q by $P \parallel Q$, such that

$$P \parallel Q \equiv P \,[\![\, \alpha P \cap \alpha Q \,]\!]\, Q$$

In addition, we denote the parallel composition of the processes $P(x)$ for each $x \in X$, with synchronisation on the events of A, by

$$||_{x \in X}[A]P(x)$$

Interleaving The process $P \mid\mid\mid Q$ represents the unsynchronised concurrent operation (interleaving) of processes P and Q. Events of P and Q occur independently, and an event can be refused if and only if both components can independently refuse it. Note that

$$P \mid\mid\mid Q \equiv P \,[\![\, \varnothing \,]\!]\, Q$$

We denote indexed interleaving by

$$|||_{x \in X} P(x)$$

Sequential Composition The process $P \fatsemi Q$ represents the sequential composition of processes P and Q. This process behaves as P until P performs $\checkmark$, and then behaves as Q.

Concealment The process $P \setminus A$ denotes the process P with the events of A concealed.[11] This means that they occur internally and immediately they are ready.

Event Renaming The process $f(P)$ represents the process P with the observable events of P renamed according to the function f.

Interrupts The process $P \triangle (e \rightarrow Q)$ represents the notion of P being interrupted by the event e. This process behaves as P until e occurs. At this point control is transferred to Q. In addition, we write $P \triangle (a : A \rightarrow Q)$ if P can be interrupted by any event a of a set of events A. In the general case, we shall write $P \triangle Q$.

We assume that sequential operators bind tighter than parallel operators. For example,

$$P \sqcap Q \mid\mid\mid R \equiv (P \sqcap Q) \mid\mid\mid R$$

We let *PROC* denote the (infinite) set of all processes over the above operators and Σ.

The Traces Model

A hierarchy of models has been developed for CSP [Ree88], one of which is the *traces model*. A *trace* of a process P is a finite sequence of events in which that process may participate *in that order*. In general, there may be many (possibly infinitely many) traces of a process.

The traces of the process *Stop* are given by

$$traces\,[\![Stop]\!] = \{\langle\rangle\}$$

The only sequence of events which this process may perform is the empty sequence.

[11] As a means of avoiding confusion, we use $-$ to denote the set difference operator.

As further examples, we have the following.

$$\begin{aligned}
&\mathit{traces}\,[\![a \rightarrow \mathit{Stop}]\!] = \{\langle\rangle, \langle a\rangle\} \\
&\mathit{traces}\,[\![a \rightarrow b \rightarrow \mathit{Stop} \,\Box\, c \rightarrow d \rightarrow \mathit{Stop}]\!] = \{\langle\rangle, \langle a\rangle, \langle a, b\rangle, \langle c\rangle, \langle c, d\rangle\} \\
&\mathit{traces}\,[\![a \rightarrow b \rightarrow \mathit{Stop} \,|||\, c \rightarrow \mathit{Stop}]\!] = \{\langle\rangle, \langle a\rangle, \langle a, b\rangle, \langle a, b, c\rangle, \langle a, c\rangle, \\
&\qquad\qquad \langle a, c, b\rangle, \langle c\rangle, \langle c, a\rangle, \langle c, a, b\rangle\}
\end{aligned}$$

The reader is referred to [Hoa85] for a comprehensive list of laws for this model.

The Failures-Divergences Model

The model which we employ in this thesis is that which is often referred to as the 'standard' model for (untimed) CSP: the *failures-divergences model* [BR85]. As it is so regarded, mechanical support for this model is readily available, as we shall see later. Also, this model treats as fundamental the property of determinism, which is central to our approach to safety in CSP.

Given any process $P \in \mathit{PROC}$, the traces, failures, and divergences of P are defined in the following way.

- A trace of P is as defined for the traces model.
- A *failure* of P is a pair (tr, X), such that
  - tr is a trace of P, and
  - X is a set of events which may be refused by P after tr has been observed.
- A *divergence* of P is a trace of P, after which an infinite sequence of internal events may be performed.

We denote the sets of traces, failures, and divergences of a process $P \in \mathit{PROC}$ by $\mathit{traces}\,[\![P]\!]$, $\mathit{failures}\,[\![P]\!]$, and $\mathit{divergences}\,[\![P]\!]$ respectively, such that

- $\mathit{traces}\,[\![P]\!] \subseteq \mathrm{seq}\,\Sigma$,
- $\mathit{failures}\,[\![P]\!] \subseteq \mathrm{seq}\,\Sigma \times \mathbb{P}\,\Sigma$, and
- $\mathit{divergences}\,[\![P]\!] \subseteq \mathrm{seq}\,\Sigma$.

The failures and divergences of any process $P \in \mathit{PROC}$ satisfy the following axioms [BR85].

1. $\forall\, P : \mathit{PROC} \bullet (\langle\rangle, \varnothing) \in \mathit{failures}\,[\![P]\!]$
2. $\forall\, P : \mathit{PROC};\ t_1, t_2 : \mathrm{seq}\,\Sigma \bullet (t_1 \frown t_2, \varnothing) \in \mathit{failures}\,[\![P]\!] \Rightarrow (t_1, \varnothing) \in \mathit{failures}\,[\![P]\!]$
3. $\forall\, P : \mathit{PROC};\ t_1 : \mathrm{seq}\,\Sigma;\ X, Y : \mathbb{P}\,\Sigma \bullet$
   $(t_1, X) \in \mathit{failures}\,[\![P]\!] \wedge Y \subseteq X \Rightarrow (t_1, Y) \in \mathit{failures}\,[\![P]\!]$
4. $\forall\, P : \mathit{PROC};\ t_1 : \mathrm{seq}\,\Sigma;\ X : \mathbb{P}\,\Sigma;\ e : \Sigma \bullet$
   $(t_1, X) \in \mathit{failures}\,[\![P]\!] \wedge (t_1 \frown \langle e\rangle, \varnothing) \notin \mathit{failures}\,[\![P]\!] \Rightarrow (t_1, X \cup \{e\}) \in \mathit{failures}\,[\![P]\!]$
5. $\forall\, P : \mathit{PROC};\ t_1 : \mathrm{seq}\,\Sigma;\ X : \mathbb{P}\,\Sigma \bullet$
   $(\forall\, Y : \mathbb{F}\,\Sigma \mid Y \subseteq X \bullet (t_1, Y) \in \mathit{failures}\,[\![P]\!]) \Rightarrow (t_1, X) \in \mathit{failures}\,[\![P]\!]$

6. $\forall P : PROC;\ t_1, t_2 : \operatorname{seq}\Sigma \bullet$
   $t_1 \in divergences[\![P]\!] \Rightarrow t_1 \frown t_2 \in divergences[\![P]\!]$

7. $\forall P : PROC;\ t_1, t_2 : \operatorname{seq}\Sigma;\ X : \mathbb{P}\Sigma \bullet$
   $t_1 \in divergences[\![P]\!] \Rightarrow (t_1 \frown t_2, X) \in failures[\![P]\!]$

Of course, axiom 7 implies

$$\forall P : PROC;\ t_1 : \operatorname{seq}\Sigma;\ X : \mathbb{P}\Sigma \bullet$$
$$t_1 \in divergences[\![P]\!] \Rightarrow (t_1, X) \in failures[\![P]\!]$$

Together, axioms 1 and 2 state that the traces of a process form a non-empty, prefix-closed set. Axiom 3 states that if a process P may refuse the set of events X after trace t_1, then each subset $Y \subseteq X$ is similarly refused. Axiom 4 states that if any event $e \in \alpha P$ cannot occur following a trace t_1, then it is refused. Axiom 5 states that a set of events is refusable if all of its finite subsets are refusable. Axiom 6 states that if a trace t_1 is a divergence, then so are all traces t_2 such that t_1 **prefix** t_2. Finally, axiom 7 states that all events of a process are refused following a divergence.

We take the following as examples.

$$failures[\![Stop]\!] = \{X : \mathbb{P}\Sigma \bullet (\langle\rangle, X)\}$$

$$\begin{aligned} failures[\![a \rightarrow Stop]\!] = {} & \{X : \mathbb{P}(\Sigma - \{a\}) \bullet (\langle\rangle, X)\} \\ & \cup \\ & \{X : \mathbb{P}\Sigma \bullet (\langle a\rangle, X)\} \end{aligned}$$

$$\begin{aligned} failures[\![a \rightarrow Stop \,\Box\, b \rightarrow Stop]\!] = {} & \{X : \mathbb{P}(\Sigma - \{a, b\}) \bullet (\langle\rangle, X)\} \\ & \cup \\ & \{X : \mathbb{P}\Sigma \bullet (\langle a\rangle, X)\} \\ & \cup \\ & \{X : \mathbb{P}\Sigma \bullet (\langle b\rangle, X)\} \end{aligned}$$

$$\begin{aligned} failures[\![a \rightarrow Stop \sqcap b \rightarrow Stop]\!] = {} & \{X : \mathbb{P}(\Sigma - \{a\}) \bullet (\langle\rangle, X)\} \\ & \cup \\ & \{X : \mathbb{P}(\Sigma - \{b\}) \bullet (\langle\rangle, X)\} \\ & \cup \\ & \{X : \mathbb{P}\Sigma \bullet (\langle a\rangle, X)\} \\ & \cup \\ & \{X : \mathbb{P}\Sigma \bullet (\langle b\rangle, X)\} \end{aligned}$$

The reader is referred to [BR85] and [Hoa85] for a comprehensive list of laws related to this model.

Determinism

The notion of determinism is central to our theory of safety, which is described in Chapter 2. As such, we define what it means for a CSP process to have this property.

Definition 1.1 A process $P \in PROC$ is *deterministic*, denoted $\mathbf{det}(P)$, if and only if

$$divergences\,[\![P]\!] = \varnothing \land \\ \forall\, tr : \mathrm{seq}\,\Sigma;\ e : \Sigma \bullet tr \frown \langle e \rangle \in traces\,[\![P]\!] \Rightarrow (tr, \{e\}) \notin failures\,[\![P]\!]$$

□

That is, a process P is deterministic if and only if it can never diverge, and if it is possible in some state for P to perform the event e, then it cannot also be possible for P to refuse e in that state. We shall write $\mathbf{nondet}(P)$ if P is nondeterministic. In addition, if $divergences\,[\![P]\!] = \varnothing$, then we shall write $\mathbf{nondiv}(P)$ and say that P is nondivergent. Of course, if $\mathbf{det}(P)$, then $\mathbf{nondiv}(P)$.

Take, for example, process P_1.

$$P_1 \mathrel{\hat{=}} a \rightarrow b \rightarrow P_1 \,\Box\, a \rightarrow c \rightarrow P_1$$

Here, it can be observed that there are two different 'paths' which follow a. As such,

$$\langle a, b \rangle \in traces\,[\![P_1]\!] \land (\langle a \rangle, \{b\}) \in failures\,[\![P_1]\!]$$

Therefore, $\mathbf{nondet}(P_1)$.

Alternatively, process P_2 is deterministic.

$$P_2 \mathrel{\hat{=}} a \rightarrow b \rightarrow P_2 \,\Box\, c \rightarrow d \rightarrow P_2$$

Following a (respectively, c), there is no choice as to what communication may occur next — it must be b (respectively, d).

Refinement

A process Q is said to be a *refinement* of another process P (equivalently, P is *refined by* Q), denoted $P \sqsubseteq Q$, if and only if every behaviour of Q is a valid behaviour of P, i.e., Q cannot behave in a manner forbidden by P. In the failures-divergences model, this means that every failure of Q is a failure of P, and every divergence of Q is a divergence of P. This property is defined formally as follows.

Definition 1.2 Process Q is a *failures-divergences refinement* of process P, denoted $P \sqsubseteq_{FD} Q$, if and only if

$$failures\,[\![Q]\!] \subseteq failures\,[\![P]\!] \land divergences\,[\![Q]\!] \subseteq divergences\,[\![P]\!]$$

□

As an example, $P_1 \sqsubseteq_{FD} P_3$.

$$P_3 \mathrel{\hat{=}} a \rightarrow (b \rightarrow P_3 \,\Box\, c \rightarrow P_3)$$

Here,

$$\mathit{failures}[\![P_3]\!] \subseteq \mathit{failures}[\![P_1]\!] \wedge \mathit{divergences}[\![P_3]\!] = \mathit{divergences}[\![P_1]\!]$$

Furthermore, P_3 is deterministic, whereas P_1 is not. Note that any further refinement of P_3 will be deterministic, as any refinement of a deterministic process is also deterministic. Note also that two processes P and Q are equivalent in this model, denoted $P =_{FD} Q$, if and only if $P \sqsubseteq_{FD} Q$ and $Q \sqsubseteq_{FD} P$.

The refinement relation for the traces model is defined as follows.

Definition 1.3 Process Q is a *traces refinement* of process P, denoted $P \sqsubseteq_T Q$, if and only if

$$\mathit{traces}[\![Q]\!] \subseteq \mathit{traces}[\![P]\!]$$

□

Of course, if $P \sqsubseteq_{FD} Q$, then $P \sqsubseteq_T Q$. This is a consequence of the semantic models of CSP being organised in a hierarchy. As a means of determining safety properties, which are often based upon proving that particular (unsafe) traces cannot occur in a given process, this notion of refinement is perfectly satisfactory. However, the traces model is incapable of distinguishing between deterministic and nondeterministic behaviour, which is fundamental to our approach to safety. As such, we shall concentrate mainly on the failures-divergences model — whenever we write $\sqsubseteq$ it should be assumed to mean failures-divergences refinement.

Mechanical Verification

One of the prerequisites for the widespread acceptance of any formal method is the availability of reliable mechanical support. The refinement checker, FDR [Ros94], and its successor, FDR2 [RGG+95], play such a role for CSP.

FDR is a mechanical proof tool which allows one to check properties of finite state CSP processes. Specifically, it allows the testing of refinement between, and determinism of, processes. For example, given two processes P and Q, FDR checks $P \sqsubseteq_{FD} Q$ in the following way [Ros94].

Algorithm 1.1

1. Explore the state space of Q for possible divergence.
2. Explore the product of the state spaces of the two processes. A set of checked pairs and a set of pairs to be checked are maintained. The check fails if it is detected that the process Q has a trace which introduces behaviour which is impossible for P. If this is the case, then the shortest such trace is presented to the user as an aid to debugging. Alternatively, the refinement holds.

□

In addition, given a process $P \in PROC$, the FDR verification that P is deterministic is based upon the following algorithm [Ros95].

Algorithm 1.2

1. Find a deterministic process Q, such that $P \sqsubseteq_{FD} Q$. This is achieved by searching through the state space of P and resolving all nondeterminism that is encountered. When at a stable node (where no internal progress is possible), a single successor for each possible action is chosen. When at a non-stable node, an arbitrary internal action is chosen. This will result in the discovery either of a divergence of P (meaning that the process is nondeterministic), or of a process Q such that $P \sqsubseteq_{FD} Q$.
2. Check whether $Q \sqsubseteq_{FD} P$. If this is so, then it follows that P is deterministic.

□

In the following, the automated analysis of both determinism and refinement is used extensively.

1.4 Structure of the Thesis

The structure of the remainder of this thesis is as follows.

Chapter 2 details our theory of 'safety through security'. We start by providing a review of the work of [RWW94] and [Ros95], in which the security concept of non-interference is described in terms of deterministic CSP processes. We then investigate how this concept can be applied to the analysis of safety-critical systems. After presenting a general model of safety-critical systems, we introduce a notion of safety-critical non-interference, called *protection*, which is used to describe and analyse a number of safety-critical properties.

A proof system for protection is developed in Chapter 3. This provides both a means of reducing proof obligations concerned with protection and a rigorous approach to the development of systems in which this property is desirable.

Chapter 4 identifies a number of safety-critical properties which can be reasoned about in terms of protection. Many of these properties are adapted from our introduction to safety-critical systems given in this chapter, and include approaches to failure modes and fault-tolerance. A number of examples are presented as a means of motivating these properties.

Chapter 5 builds upon the work of Chapters 2-4, and applies our approach to a case study. The system in question is Automatic Train Protection (ATP). Such a system protects trains against the consequences of driver error. We characterise a number of safety conditions which should hold of an ATP specification in terms of protection. Finally, we present a specification which satisfies these properties.

Chapter 6 concentrates on an important topic in concurrent safety-critical system design: mutual exclusion. We argue that mutual exclusion is a form of non-interference, and investigate how this property can be verified via FDR by analysing a number of well known mutual exclusion algorithms. Mutual exclusion also acts as a case study for our approach to fault-tolerance.

Chapter 7 applies FDR to the verification of geographic data associated with Solid State Interlocking (SSI) railway signalling systems. Fundamental safety invariants which must hold

of such data can be stated in terms of mutual exclusion. A number of safety invariants are presented, and we illustrate how such properties can be verified of interlockings with FDR by analysing a small example junction. We also show that these methods scale up to industrial size by applying them to a simplified version of geographic data associated with a real interlocking.

Finally, Chapter 8 summarises, and provides some conclusions arising from, the work described in this thesis. It also identifies a number of areas for possible future work.

Chapter 2

Safety through Determinism

In general, security models can be divided into three categories [Gas88]: access models, information flow models, and non-interference models. *Access* models describe security states in terms of the security attributes of a system's users and data. *Information flow* models attempt to control the transfer of information from one object to another, with such transfer being dependent upon the security attributes of the two objects. Finally, *non-interference* models prevent users operating in different domains from affecting one another in a way that might violate the security policy of the system.

There are three broad categories of problems associated with security-critical systems [RWW94]:

- Confidentiality: the problem of protecting information from unauthorised disclosure.
- Integrity: the problem of protecting information from unauthorised modification or destruction.
- Availability: the problem of avoiding major reduction in system performance.

Each of the above properties can be expressed in terms of information flow (e.g., [Den76] and [GM82]). Non-interference is representative of the notion that if user A's interactions can never affect user B's view, then there is no information flow from the former to the latter. If this is the case, then we say that user A is *non-interfering* with user B.

It is shown in [RWW94], [Wul94], and [Ros95] that non-interference can be described in terms of determinism in CSP. We provide a review of that work in this chapter, and develop a similar approach for the analysis of non-interference properties of safety-critical systems.

Other CSP treatments of non-interference (e.g., those of [All91], [Rya91], and [GC92]) are all equivalent to, or implied by, that work.[1] Other work of interest includes [GM82], [Fol87], [McC87], [McC88], [JT89], [Jac90], [McL90], and [O'H90]. The reader is referred to [GC92] for an excellent survey of work in this area.

By comparison, work in the area of non-interference and safety-critical system, with the exceptions of [Rus89] and [TRC93], is somewhat thin on the ground. Thus, we claim that the work described in this and subsequent chapters not only helps to bridge the gap between

[1] However, as is noted in [Ros95], other CSP approaches have tended to concentrate on the traces model, rather than analysing determinism in the failures-divergences model.

treatments of safety and security, but also provides a novel treatment for the analysis of non-interference properties of safety-critical systems.

In the following, we describe the approach of [RWW94] and [Ros95], which includes amongst its advantages the following:

- It is closed under refinement.
- It is model-independent.
- It can be automatically verified.

The reader is referred to [Wul94] for explanations and justifications of these claims.

Work described in this chapter has previously appeared as [Sim96] and [SW96].

2.1 Security through Determinism

In [RWW94] and [Ros95], determinism in CSP is used to reason about security-critical non-interference properties. The central notion is that if there is the possibility of information flow from one user's interface with a system to another user's interface with the same system, then the former is interfering with the latter in that system.

Assume a system P which interacts with users u_L and u_H in the respective alphabets L and H, such that L and H partition αP. Assume further that u_L has a lower security clearance than u_H. We wish to show that the former cannot gain any information about the latter's interactions with P.

Three properties, all expressible in terms of determinism, can be used to reason about such non-interference. The first, which is termed *eager independence*, conceals the events of H, while a second, *lazy independence*, interleaves them.

Definition 2.1 A process P is said to be *eagerly independent* with respect to a set of events H, denoted $E\text{-}Ind_H(P)$, if and only if $\mathbf{det}(P \setminus H)$. □

Definition 2.2 A process P is said to be *lazily independent* with respect to a set of events H, denoted $L\text{-}Ind_H(P)$, if and only if $\mathbf{det}(P \,|||\, RUN_H)$. □

In the above,

$$RUN_A \mathrel{\hat{=}} \Box_{a \in A}\, a \rightarrow RUN_A$$

is always willing to participate in, and will never refuse, events from the set A.

The terminology employed reflects the semantics of the operators used to define these properties. The abstraction of the events of H by concealment is described as being eager because no other events are guaranteed to occur before any $h \in H$. On the other hand, the abstraction of the events of H by interleaving prevents user u_L from knowing whether an observed H event came from P or from RUN_H. Such a process is deterministic if and only if, following an occurrence of some $h \in H$ in P, the future behaviour of P with respect to the events of L is identical to what its future behaviour with respect to these events would have been had h not occurred.

Therefore, lazy independence is sensitive both to the effects of u_H's actions and to the difference between activity and inactivity [RWW94]: u_H should neither alter u_L's view of

the system, nor delay u_L's progress. Note that in a timed context, eager independence may disturb the timing of events. When H events are abstracted by concealment, they occur internally and immediately they are available — this may result in events from L being 'brought forward'. Lazy independence does not suffer from this drawback.

A further property, termed *strong independence*, can also be defined.

Definition 2.3 A process P is said to be *strongly independent* with respect to a set of events H, denoted $S\text{-}Ind_H(P)$, if and only if $\mathbf{det}((P \,[\![\, H \,]\!]\, CHAOS_H) \setminus H)$. □

In the above,

$$CHAOS_A \mathrel{\widehat{=}} (\sqcap_{a \in A}\, a \rightarrow CHAOS_A) \sqcap Stop$$

is the most nondeterministic nondivergent process offering events from the set A.

We refer to P as being *strongly independent* with respect to H in the sense that a process satisfies this property if and only if it satisfies both eager and lazy independence with respect to H.

Proposition 2.1 $\forall P : PROC;\ H : \mathbb{P}\,\Sigma \bullet S\text{-}Ind_H(P) \Leftrightarrow (E\text{-}Ind_H(P) \wedge L\text{-}Ind_H(P))$

Proof. See [Ros95]. □

We present some examples to motivate these properties. Assume a system with two users, Alan and Bill, who have low and high level security clearance respectively. Bill can both read and write to a file, while Alan can only read it. Furthermore, Bill should be non-interfering with Alan. The alphabets of the two interfaces are defined thus:

$$\alpha Alan = A = \{read_A\}$$
$$\alpha Bill = B = \{read_B, write_B\}$$

The process P_4 is lazily independent with respect to B.

$$\alpha P_4 = \{read_A, read_B, write_B\}$$
$$P_4 \mathrel{\widehat{=}} read_A \rightarrow P_4 \,\Box\, read_B \rightarrow P_4$$

Following an occurrence of $read_B$, Alan's view of the system remains unaffected — he can always perform a $read_A$. Unfortunately, $P_4 \setminus B$ diverges because of the possibility of an infinite sequence of reads by Bill. As such, this process fails to be eagerly independent with respect to B.

Our next example, P_5, is eagerly independent with respect to B.

$$\alpha P_5 = \{read_A, read_B, write_B\}$$
$$P_5 \mathrel{\widehat{=}} read_B \rightarrow write_B \rightarrow read_A \rightarrow P_5 \,\Box\, read_A \rightarrow P_5$$

After the concealment of Bill's reading and writing, Alan's view of the system is unaffected — $read_A$ is always available. However, the process $P_5 \,|||\, RUN_B$ fails to be deterministic because, following a $read_B$ in P_5, Bill must perform a $write_B$ before Alan can perform a $read_A$. As such,

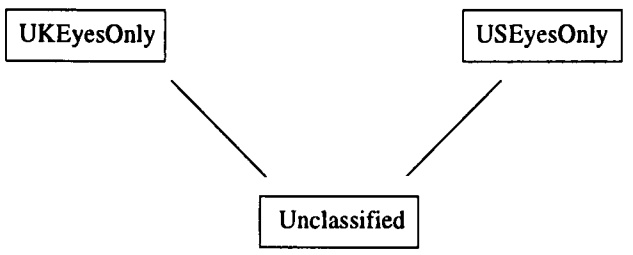

Figure 2.1: An example partial order.

Alan's progress is delayed. This process is eagerly independent, but not lazily independent, with respect to B.

Finally, P_6 is lazily, eagerly, and therefore, by Proposition 2.1, strongly independent with respect to B.

$$\alpha P_6 = \{read_A, read_B, write_B\}$$
$$P_6 \mathrel{\widehat{=}} read_B \rightarrow read_A \rightarrow P_6 \,\Box\, read_A \rightarrow P_6$$

Here, as far as Alan is concerned, $read_A$ is always available. Also, there can never be an infinite uninterrupted sequence of events from B.

If we apply our parallel operators to the above processes, we find that $P_4 \parallel P_6$ and $P_5 \mid\mid\mid P_6$ are both strongly independent with respect to B; neither property holds of $P_4 \parallel P_5$ or $P_5 \parallel P_6$ with respect to B; and $P_4 \mid\mid\mid P_5$ and $P_4 \mid\mid\mid P_6$ are both lazily (but not eagerly) independent with respect to B.

We take $P_4 \parallel P_6$ as an example.

$$\begin{aligned} P_4 \parallel P_6 &= (read_A \rightarrow P_4 \,\Box\, read_B \rightarrow P_4) \parallel (read_B \rightarrow read_A \rightarrow P_6 \,\Box\, read_A \rightarrow P_6) \\ &= read_B \rightarrow (P_4 \parallel read_A \rightarrow P_6) \\ &\quad \Box \\ &\quad read_A \rightarrow (P_4 \parallel P_6) \end{aligned}$$

Expanding again, this is equivalent to

$$\begin{aligned} P_4 \parallel P_6 = {} & read_B \rightarrow ((read_A \rightarrow P_4 \,\Box\, read_B \rightarrow P_4) \parallel (read_A \rightarrow P_6)) \\ & \Box \\ & read_A \rightarrow (P_4 \parallel P_6) \end{aligned}$$

Given the alphabets of P_4 and P_6, $read_A$ must follow an initial $read_B$. As such, the above is equivalent to

$$P_4 \parallel P_6 = read_B \rightarrow read_A \rightarrow (P_4 \parallel P_6) \,\Box\, read_A \rightarrow (P_4 \parallel P_6)$$

In turn, this is equivalent to P_6. As $S\text{-}Ind_B(P_6)$, we can conclude that this condition also holds of $P_4 \parallel P_6$.

The conditions described above can be generalised in the following way [RWW94].

Assume a set *User*, the members of which interact with a system P, together with a partially ordered set *Class* of security clearances. If some $c_2 \in Class$ is considered to be a

higher security clearance than c_1, we shall write $c_1 \preceq c_2$. For example, *Class* might be defined by

$$Class = \{UKEyesOnly, USEyesOnly, Unclassified\}$$

such that

$$Unclassified \preceq UKEyesOnly \land Unclassified \preceq USEyesOnly$$

but neither $UKEyesonly \preceq USEyesOnly$ nor $USEyesOnly \preceq UKEyesOnly$. This partial order is illustrated by Figure 2.1.

Returning to the generic case, the total function

$$cl : User \rightarrow Class$$

assigns a security clearance to every user.

It is assumed that

$$\forall\, u_i, u_j : User \bullet cl(u_i) \neq cl(u_j) \Rightarrow \alpha u_i \cap \alpha u_j = \varnothing$$

The total function

$$\begin{array}{|l} above : Class \rightarrow \mathbb{P}\, \alpha System \\ \hline \forall\, c : Class \bullet above(c) = \alpha System - above^{-1}(c) \end{array}$$

where

$$above^{-1}(c) = \bigcup \{u_i : User \mid cl(u_i) \preceq c \bullet \alpha u_i\}$$

defines, for each $c \in Class$, the set of events which should be non-interfering with the interface of any user of clearance c.

The overall view of such a system is defined thus:

$$Overall \mathrel{\widehat{=}} (|||_{u \in User}\, u) \parallel System$$

Here, users do not interact directly — the question that must be answered is what indirect interference is brought about by the users' interactions with the system? To this end, the properties *E-Ind*, *L-Ind* and *S-Ind* are defined for the general case in the following way.

Definition 2.4 A process P is

- eagerly independent with respect to *Class* if and only if

$$\forall\, c : Class \bullet E\text{-}Ind_{above(c)}(P)$$

- lazily independent with respect to *Class* if and only if

$$\forall\, c : Class \bullet L\text{-}Ind_{above(c)}(P)$$

- strongly independent with respect to *Class* if and only if

$$\forall\, c : Class \bullet S\text{-}Ind_{above(c)}(P)$$

□

As an example, suppose that Alan and Bill have now been joined by Charlie, and that

$$cl(Alan) \preceq cl(Bill) \preceq cl(Charlie)$$

Assume further that everyone has his own file which he can read or write to. Furthermore, everyone can read files belonging to anyone of a lower security clearance. We define the following processes accordingly.

After Charlie has opened file *C*, he can either read or write to it. If he opens file *A* or *B*, then he can only read that file.

$$\begin{aligned} Charlie \mathrel{\hat{=}}\ & OpenA.Charlie \rightarrow ReadA.Charlie \rightarrow CloseA.Charlie \rightarrow Charlie \\ & \Box \\ & OpenB.Charlie \rightarrow ReadB.Charlie \rightarrow CloseB.Charlie \rightarrow Charlie \\ & \Box \\ & OpenC.Charlie \rightarrow ReadC.Charlie \rightarrow CloseC.Charlie \rightarrow Charlie \\ & \qquad\qquad\qquad\qquad \Box \\ & \qquad\qquad\qquad\qquad WriteC.Charlie \rightarrow CloseC.Charlie \rightarrow Charlie \end{aligned}$$

Note that we assume that users can operate on at most one file at any time.

The processes *Alan* and *Bill* are defined similarly.

$$\begin{aligned} Alan \mathrel{\hat{=}}\ & OpenA.Alan \rightarrow ReadA.Alan \rightarrow CloseA.Alan \rightarrow Alan \\ & \qquad\qquad\qquad\qquad \Box \\ & \qquad\qquad\qquad\qquad WriteA.Alan \rightarrow CloseA.Alan \rightarrow Alan \end{aligned}$$

$$\begin{aligned} Bill \mathrel{\hat{=}}\ & OpenA.Bill \rightarrow ReadA.Bill \rightarrow CloseA.Bill \rightarrow Bill \\ & \Box \\ & OpenB.Bill \rightarrow ReadB.Bill \rightarrow CloseB.Bill \rightarrow Bill \\ & \qquad\qquad\qquad\qquad \Box \\ & \qquad\qquad\qquad\qquad WriteB.Bill \rightarrow CloseB.Bill \rightarrow Bill \end{aligned}$$

Files can be accessed by at most one user at a time — we define the process *FileA* accordingly.

$$\begin{aligned} FileA \mathrel{\hat{=}}\ & OpenA?x \rightarrow ReadA.x \rightarrow CloseA.x \rightarrow FileA \\ & \qquad\qquad\qquad\qquad \Box \\ & \qquad\qquad\qquad\qquad WriteA.x \rightarrow CloseA.x \rightarrow FileA \end{aligned}$$

The processes *FileB* and *FileC* are defined similarly. Note that it is the user interfaces which enforce the security protocol, not the files — *FileA* will allow reads and writes by any user.

Given the above processes, our view of the system is defined by

$$System \mathrel{\widehat{=}} (Alan \;|||\; Bill \;|||\; Charlie) \;[\![\; Read\_And\_Write \;]\!]\; (FileA \;|||\; FileB \;|||\; FileC)$$

where *Read_And_Write* is defined by

$$Read\_And\_Write = \bigcup\{\, x : \{Alan, Bill, Charlie\} \bullet \{\, ReadA.x, ReadB.x, ReadC.x, WriteA.x, WriteB.x, WriteC.x \,\}\,\}$$

By Definition 2.4, if $L\text{-}Ind_C(System)$ and $L\text{-}Ind_{B\cup C}(System)$, then *System* is lazily independent. Neither of these conditions hold because of the possibility of Alan's progress being delayed by either Bill or Charlie. Also by Definition 2.4, *System* is eagerly independent if and only if $E\text{-}Ind_C(System)$ and $E\text{-}Ind_{B\cup C}(System)$. Neither of these conditions hold because of the possibility of an infinite sequence of events from *C*.

A different interpretation of this system's behaviour might assume that the actions of reading and writing to files are atomic. As such, we rewrite the processes *Alan*, *Bill*, and *Charlie* in the following way.

$$Alan \mathrel{\widehat{=}} AlanReadA \rightarrow Alan \;\Box\; AlanWriteA \rightarrow Alan$$

$$Bill \mathrel{\widehat{=}} BillReadA \rightarrow Bill \;\Box\; BillReadB \rightarrow Bill \;\Box\; BillWriteB \rightarrow Bill$$

$$\begin{aligned} Charlie \mathrel{\widehat{=}}\; & CharlieReadA \rightarrow Charlie \\ & \Box \\ & CharlieReadB \rightarrow Charlie \\ & \Box \\ & CharlieReadC \rightarrow Charlie \\ & \Box \\ & CharlieWriteC \rightarrow Charlie \end{aligned}$$

We also rewrite the processes *FileA*, *FileB* and *FileC*. Note that as far as these processes are concerned, any user can perform either operation on any file. As an example, we redefine *FileA* as follows.

$$\begin{aligned} FileA \mathrel{\widehat{=}}\; & AlanReadA \rightarrow FileA \\ & \Box \\ & AlanWriteA \rightarrow FileA \\ & \Box \\ & BillReadA \rightarrow FileA \\ & \Box \\ & BillWriteA \rightarrow FileA \\ & \Box \\ & CharlieReadA \rightarrow FileA \\ & \Box \\ & CharlieWriteA \rightarrow FileA \end{aligned}$$

The processes *FileB* and *FileC* are defined similarly.

Given the above, we find that

$$System = Alan \;|||\; Bill \;|||\; Charlie$$

As such, this new interpretation of *System* is lazily independent. However, it is not eagerly independent because there is still the possibility of an infinite uninterrupted sequence of events from C.

The above example illustrates that the absence or presence of non-interference often depends upon the chosen level of abstraction. This, together with the issue of (often mistaken) assumptions of atomicity is discussed in later chapters.

2.2 Safety through Security

One of the contentions of this thesis is that the notion of non-interference is applicable to the analysis of safety-critical systems, as well as security-critical systems. We detail these ideas in this section, and start by investigating the relationship between safety and security.

2.2.1 The Safety/Security Correspondence

There are a number of common threads which relate safety-critical and security-critical systems.[2] We identify a number of such threads here.

One way in which safety-criticality and security-criticality are related is by the notion of *protection* (albeit in subtly different ways). The former must offer protection *to* the general public, whereas security-critical systems must offer protection *from* them.

Secondly, security and safety are both *non-functional* properties. What we mean by this is that both properties may be fundamental to a system's requirements, but neither can be proved by simply observing the behaviour of particular system components (although they may be disproved in this fashion). Another way of interpreting non-functionality is to say that we cannot write either a (non-trivial) safety or a security property as a process P, and then test if a specification S satisfies this property by checking $P \sqsubseteq S$.

In recent years, the use of formal methods has been encouraged in the development of safety-critical software procured by the U.K. Ministry of Defence [MoD91a] and security-critical software procured by the U.S. Department of Defense [DoD85]. It follows that by relating the two concepts, areas of work previously thought applicable to one type of system might have new application areas made available to them. To quote [Lev95],

> "Safety and security ... are closely related, and their similarities can be used to the advantage of both in terms of borrowing effective techniques from each to deal with the other."

The idea of relating safety and security concepts has been considered before. For example, [Rus89] proposes a kernel for safety, which is based on the notion of security kernels. The idea of a security (respectively, safety) kernel is based upon the theory that, given a large system, a relatively small proportion of that system should be responsible for security (respectively, safety).

[2] See [LS87] for an introduction to security issues and [Lev95] for an introduction to safety issues.

An excellent article which considers the relationship that exists between safety and security is [BMD92]. As well as pointing out that some mistakenly regard safety as a special case of security, the following interesting point is made.

> "At a linguistic level, the common phrase 'safe and secure' indicates a limited distinction and, in German, no real distinction can be made as the term *sicherheit* means both safety and security (although some technical terminology has been introduced to make distinctions between the two notions). Further, in many dictionaries, safety is defined in terms of security — and vice versa."

Thus, from a linguistic (and philosophical) viewpoint, there are definite parallels which can be drawn between safety and security.

One such parallel is that it is important that the mechanisms for both safety and (especially) security are, as far as possible, transparent. What we mean by this is that neither requirement should extensively encroach upon the behaviour of the rest of the system. This gives rise to a key point — both safety and security are requirements and they have to be traded off against other requirements (e.g., cost and performance) as such.

The notion of *threat* (either to safety or to security) is central to descriptions of both types of systems. All possible security threats must be understood before a security-critical system may be considered secure, and all possible hazards must be understood before a safety-critical system may be considered safe. An important aspect of identifying threats (or hazards) is that it is essential that the system boundary is determined. Furthermore, it must be ensured that everything within this boundary is secure (or safe). Thus, internal security measures should be accompanied by effective external security measures, such as restricted computer access and locked doors, and internal safety measures should be accompanied by reliable hardware, etc.

Most importantly from our point of view, the behaviour of safety-critical systems and security-critical systems can be described in terms of sequences of events. This makes CSP a suitable technique for reasoning about properties of both types of system. Furthermore, as we shall see in the following, the CSP concept of determinism may be used to describe non-interference properties of safety-critical systems, as well as security-critical systems.

2.2.2 The Characterisation of Events

Before we can discuss non-interference in a safety-critical context, we establish types of non-interference properties one might wish to prove of such systems. One example is that the safety-critical events of a system should be oblivious to the actions taken to prevent or limit the damage of possible faults, i.e., it should be fault-tolerant. Another is that the actions taken to recover from a failure should not affect the system's normal functional behaviour, i.e., it should fail-operational.

Both of these properties identify generic sets of events which are present in safety-critical systems. In the first example, we identified two distinct sets of events — *faults* and *safety-critical* events. *Recovery* events and non-safety-critical behaviour (which we refer to as *benign* events) are two further types of event identified by the latter example. Continuing in this vein, we identify *failures*, which might arise as the result of an unprotected fault, and *hazards*, the occurrence of which leave a system in an uncontrollable and unpredictable state.

We divide the events in which a safety-critical system may participate into the following categories.

Benign events are those which are considered non-safety-critical. This is in the sense that they may form part of a system's functional requirements, but a failure in their operation will not result in a dangerous situation. An example of such behaviour is the updating of non-essential information on a VDU.

Safety-critical events are those which are considered safety-critical, in the sense that if they were not performed, or if they were performed incorrectly, a hazardous situation might arise. An example is the comparison of a train's current speed with its required speed.

Failures are events (or sequences of events) which may lead to a life-threatening situation. An example is the incorrect calculation of a train's required speed, leading it to approach a signal at 'stop' at too high a speed.

Recovery events provide a means of combatting failures. These correspond to the notion of *curative maintenance* (q.v.) of [Lap89], with an example being the immediate application of a train's emergency brakes following the detection of the above failure.

Hazards are beyond our control, and could result in a threat to safety. Such events are, in some sense, a 'point of no return' from which we cannot recover. Given the above failure, a hazard would be the train passing the signal, which might result in a collision.

2.2.3 Failures

Recall from Section 1.1 that the notion of *failure* is central to the analysis of any safety-critical system. Before we can develop a theory of safety for CSP processes, we must identify what it is that we actually mean by failure. It should, of course, not need mentioning that the safety-critical notion of failure should not be confused with the CSP semantic concept of the same name.

In the following, we shall deal with two types of failures — *observational failures* and *causitive failures.*

Observational Failures

Assume a chemical reactor system which deals with five different liquids: A, B, C, D, and E. In addition, there are two neutralising agents associated with the system: N_1 and N_2. Assume further that the system has been set to a suitable temperature and that the conditions are ready for the production of a new chemical. The functional requirement for this process under the current conditions is stated as follows.

To produce the desired result, either

- add A to an empty tank, followed by C and then B; or
- add C to an empty tank, followed by A and then B.

The safety requirements for the process are stated in the following way.

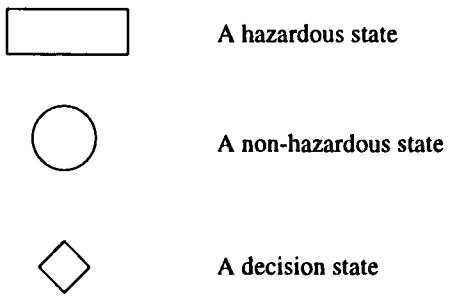

Figure 2.2: State-based hazard analysis symbols.

1. If only A is present in the tank, then B should not be added before C; this could result in an explosion.
2. If only C is present in the tank and B is added, then N_1 should be added within 1 minute to combat a hazardous reaction.
3. If D is added to the tank, then N_2 should be added within 2 minutes to combat a hazardous reaction.
4. E should not be added to the tank; this could result in the emission of poisonous fumes.

In the following, we denote the hazardous reactions resulting from the failure to satisfy safety requirement 2 and safety requirement 3 as *reactionX* and *reactionY* respectively.

We use a simple state-based hazard analysis technique for describing these hazards. Such hazard analyses are labelled according to Figure 2.2. These labels are interpreted as follows.

- A hazardous state represents the fact that a hazard has occurred, and the future behaviour of the system is neither controllable nor predictable.
- A non-hazardous state represents the fact that a hazard has not occurred. Each non-hazardous state has a (possibly empty) set of transitions, corresponding to observations of events relevant to the hazard which are possible in that state.
- A decision state is a Boolean function on a non-hazardous state, from which one of two transitions are possible.

As we shall see later, it is often possible to rewrite a decision state in terms of relevant non-hazardous states and transitions.

Given the above, hazard analyses for our chemical reactor system are given by Figure 2.3, Figure 2.4, Figure 2.5, and Figure 2.6.

For simplification, we shall not consider factors such as the ratio of chemicals necessary for given reactions. We also assume that neutralising agents have no affect if already present in the tank.

For comparison, fault tree and event tree hazard analyses of hazard 3 are given by Figure 2.7 and Figure 2.8 respectively. It can be observed that our state-based hazard analyses are something of a hybrid of fault trees and event trees, and that translation between the notations is a relatively straightforward task.

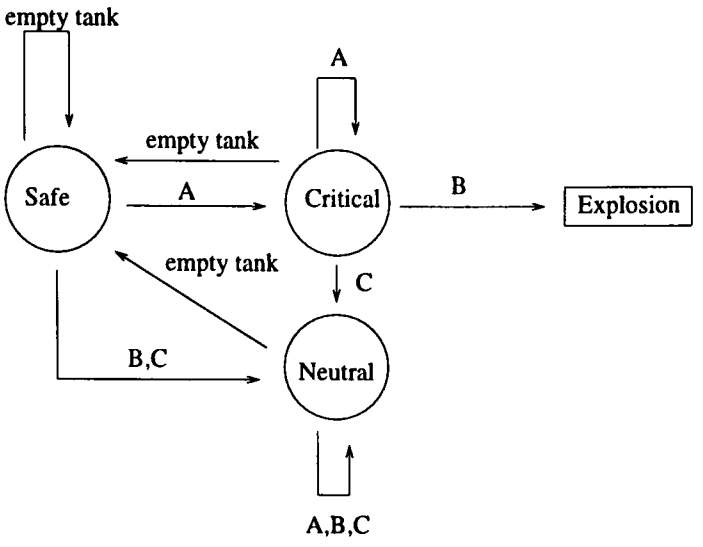

Figure 2.3: Hazard 1.

The safe operation of this reactor is dependent upon *B* being added to the tank only if it contains both *A* and *C*. Also, under no circumstances should *D* or *E* ever be added to the tank. One of this system's failures is *B* being added to the tank when it contains only *C*. The addition of *B* is not, in itself, a failure, but the sequence of *C* being added to an empty tank, followed by *B* being added to the tank containing *C* is. We imagine a 'safety controller' which observes the behaviour of the system and, if this particular sequence of events is observed, blows a metaphorical whistle to indicate the fact a failure has occurred.

In the following, we look to develop a means of analysing the safety (or otherwise) of a safety-critical specification against a hazard analysis in the form of a monitor process.

If we denote the addition of *B* and *C* to the tank by the events *addB* and *addC* respectively, then we might define a monitor process, *SafetyController*, which observes this failure.

$$SafetyController \mathrel{\widehat{=}} addC \rightarrow addB \rightarrow failure \rightarrow Stop$$

Here, if *addB* follows *addC*, then a failure is observed by the safety controller.

We term such failures *observational failures* (q.v.). The event *failure* is not an event in which our chemical reactor system willingly engages, but is an observation made by the safety controller following the occurrence of a particular sequence of events. It is representative of a marker denoting a transition from a safe state to an unsafe state. Because of their nature, we shall refer to observational failures as *ghost events*, and shall label such events by means of underlining. For example, if *e* is a ghost event, then we shall write it in the form $\underline{e}$.

As such, we rewrite the above process in the following way.

$$NewSafetyController \mathrel{\widehat{=}} addC \rightarrow addB \rightarrow \underline{failure} \rightarrow Stop$$

We also denote the system's hazards in this way.

$$Hazards = \{\underline{explosion}, \underline{fumes}, \underline{reactionX}, \underline{reactionY}\}$$

Modelling hazards in this way allows us to reason about their possible occurrence, despite the fact that they do not form part of the system's functional specification.

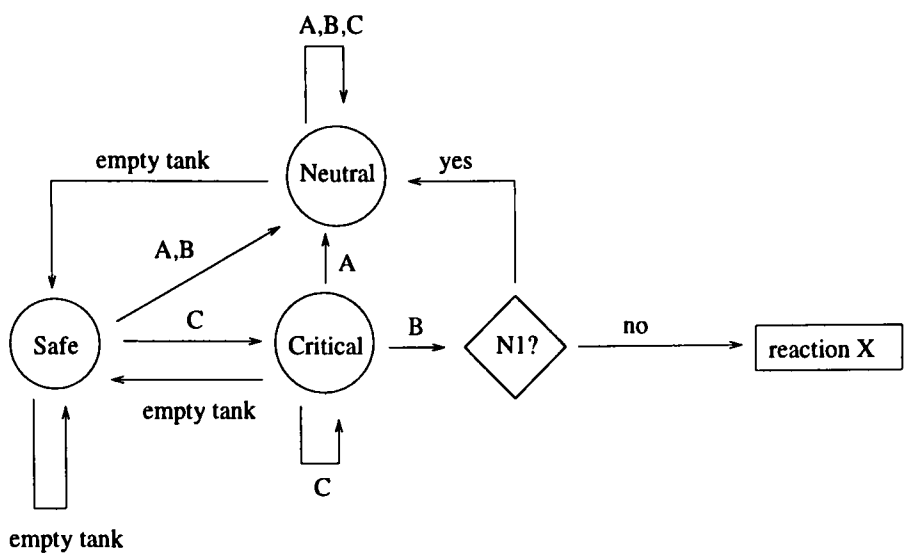

Figure 2.4: Hazard 2.

Unfortunately, the above process does not adequately describe the behaviour of hazard 2 with respect to B and C — the events *addB* and *addC* may well occur in their desired order. Therefore, for completeness, we define the process *CompleteSafetyController*. Here, if B is the first chemical to be added to the tank, then the system enters a safe state. Alternatively, if C is added first, then it enters an unsafe state.

$$\begin{aligned} \mathit{CompleteSafetyController} \mathrel{\widehat{=}} & \mathit{addB} \rightarrow \mathit{SafeState} \\ & \Box \\ & \mathit{addC} \rightarrow \mathit{UnsafeState} \\ & \Box \\ & \mathit{emptyTank} \rightarrow \mathit{CompleteSafetyController} \end{aligned}$$

The system remains in a safe state with respect to this hazard until the tank is emptied, where *emptyTank* is the event representing the action of the tank being emptied. At this point, it returns to the initial state.

$$\begin{aligned} \mathit{SafeState} \mathrel{\widehat{=}} & \mathit{addB} \rightarrow \mathit{SafeState} \\ & \Box \\ & \mathit{addC} \rightarrow \mathit{SafeState} \\ & \Box \\ & \mathit{emptyTank} \rightarrow \mathit{CompleteSafetyController} \end{aligned}$$

Finally, if *addB* is observed when the system is in an unsafe state, then the failure is observed.

$$\begin{aligned} \mathit{UnsafeState} \mathrel{\widehat{=}} & \mathit{addB} \rightarrow \underline{\mathit{failure}} \rightarrow \mathit{Stop} \\ & \Box \\ & \mathit{addC} \rightarrow \mathit{UnsafeState} \\ & \Box \\ & \mathit{emptyTank} \rightarrow \mathit{CompleteSafetyController} \end{aligned}$$

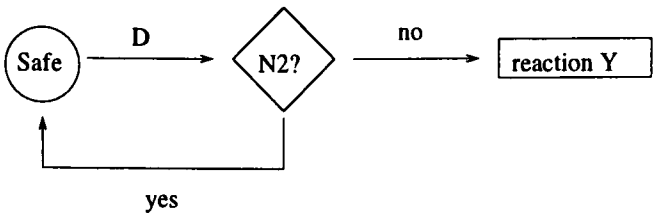

Figure 2.5: Hazard 3.

Note that *failure* is observable by our safety controller if and only if the specific sequence of events leading to this failure occurs.

Causitive Failures

Recall that safety requirement 3 for our chemical reactor system is stated thus:

> If D is added to the tank, then N_2 should be added within 2 minutes to combat a hazardous reaction.

Here, the single event of chemical D being added to the reactor is a failure. D should never be added to the tank, no matter what sequence of events precedes it.

In this case, we define the process $SafetyController_2$.

$$SafetyController_2 \mathrel{\widehat{=}} addD \rightarrow Recovery$$

After observing the addition of D to the tank, the appropriate recovery action, which is defined by *Recovery*, is taken.

At its simplest, the main difference between observational and causitive failures is characterised by the fact that the former are observations of particular sequences of events and can be thought of as *multiple event failures*, in which case the system fails due to an unfortunate combination of events. Alternatively, the latter are *single event failures*, involving events which must be avoided at all costs, in which case the system is seen to fail immediately.

2.2.4 Recovery

Having seen how it is possible to describe failures in terms of CSP, we now investigate how we can describe both the prevention of, and subsequent recovery from, such occurrences. We shall set in place two distinct approaches for dealing with failures. The first method involves combatting their occurrence, while the second involves limiting their consequences. These approaches are termed *preventative* and *recovery* measures respectively.

Our treatment of preventative measures is described further in Section 4.3, in which we discuss fault-tolerance. In this section, we concentrate solely on recovery measures. We characterise recovery in terms of events, or sequences of events, which return a system from an unsafe state to a safe one.

Recall that our chemical reactor system involves two recovery measures: the addition of neutralising agents N_1 and N_2. In the case of a failure resulting from the addition of B to C, the relevant recovery procedure is the addition of N_1 within 1 minute. If D is added to the

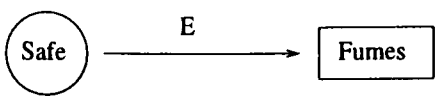

Figure 2.6: Hazard 4.

tank, then the recovery procedure is the addition of N_2 within 2 minutes. If we indicate the passing of a minute by the event *tock*, then a hazard analysis concerning the addition of D to the tank can be described by the following process.

$$
\begin{aligned}
SafetyController_2 \mathrel{\widehat{=}} \; & addD \rightarrow Failure_0 \\
& \Box \\
& tock \rightarrow SafetyController_2 \\
& \Box \\
& addN_2 \rightarrow SafetyController_2
\end{aligned}
$$

$$
\begin{aligned}
Failure_t \mathrel{\widehat{=}} \; & \textbf{if} \quad t \geq 2 \\
& \textbf{then } \underline{reactionY} \rightarrow Stop \\
& \textbf{else } \; addN_2 \rightarrow SafetyController_2 \\
& \qquad \Box \\
& \qquad addD \rightarrow Failure_t \\
& \qquad \Box \\
& \qquad tock \rightarrow Failure_{t+1}
\end{aligned}
$$

We categorise recovery procedures in the same way as we categorise failures. In the above, a *causitive recovery* is brought about by the addition of N_2 to the tank. If the recovery process also involved emptying the tank, then, following an observation of the subtrace $\langle addN_2, emptyTank \rangle$, a ghost event, $\underline{recovery}$, might be observed by a safety controller. This would be an *observational recovery.*

The reader might find it strange that the above process allows an infinite amount of chemical D to be added without any notion of time passing. One should remember, however, that the above does not specify how these events should behave, but rather monitors their progress, updating state information along the way — it will only observe *addD* an infinite number of times without time passing if the system specification allows such behaviour, which, one would hope, it does not.

2.2.5 Hazards

Referring again to our chemical reactor, there are a number of things which this system should never allow. First, B should never be added to the tank which contains only C — under these circumstances, the subtrace $\langle addC, addB \rangle$ is an observational failure. Secondly, D should never be added to the tank — the event *addD* is a causitive failure. In both cases, if recovery does not occur within the allotted time period, then hazardous reactions will occur,

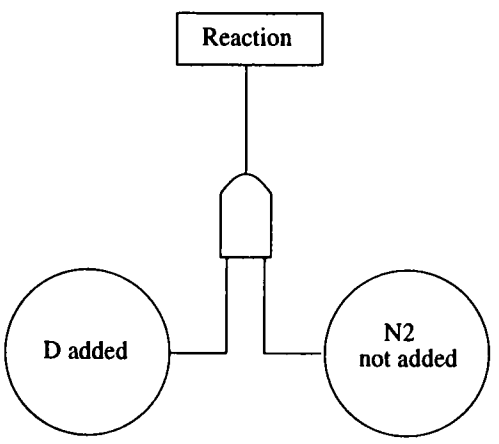

Figure 2.7: A fault tree for hazard 3.

represented by the ghost events *reactionX* and *reactionY* respectively. Furthermore, the future behaviour of the system is no longer predictable — we term such situations *hazards.*

We think of the difference between failures and hazards in the following way. A failure is a deviation from the system's normal behaviour which we may be able to compensate for and recover from, whereas a hazard may occur either as a consequence of a failure or independently, and leaves the future behaviour of the system unpredictable and beyond any control.

As hazardous states are so uncontrollable and unpredictable, we model them by $\perp_A$, which is the most nondeterministic process offering events from the set A. It should be noted that $\perp_A$ can diverge, unlike $CHAOS_A$, which is the most *nondivergent* nondeterministic process. The process $\perp_A$ is, however, equivalent to $CHAOS_A$ of [Hoa85], where it is described as being "the least predictable, the least controllable, and in short the worst [process]."

Denoting hazardous states in this way is representative of the notion that, following a hazard, we have no further control over system behaviour. It is as if the worst has already happened, and we neither have any influence over, nor can we predict, what will happen in the future.

As an example, we define the process $SafetyController_3$.

$$SafetyController_3 \mathrel{\widehat{=}} addE \rightarrow \underline{fumes} \rightarrow \perp_{\{addE, \underline{fumes}\}}$$

This process represents the hazard associated with safety requirement 4. We can think of this process as a hazard analysis, stating what will occur if this system was to allow the event *addE*: following the discharge of poisonous fumes, the behaviour of the system with respect to the alphabet of $SafetyController_3$ is totally unpredictable.

We redefine the process $SafetyController_2$ to take into account this approach to hazards.

$$\begin{aligned} SafetyController_2 \mathrel{\widehat{=}}\ & addD \rightarrow Failure_0 \\ & \Box \\ & tock \rightarrow SafetyController_2 \\ & \Box \\ & addN_2 \rightarrow SafetyController_2 \end{aligned}$$

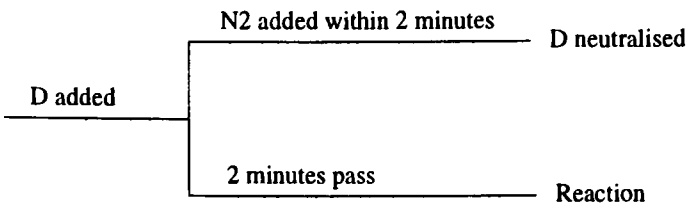

Figure 2.8: An event tree for hazard 3.

As before, if *addD* is observed then the process $Failure_0$ is invoked.

If $addN_2$ is not observed within 2 minutes, then a hazardous state is entered.

$$
\begin{aligned}
Failure_t \mathrel{\hat{=}} \textbf{if}\ & t \geq 2 \\
\textbf{then}\ & \underline{reactionY} \rightarrow \bot_{\{tock, addD, addN_2, \underline{reactionY}\}} \\
\textbf{else}\ & addN_2 \rightarrow SafetyController_2 \\
& \Box \\
& addD \rightarrow Failure_t \\
& \Box \\
& tock \rightarrow Failure_{t+1}
\end{aligned}
$$

The approach outlined above illustrates how CSP might be used as an aid to hazard analysis.

Recall how our approach to hazard analysis corresponds to finite state automata. As every finite state automaton corresponds to a CSP process, it follows that we can rewrite such hazard analyses in terms of CSP processes and test for safety via FDR. Our translation from a state-based hazard analysis to a CSP process can be achieved with reference to the following informal algorithm.

Algorithm 2.1

1. Replace all decision states with standard state/event transitions.
2. Write all non-hazardous states as CSP processes which offer the external choice between all possible transitions from that state.
3. Write all hazardous states as $\bot$.

□

As an example, if we were to replace the decision state of the hazard analysis of Figure 2.5, then we would obtain the equivalent hazard analysis of Figure 2.9. In the next section, we shall see examples of moving from CSP representations of hazard analyses, and how we can test for safety with regard to such processes.

The reader might wonder why we do not test for absence of hazards by means of traces refinement. The simple reason is, as was stated previously, we test for absence of hazardous states — a process enters a hazardous state following a particular sequence (or sequences) of events. Hazards themselves do not form part of a functional specification and, as such, we cannot test for their presence or absence by means of refinement — we are restricted to analysing specifications with regard to the possibility of sequences leading to them.

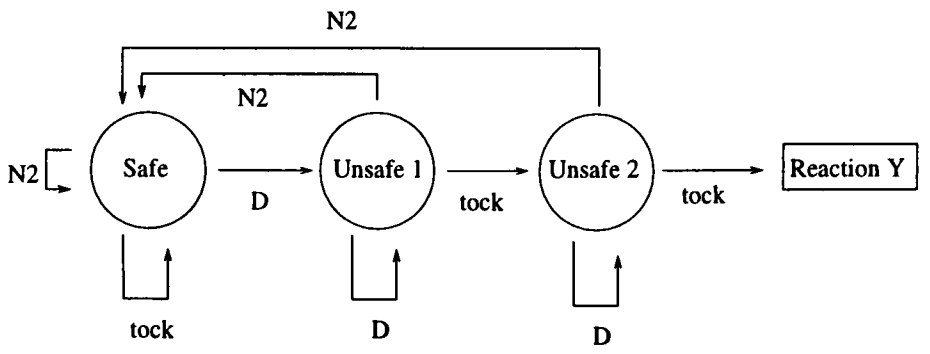

Figure 2.9: Hazard 3 rewritten.

2.2.6 A Safety-Critical Model

For the moment, we assume that, for any given system, the alphabet of that system is partitioned by the sets representing safety-critical events, benign events, failures, recovery measures, and hazards.[3]

Our model of a safety-critical system is given by Figure 2.10. This is labelled in the following way.

- A system is in a *safe* state when it is participating in benign events only, and is not in the presence of any failures or hazards.
- A system is in a *critical* state when it is participating in safety-critical events, and is not in the presence of any failures or hazards.
- A system is in an *unsafe* state when a failure, denoted f, has been observed. Following a failure, either a recovery event, denoted r, returns the system to a critical state, or a hazard, denoted h, occurs, in which case a hazardous state is entered. Note that hazards can occur independently of failures and at any time.
- A system is in a *hazardous* state when a hazard has occurred. No predictions can be made about, nor control exercised over, the system's future behaviour, and it remains in this state *ad infinitum*.

Note that the labels b and e denote the beginning and end of safety-critical activity respectively.

As an example, our chemical reactor system is initially in a safe state. When the system is ready to add chemicals to the tank, it enters a critical state. It remains in this state until one of two possible events happens. If the system is shut down, then it returns to a safe state. Alternatively, if a failure occurs, then an unsafe state is entered. If recovery from the failure is observed, then the system returns to a critical state. Alternatively, if a hazard occurs, then the system enters a hazardous state.

We represent our chemical reactor system's specified functional behaviour and its hazard analysis by the processes *MissionInit* and *Hazards* respectively.

[3]This restriction is removed in Chapter 4.

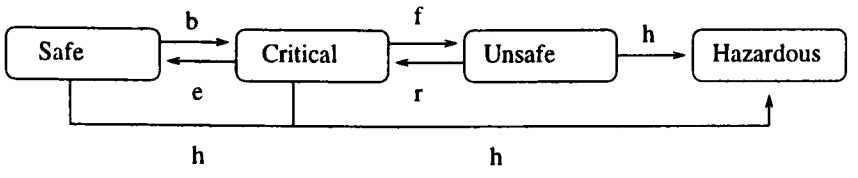

Figure 2.10: A safety-critical model.

The process *MissionInit* is defined in accordance with the system's functional requirement.

$$\alpha MissionInit = \{ start, addA, addB, addC, addD, addE, emptyTank, shutDown \}$$

$$MissionInit \mathrel{\widehat{=}} start \rightarrow (Mission \triangle (shutDown \rightarrow emptyTank \rightarrow MissionInit))$$

$$\begin{aligned} Mission \mathrel{\widehat{=}}\ & addA \rightarrow addC \rightarrow addB \rightarrow emptyTank \rightarrow Mission \\ & \Box \\ & addC \rightarrow addA \rightarrow addB \rightarrow emptyTank \rightarrow Mission \end{aligned}$$

The events *start* and *shutDown* are interpreted in the obvious ways.

The hazard analysis for this system takes into account the four hazards described by the safety requirements. We define a process for each hazard, with the process *Hazards* providing an overall picture with regard to their composition.

$$Hazards \mathrel{\widehat{=}} \|_{i \in \{1,2,3,4\}} Hazard_i^{safe}$$

We think of these processes as state machines which represent our hazard analyses. For example, the process $Hazard_1^{safe}$ is our CSP interpretation of the hazard analysis of Figure 2.3.

$$\alpha Hazard_1^{safe} = \{addA, addB, addC, emptyTank, \underline{explosion}\}$$

$$\begin{aligned} Hazard_1^{safe} \mathrel{\widehat{=}}\ & addA \rightarrow Hazard_1^{A} \\ & \Box \\ & addB \rightarrow Hazard_1^{neutral} \\ & \Box \\ & addC \rightarrow Hazard_1^{neutral} \\ & \Box \\ & emptyTank \rightarrow Hazard_1^{safe} \end{aligned}$$

Initially, if B or C is added to the tank, then this hazard, which is associated with only A being present, is, for the moment not a worry. Therefore, following an observation of *addB* or *addC*, the process $Hazard_1^{neutral}$ is invoked. Alternatively, if *addA* is observed, then this process behaves as $Hazard_1^{A}$. The other observable event in this state is *emptyTank*. If this

occurs, then there is no change in state.

$$
\begin{aligned}
Hazard_1^A \mathrel{\hat{=}} \; & addA \rightarrow Hazard_1^A \\
& \Box \\
& addB \rightarrow \underline{explosion} \rightarrow \perp_{\alpha Hazard_1^{safe}} \\
& \Box \\
& addC \rightarrow Hazard_1^{neutral} \\
& \Box \\
& emptyTank \rightarrow Hazard_1^{safe}
\end{aligned}
$$

Here, if B is added to A, then the hazard *explosion* occurs. Other changes in state are brought about by the events *addC* and *emptyTank*, while an observation of *addA* results in no change of state.

The process $Hazard_1^{neutral}$ is defined as follows.

$$
\begin{aligned}
Hazard_1^{neutral} \mathrel{\hat{=}} \; & addA \rightarrow Hazard_1^{neutral} \\
& \Box \\
& addB \rightarrow Hazard_1^{neutral} \\
& \Box \\
& addC \rightarrow Hazard_1^{neutral} \\
& \Box \\
& emptyTank \rightarrow Hazard_1^{safe}
\end{aligned}
$$

Here, if the event *emptyTank* is observed, then the process returns to its initial state.

We represent our second hazard analysis in a similar fashion.

Initially, if either of A or B is added to an empty tank, then the system enters a neutral state with respect to this hazard. Alternatively, if *addC* is observed, then the process $Hazard_2^C$ is invoked. Observations of $addN_1$, *tock* and *emptyTank* leave the state unchanged.

$$\alpha Hazard_2^{safe} = \{addA, addB, addC, addN_1, tock, emptyTank, \underline{reactionX}\}$$

$$
\begin{aligned}
Hazard_2^{safe} \mathrel{\hat{=}} \; & addA \rightarrow Hazard_2^{neutral} \\
& \Box \\
& addB \rightarrow Hazard_2^{neutral} \\
& \Box \\
& addC \rightarrow Hazard_2^{C} \\
& \Box \\
& addN_1 \rightarrow Hazard_2^{safe} \\
& \Box \\
& tock \rightarrow Hazard_2^{safe} \\
& \Box \\
& emptyTank \rightarrow Hazard_2^{safe}
\end{aligned}
$$

The process $Hazard_2^C$ is defined as follows.

$$
\begin{aligned}
Hazard_2^C \mathrel{\hat{=}} \; & addA \rightarrow Hazard_2^{neutral} \\
& \Box \\
& addB \rightarrow Hazard_2^{unsafe} \\
& \Box \\
& addC \rightarrow Hazard_2^C \\
& \Box \\
& addN_1 \rightarrow Hazard_2^C \\
& \Box \\
& tock \rightarrow Hazard_2^C \\
& \Box \\
& emptyTank \rightarrow Hazard_2^{safe}
\end{aligned}
$$

An unsafe state is entered if the event *addB* is observed. Alternatively, the addition of A will result in a change to a neutral state. No change of state occurs if any of *addC*, $addN_1$ or *tock* are observed. However, an observation of *emptyTank* returns this process to its initial state.

$$
\begin{aligned}
Hazard_2^{neutral} \mathrel{\hat{=}} \; & addA \rightarrow Hazard_2^{neutral} \\
& \Box \\
& addB \rightarrow Hazard_2^{neutral} \\
& \Box \\
& addC \rightarrow Hazard_2^{neutral} \\
& \Box \\
& addN_1 \rightarrow Hazard_2^{neutral} \\
& \Box \\
& tock \rightarrow Hazard_2^{neutral} \\
& \Box \\
& emptyTank \rightarrow Hazard_2^{safe}
\end{aligned}
$$

$$
\begin{aligned}
Hazard_2^{unsafe} \mathrel{\hat{=}} \; & addA \rightarrow Hazard_2^{unsafe} \\
& \Box \\
& addB \rightarrow Hazard_2^{unsafe} \\
& \Box \\
& addC \rightarrow Hazard_2^{unsafe} \\
& \Box \\
& addN_1 \rightarrow Hazard_2^{neutral} \\
& \Box \\
& tock \rightarrow \underline{reactionX} \rightarrow \perp_{\alpha Hazard_2^{safe}} \\
& \Box \\
& emptyTank \rightarrow \underline{reactionX} \rightarrow \perp_{\alpha Hazard_2^{safe}}
\end{aligned}
$$

When in a neutral state, the only change in behaviour can be brought about by an observation

of *emptyTank*, in which case the process $Hazard_2^{safe}$ is invoked. Observations of *addA*, *addB*, *addC*, $addN_1$ and *tock* leave the system in this neutral state.

If the system is in an unsafe state and the neutralising agent N_1 is not added within a minute, then the hazard $\underline{reactionX}$ will occur. Addition of N_1 results in a change to a neutral state. Further addition of A, B or C leaves the system in an unsafe state.

Note that we assume that emptying the tank when in an unsafe state will not prevent a hazardous reaction from occurring.

The process $Hazard_3^{safe}$ has the alphabet

$$\alpha Hazard_3^{safe} = \{addD, addN_2, tock, emptyTank, \underline{reactionY}\}$$

and is defined in the following way.

Initially, observations of $addN_2$, *tock* or *emptyTank* leave the process representing this hazard analysis in its initial state. Alternatively, if D is added to the system, then the process $Hazard_3^{unsafe_1}$ is invoked.

$$\begin{aligned} Hazard_3^{safe} \mathrel{\widehat{=}} & \; addD \rightarrow Hazard_3^{unsafe_1} \\ & \Box \\ & addN_2 \rightarrow Hazard_3^{safe} \\ & \Box \\ & tock \rightarrow Hazard_3^{safe} \\ & \Box \\ & emptyTank \rightarrow Hazard_3^{safe} \end{aligned}$$

Here, if $addN_2$ is observed, then the system returns to a safe state. If *tock* is observed, then the process $Hazard_3^{unsafe_2}$ is invoked. If *addD* is observed, then there is no change in state. Finally, if *emptyTank* is observed, then we assume that the hazard will occur.

$$\begin{aligned} Hazard_3^{unsafe_1} \mathrel{\widehat{=}} & \; addD \rightarrow Hazard_3^{unsafe_1} \\ & \Box \\ & addN_2 \rightarrow Hazard_3^{safe} \\ & \Box \\ & tock \rightarrow Hazard_3^{unsafe_2} \\ & \Box \\ & emptyTank \rightarrow \underline{reactionY} \rightarrow \bot_{\alpha Hazard_3^{safe}} \end{aligned}$$

$$\begin{aligned} Hazard_3^{unsafe_2} \mathrel{\widehat{=}} & \; addD \rightarrow Hazard_3^{unsafe_2} \\ & \Box \\ & addN_2 \rightarrow Hazard_3^{safe} \\ & \Box \\ & tock \rightarrow \underline{reactionY} \rightarrow \bot_{\alpha Hazard_3^{safe}} \\ & \Box \\ & emptyTank \rightarrow \underline{reactionY} \rightarrow \bot_{\alpha Hazard_3^{safe}} \end{aligned}$$

Finally, we define $Hazard_4^{safe}$ in the following way.

$$\alpha Hazard_4^{safe} = \{addE, \underline{fumes}\}$$
$$Hazard_4^{safe} \mathrel{\widehat{=}} addE \rightarrow \underline{fumes} \rightarrow \perp_{\alpha Hazard_4^{safe}}$$

It follows that we define *Analysis* thus:

$$Analysis \mathrel{\widehat{=}} Hazards \parallel MissionInit$$

This process provides a full description of the system's behaviour with respect to its functional behaviour and its hazard analysis. If this process leads to $\perp$, then the functional specification is unsafe with respect to its hazards. Alternatively, if it is deterministic (therefore, not leading to $\perp$), then we can conclude that the system is safe.

Via FDR, we can test such properties of CSP processes quickly and efficiently. For a small problem like this, it is possible to reason by observation and some simple calculations that this process is safe with respect to its hazards. However, when we deal with larger systems in which hazard analyses are complex, a structured mechanical approach to the analysis of hazards is essential. As FDR allows us to check (relatively) large processes for determinism, we can claim that it is possible for this approach to 'scale up'.

Thus, our general approach to hazard analysis with CSP is as follows. During the early stages of development, we might test a specification against CSP processes representing hazard analyses. Via FDR, we can detect possible behaviours leading to hazards, and develop means of preventing such behaviours occurring. When we can prove that our specification will never allow the possibility of a hazard, we can move towards an implementation safe in the knowledge that any further refinements of our specification (and thus any implementation) will not allow hazardous behaviour.

Proposition 2.2 Given two specifications $Spec_1$ and $Spec_2$ such that $Spec_1 \sqsubseteq Spec_2$, and a hazard analysis *HA*,

$$\mathbf{det}(Spec_1 \parallel HA) \Rightarrow \mathbf{det}(Spec_2 \parallel HA)$$

Proof. Assume $\mathbf{det}(Spec_1 \parallel HA)$ and $Spec_1 \sqsubseteq Spec_2$. As failures-divergences refinement is preserved within context, and any refinement of a deterministic process is also deterministic, it follows that $\mathbf{det}(Spec_2 \parallel HA)$. □

2.2.7 Determinism versus Refinement

In the above example, the processes *MissionInit* and *Hazards* fulfill two very different roles. The former is a functional specification, whereas the latter is a hazard analysis, against which we verify that *MissionInit* is safe with respect to its hazards. Specifically, if the process *Analysis* is deterministic, then we can conclude that any process which refines our functional specification can never lead to a (known) hazard. This is due to the fact that if *MissionInit* could allow a hazard, the process *Hazards* would lead to $\perp$, resulting in a nondeterministic *Analysis*.

One disadvantage of the above approach is that the mechanical verification of such analyses is extremely complex when dealing with systems involving many hazards. Of course,

we can counter this problem by analysing a specification against each hazard in turn, and proving that it is deterministic with respect to each. For example, given the set of hazards

$$H = \{Hazard_1^{safe}, Hazard_2^{safe}, Hazard_3^{safe}, Hazard_4^{safe}\}$$

we might endeavour to prove

$$\forall\, h : H \bullet \mathbf{det}(MissionInit \parallel h)$$

The above should not be thought of as simply introducing 'artificial non-determinism' to test for determinism-based conditions. The fact is, the task of proving the absence of hazardous traces by refinement is not an easy exercise.

Take, as an example, the following safety requirement of a laser control system.

If the laser is fired, then the safety interlock is on.

Assuming that the safety interlock is initially off, we might define the process *LaserReq*.

$$LaserReq \mathrel{\widehat{=}} interlockOn \rightarrow LaserReq'$$
$$LaserReq' \mathrel{\widehat{=}} interlockOff \rightarrow LaserReq \,\Box\, fireLaser \rightarrow LaserReq'$$

Given a specification *LaserSpec* of our system, we denote the process which insists on this safety requirement and is unconcerned with the behaviour of all other events of $\alpha LaserSpec$ by $LaserReq \,[\![\, \{interlockOn, interlockOff, fireLaser\} \,]\!]\, CHAOS_\Sigma$. If this process is refined by *LaserSpec*, then we can conclude that the specification *LaserSpec* satisfies this requirement. But is this what our requirement states? This process only denotes the acceptable behaviours of the events of $\alpha LaserReq$ with respect to our interpretation of the stated requirement.

This approach presents several problems. First, it is only by examination that we can determine that the CSP interpretation of this requirement is correct. It is obviously true for a trivial example such as this, but we have no formal means of accomplishing this for complex requirements statements. Secondly, the task of building a CSP process which is representative of the acceptable behaviours of relevant events is a complex task for anything but the simplest of safety requirements. Finally, what if our specification will not allow the laser to be fired under any circumstances? In this case, under a logical interpretation of our safety requirement, the property is satisfied trivially as the event *fireLaser* cannot be observed by the specification. However, under the failures-divergences semantics, the system would not be a refinement of *LaserReq*.

An alternative approach to the analysis of safety properties is to write a process such as *LaserHazard*, which is defined thus:

$$\begin{aligned} LaserHazard \mathrel{\widehat{=}}\ & interlockOff \rightarrow LaserHazard \\ & \Box \\ & interlockOn \rightarrow LaserHazard' \\ & \Box \\ & fireLaser \rightarrow \bot_{\{interlockOn, interlockOff, fireLaser\}} \end{aligned}$$

$$\begin{aligned} LaserHazard' \mathrel{\widehat{=}} & \ interlockOff \rightarrow LaserHazard \\ & \Box \\ & interlockOn \rightarrow LaserHazard' \\ & \Box \\ & fireLaser \rightarrow LaserHazard' \end{aligned}$$

If $\mathbf{det}(LaserHazard \parallel LaserSpec)$ holds for a specification *LaserSpec*, then we can conclude that the safety requirement is satisfied by that process.

An advantage of this approach is that it is easier to relate such a hazard analysis to the stated requirement — *if* this sequence of events is observed, *then* a hazard might occur. This is a far easier task than attempting to define a process which represents the acceptable behaviours of a particular set of events and then testing for refinement. Furthermore, this approach does not force one to impose a (possibly incorrect) interpretation of the correct behaviour of these events when writing the CSP requirement.

Unfortunately, this approach introduces a drawback of its own — the need for deterministic specifications. After all, a nondeterministic system which guarantees the safety requirement might fail this test. One might argue that implementations should be deterministic with respect to safety anyway. After all, we would not wish there to be circumstances under which the behaviour of a safety-critical system is unpredictable.

One method of combatting this drawback is to test for nondivergence rather than determinism. Specifically, if a specification *Spec* composed with a hazard analysis *HA* is nondivergent, then we can conclude that $\perp$ (and therefore the hazard(s) described by *HA*) is avoided by the specification. This property is also preserved by refinement.

Proposition 2.3 Given two specifications $Spec_1$ and $Spec_2$ such that $Spec_1 \sqsubseteq Spec_2$, and a hazard analysis *HA*,

$$\mathbf{nondiv}(Spec_1 \parallel HA) \Rightarrow \mathbf{nondiv}(Spec_2 \parallel HA)$$

Proof. Assume $\mathbf{nondiv}(Spec_1 \parallel HA)$ and $Spec_1 \sqsubseteq Spec_2$. As failures-divergences refinement is preserved within context,

$$Spec_1 \parallel HA \sqsubseteq Spec_2 \parallel HA$$

As $Spec_1 \parallel HA$ is nondivergent, we can conclude that $\mathbf{nondiv}(Spec_2 \parallel HA)$. □

2.3 Protection

In the previous section, we saw how determinism can be used to prove the absence of hazardous traces from specifications. In this section, we shall see how the determinism-based analysis of non-interference can be applied to reason about a number of properties of safety-critical systems.

As an example, assume that a signal is prone to some sort of failure. If we are aware of, and have a means of working around, this failure, then we might develop a specification in which the system behaves in such a way that this failure does not affect the system's safety-critical events. If such a property holds, then we shall say that the system's safety-critical

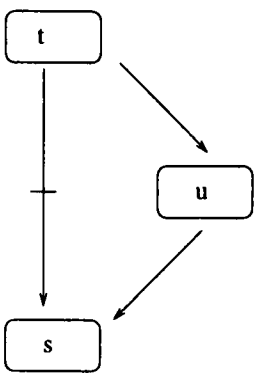

Figure 2.11: Information flow (1).

events are *protected* from its failures. In this section, we define such a notion of safety-critical non-interference which is based upon lazy independence.

Assume a process P and disjoint sets of events s, t, and u. By denoting non-interference between sets of events in P by $\not\leftarrow$ and interference by $\leftarrow$, information is prevented from flowing from t to s directly in Figure 2.11. However, there is the possibility of *secondary interference* — information flowing from t to u, and subsequently from u to s.

To prove non-interference, it is enough to establish that there is no direct or indirect information flow from t to s, i.e.,

- if there is information flow into s, then there is no information flow from t (Figure 2.12), and
- if there is information flow from t, then there is no information flow into s (Figure 2.13).

Recall that the independence properties of Section 2.1 are defined in terms of disjoint sets of events L and H which partition the whole of αP. In the following, we shall often need to reason about non-interference between sets of events $s, s' \in \mathbb{P}\Sigma$, such that $s \cup s' \subset \alpha P$. It follows that we need to define some way of expressing non-interference between such subsets.

Our first task is to decide on which notion of independence we should concentrate. Intuitively, the fact that strong independence is equivalent to the conjunction of the eager and lazy properties leads us to find that property appealing. However, consider the process

$$P \mathrel{\widehat{=}} u_H \;|||\; u_L$$

such that αu_H, denoted H, and αu_L, denoted L, partition αP.

Despite the fact that there is (intuitively) no information flow between u_H and u_L — the alphabets of the two processes are disjoint — the properties $E\text{-}Ind_H(P)$ and $S\text{-}Ind_H(P)$ both fail to hold because of the possibility of the observation of an infinite sequence of H events in P. Thus, given processes such as u_H and u_L which are composed via interleaving, we will not be able to say that the alphabet of one is non-interfering with the other, which is fundamental to a number of non-interference properties which we wish to analyse in the following. Therefore, we shall concentrate on the lazy property, with respect to which, independence

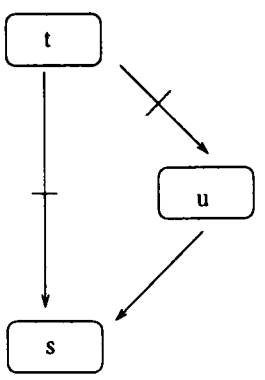

Figure 2.12: Information flow (2).

holds of the above, i.e., $L\text{-}Ind_H(P)$.[4]

Our interpretation of non-interference is as follows: Given a process P and disjoint subsets $s, s' \subseteq \alpha P$, we need to find two disjoint sets $t, u \subseteq \alpha P$ which partition the whole of αP, such that $s \subseteq t$, $s' \subseteq u$, and u is non-interfering with t in P.

Given this interpretation, we define non-interference in the following way.

Definition 2.5 The set of events s' is *non-interfering* with another set of events s in the process P, denoted $s \nLeftarrow s'\ [P]$, if and only if

$$\alpha P \subseteq s \cup s' \land s \cap s' = \varnothing \land \mathbf{det}(P \,|||\, RUN_{s'})$$

□

That is, s' is non-interfering with s in P if and only if s and s' are disjoint sets of events which partition the whole of αP and lazy independence holds of P with respect to s'. We write $s \Leftarrow s'\ [P]$ if s' is interfering with s in P.

We define our property of protection in the following way.

Definition 2.6 The set of events s_1 is *protected* from another set of events s_2 in the process P, denoted $s_1 \dashv\!\!+ s_2\ [P]$, if and only if

$$\exists\, s_3 : \mathbb{P}\,\Sigma \bullet s_1 \cup s_3 \nLeftarrow (\alpha P - (s_1 \cup s_3)) \cup s_2\ [P]$$

□

That is, s_1 is protected from s_2 in P if and only if the events contained in the sets αP, s_1, and s_2 can be partitioned into disjoint sets, such that the set of events containing s_2 is non-interfering with the set of events containing s_1 in P. We write $s_1 \leftarrow\!\!- s_2\ [P]$ if s_1 is not protected from s_2 in P.

As a means of illustrating this definition, consider Figure 2.14. Here, αP consists of subsets s_1, s_2 and (possibly) other events not included in either set. We assume that s_3 is a (possibly empty) subset of these other events. In order for protection to hold, there must be

[4]However, it should be noted that we shall often find it convenient to define non-interference in terms of hybrid conditions involving both concealment and interleaving.

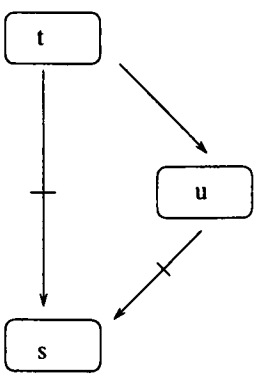

Figure 2.13: Information flow (3).

some partition of these 'other events', such that P is lazily independent with respect to s_2, as well as αP less the union of s_1 and s_3. This is illustrated by Figure 2.15.

The following proposition, while appearing somewhat trivial, helps to formulate the relationship between Definitions 2.5 and 2.6, and states that non-interference implies protection. It is also one of the key steps to the development of the proof system of the following chapter.

Proposition 2.4 $\forall P : PROC;\ s_1, s_2 : \mathbb{P}\,\Sigma \bullet s_1 \not\Leftarrow s_2\ [P] \Rightarrow s_1 \dashv\!\!\!+ s_2\ [P]$

Proof. Assume $s_1 \not\Leftarrow s_2\ [P]$. It follows that

$$s_1 \cup \varnothing \not\Leftarrow \alpha P - (s_1 \cup \varnothing) \cup s_2$$

Therefore, by Definition 2.6, $s_1 \dashv\!\!\!+ s_2\ [P]$. □

We present the following example as a means of motivating and highlighting the differences between these definitions.

Assume that a file can be accessed by Alan, Bill and Charlie. Alan and Bill, who can both read and write to the file, can access the file via $View_{low}$. In addition, Bill is kept informed of Alan's actions by reading the file after either *AlanRead* or *AlanWrite* is observed. This process is defined thus:

$$\begin{aligned}
View_{low} \mathrel{\hat{=}}\ & BillRead \rightarrow AlanRead \rightarrow BillRead \rightarrow Skip \\
& \quad \Box \\
& \quad AlanWrite \rightarrow BillRead \rightarrow Skip \\
& \Box \\
& BillWrite \rightarrow AlanRead \rightarrow BillRead \rightarrow Skip \\
& \quad \sqcap \\
& \quad AlanWrite \rightarrow BillRead \rightarrow Skip \\
& \Box \\
& AlanRead \rightarrow BillRead \rightarrow Skip \\
& \Box \\
& AlanWrite \rightarrow BillRead \rightarrow Skip
\end{aligned}$$

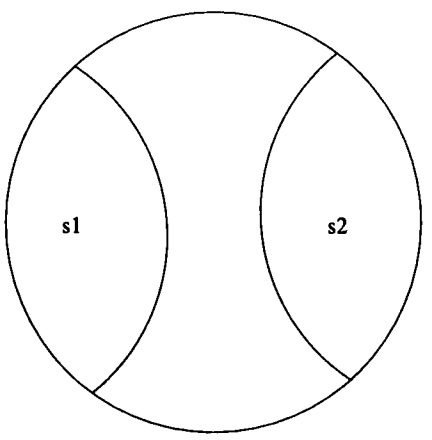

Figure 2.14: Partitioning αP : Part 1.

The process $View_{high}$ on the other hand, deals with Bill and Charlie. Charlie needs only to read the file. This process is defined thus:

$$\begin{aligned} View_{high} \mathrel{\widehat{=}} & \; CharlieRead \rightarrow BillRead \rightarrow (BillRead \rightarrow Skip \;\Box\; Skip) \\ & \Box \\ & BillRead \rightarrow (BillRead \rightarrow Skip \;\Box\; Skip) \end{aligned}$$

We denote the composition of these processes by

$$File \mathrel{\widehat{=}} View_{low} \;[\![\; \{BillRead, BillWrite\} \;]\!]\; View_{high}$$

Assume that $cl(Alan) \preceq cl(Bill) \preceq cl(Charlie)$, and that the alphabets of these processes are given by the following sets.

$$\begin{aligned} \alpha Alan &= A = \{AlanRead, AlanWrite\} \\ \alpha Bill &= B = \{BillRead, BillWrite\} \\ \alpha Charlie &= C = \{CharlieRead\} \end{aligned}$$

Lazy independence fails to hold of process $View_{low}$ with respect to B because the type of choice offered between A events is dependent upon which B event is chosen. Therefore, by Definitions 2.5 and 2.6, $A \leftarrow\!\!\!- B\ [View_{low}]$. On the other hand,

$$\alpha View_{high} \subseteq B \cup C \wedge B \cap C = \varnothing \wedge \mathbf{det}(View_{high} \;|||\; RUN_C)$$

Therefore, by Definition 2.5, $B \not\Leftarrow C\ [View_{high}]$ and by Proposition 2.4, $B \not\leftarrow C\ [View_{high}]$. Furthermore, A and $\alpha View_{high}$ are disjoint, as are A and C. Therefore,

$$\alpha View_{high} \subseteq (A \cup B) \cup C \wedge (A \cup B) \cap C = \varnothing \wedge \mathbf{det}(View_{high} \;|||\; RUN_C)$$

Again, by Definition 2.5 and Proposition 2.4, $A \cup B \not\leftarrow C\ [View_{high}]$.

In addition, $A \cup B \not\Leftarrow C\ [File]$ and $A \not\Leftarrow B \cup C\ [File]$. Therefore, by Definition 2.6,

$$A \not\leftarrow B\ [File] \wedge A \not\leftarrow C\ [File] \wedge B \not\leftarrow C\ [File]$$

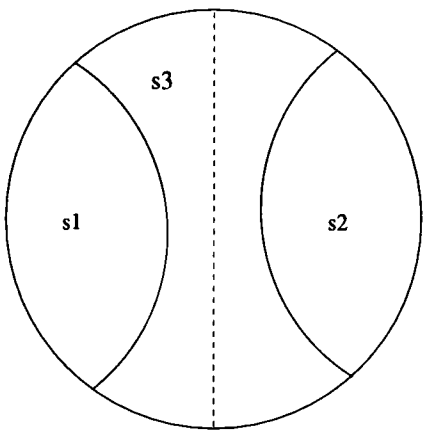

Figure 2.15: Partitioning αP : Part 2.

As a further example, recalling our chemical reactor system, the addition of chemicals A, B and C (safety-critical events) are all protected from the system's hazards in *Analysis*, i.e.,

$$\{addA, addB, addC\} \dashv\!\!+ Hazards\ [Analysis]$$

Below are three simple propositions concerning our property of protection. First, if a process P is nondivergent, then the empty set is protected from every set of events in P.

Proposition 2.5 $\forall P : PROC;\ s : \mathbb{P}\,\Sigma \bullet \mathbf{nondiv}(P) \Rightarrow \varnothing \dashv\!\!+ s\ [P]$

Proof. If **nondiv**(P), then

$$P \mid\mid\mid RUN_\Sigma \equiv RUN_\Sigma$$

which is deterministic. Trivially,

$$\alpha P \subseteq \varnothing \cup \Sigma \wedge \varnothing \cap \Sigma = \varnothing \wedge \mathbf{det}(P \mid\mid\mid RUN_\Sigma)$$

By Definition 2.5, $\varnothing \not\dashv\!\!+ \Sigma\ [P]$. Given any $s \in \mathbb{P}\,\Sigma$, by Definition 2.6, $\varnothing \dashv\!\!+ s\ [P]$.

Finally, by $\Rightarrow$ introduction and $\forall$ introduction,

$$\forall P : PROC;\ s : \mathbb{P}\,\Sigma \bullet \mathbf{nondiv}(P) \Rightarrow \varnothing \dashv\!\!+ s\ [P]$$

□

Our next proposition states that if a process P is deterministic, then every set of events is protected from the empty set in P.

Proposition 2.6 $\forall P : PROC;\ s : \mathbb{P}\,\Sigma \bullet \mathbf{det}(P) \Rightarrow s \dashv\!\!+ \varnothing\ [P]$

Proof. For any process P,

$$P \mid\mid\mid RUN_\varnothing \equiv P$$

If $\mathbf{det}(P)$, then $\mathbf{det}(P \mid\mid\mid RUN_{\varnothing})$. Trivially,

$$\alpha P \subseteq \Sigma \cup \varnothing \wedge \Sigma \cap \varnothing = \varnothing \wedge \mathbf{det}(P \mid\mid\mid RUN_{\varnothing})$$

By Definition 2.5, $\Sigma \not\Leftarrow \varnothing\ [P]$. Given any $s \in \mathbb{P}\,\Sigma$, by Definition 2.6, $s \nleftarrow \varnothing\ [P]$.

Finally, by $\Rightarrow$ introduction and $\forall$ introduction,

$$\forall\, P : PROC;\ s : \mathbb{P}\,\Sigma \bullet \mathbf{det}(P) \Rightarrow s \nleftarrow \varnothing\ [P]$$

□

Finally, if two sets of events have elements in common, then there is interference between them in any process $P \in PROC$, even if neither set forms part of that process's alphabet.

Proposition 2.7 $\forall\, P : PROC;\ s_1, s_2 : \mathbb{P}\,\Sigma \bullet s_1 \cap s_2 \neq \varnothing \Rightarrow s_1 \longleftarrow s_2\ [P]$

Proof. Assume $s_1 \cap s_2 \neq \varnothing$. By Definition 2.6, $s_1 \nleftarrow s_2\ [P]$ if and only if

$$\exists\, s_3 : \mathbb{P}\,\Sigma \bullet s_3 \cup s_1 \not\Leftarrow (\alpha P - (s_3 \cup s_1)) \cup s_2\ [P]$$

By Definition 2.5, if there are events common to s_1 and s_2, then it follows that we cannot find disjoint sets s and s' such that $s_1 \subseteq s$ and $s_2 \subseteq s'$. Therefore, $s_1 \longleftarrow s_2\ [P]$.

Finally, by $\Rightarrow$ introduction and $\forall$ introduction,

$$\forall\, P : PROC;\ s_1, s_2 : \mathbb{P}\,\Sigma \bullet s_1 \cap s_2 \neq \varnothing \Rightarrow s_1 \longleftarrow s_2\ [P]$$

□

In the next chapter, we investigate further properties of, and develop a proof system for, protection. This will aid us in Chapters 4 and 5, in which we reason about a number of safety-critical properties in terms of non-interference.

Chapter 3

A Proof System for Protection

In the previous chapter, we saw how non-interference can be described in terms of deterministic CSP processes. We also saw how determinism could be used to define a property called protection, which allows us to reason about non-interference in safety-critical systems. In this chapter, we investigate the theory which underlies protection, and develop a proof system for it. We start by examining a number of properties of protection and introduce a collection of laws which enable us to develop, via compositional methods, systems of which we can guarantee that there is relevant non-interference. Finally, we prove that protection is preserved by refinement.

3.1 Some Properties of Protection

We shall often find it useful to add or remove elements from two sets of events between which protection holds. For the case of removing elements, we introduce the following laws, which state that if protection holds between sets of events s_1 and s_2 in any process $P \in PROC$, then it also holds between all subsets of s_1 and s_2 in that process.

Note that proofs of all of the laws introduced in this chapter are given in Appendix A.

Our first two laws (given as Law A.1 and Law A.2 in Appendix A), along with Proposition 2.4 of the previous chapter, play a vital role in the proofs of our laws of protection which are described in this chapter. At their simplest, they state that, given any sets of events s_1 and s_2, if s_1 is protected from s_2 in a process P, then all subsets of s_1 are protected from all subsets of s_2 in that process.

$$\frac{s_1 \nrightarrow s_2 \ [P] \qquad s_3 \subseteq s_1}{s_3 \nrightarrow s_2 \ [P]}$$

$$\frac{s_1 \nrightarrow s_2 \ [P] \qquad s_3 \subseteq s_2}{s_1 \nrightarrow s_3 \ [P]}$$

We combine the above to produce the following law.

$$\frac{s_1 \leftarrow\!\!+ s_2\ [P] \qquad s_3 \subseteq s_1 \qquad s_4 \subseteq s_2}{s_3 \leftarrow\!\!+ s_4\ [P]}$$

Having introduced laws for removing elements from sets of events, between which protection holds, we now determine the conditions necessary for adding elements to such sets.

We first introduce two new pieces of notation. We start by defining a function which denotes the initials of a process P, i.e., those elements of αP which may be performed initially.

Definition 3.1 The *initials* of a process P, denoted $inits(P)$, are defined by the set

$$inits(P) = \{e : \Sigma \mid \langle e \rangle \in traces\,[\![P]\!]\}$$

□

The *refusals* of a process P are defined similarly.

Definition 3.2 The *refusals* of a process P, denoted $\overline{inits}(P)$, are defined by the set

$$\overline{inits}(P) = \{e : \Sigma \mid (\langle\rangle, \{e\}) \in failures\,[\![P]\!]\}$$

□

As an example, we define P_7,

$$P_7 \mathrel{\hat{=}} addA \rightarrow addB \rightarrow Skip \,\Box\, addC \rightarrow addD \rightarrow Skip$$

the initials and refusals of which are given by

$$inits(P_7) = \{addA, addC\}$$
$$\overline{inits}(P_7) = \Sigma - \{addA, addC\}$$

Furthermore, following [Hoa85], we introduce the **after** operator, which describes the behaviour of a process P following the observation of some trace $tr \in traces\,[\![P]\!]$. Note that P **after** tr is undefined if $tr \notin traces\,[\![P]\!]$.

We now provide a formal interpretation of lazy independence in terms of **after** .

Proposition 3.1

$$\begin{aligned}
&\forall\, P : PROC;\ s : \mathbb{P}\,\Sigma;\ t_1, t_2 : \mathrm{seq}\,\Sigma;\ e : \Sigma \bullet \\
&\quad \mathbf{det}(P \mid\mid\mid RUN_s) \\
&\quad \Leftrightarrow \\
&\quad \mathbf{nondiv}(P) \\
&\quad \wedge \\
&\quad t_1 \frown t_2 \in traces\,[\![P]\!] \wedge t_1 \frown \langle e \rangle \frown t_2 \in traces\,[\![P]\!] \wedge e \in s \Rightarrow \\
&\qquad (\Sigma - s) \cap \overline{inits}(P \textbf{ after } t_1 \frown \langle e \rangle \frown t_2) = (\Sigma - s) \cap \overline{inits}(P \textbf{ after } t_1 \frown t_2) \\
&\quad \wedge \\
&\quad t_1 \frown t_2 \in traces\,[\![P]\!] \wedge t_1 \frown \langle e \rangle \frown t_2 \in traces\,[\![P]\!] \wedge e \notin s \Rightarrow \\
&\qquad (t_1, \{e\}) \notin failures\,[\![P]\!]
\end{aligned}$$

Proof. We start by proving the $\Rightarrow$ direction. Assume $\mathbf{det}(P \mid\mid\mid RUN_s)$. By Definition 1.1, $\mathbf{nondiv}(P \mid\mid\mid RUN_s)$. Trivially therefore, $\mathbf{nondiv}(P)$. By Definition 2.2, the future behaviour of the events of P with respect to the events of $\Sigma - s$, following an occurrence of some $e \in s$ in P, is identical to what it would have been had e not occurred. That is, $\mathbf{det}(P \mid\mid\mid RUN_s)$ implies

$$t_1 \frown t_2 \in traces[\![P]\!] \wedge t_1 \frown \langle e \rangle \frown t_2 \in traces[\![P]\!] \wedge e \in s \Rightarrow (\Sigma - s) \cap \overline{inits}(P \textbf{ after } t_1 \frown \langle e \rangle \frown t_2) = (\Sigma - s) \cap \overline{inits}(P \textbf{ after } t_1 \frown t_2)$$

As $\mathbf{det}(P \mid\mid\mid RUN_s)$, by Definition 1.1,

$$tr \frown \langle e \rangle \in traces[\![P \mid\mid\mid RUN_s]\!] \Rightarrow (tr, \{e\}) \notin failures[\![P \mid\mid\mid RUN_s]\!]$$

Assume $e \notin s$. There must be traces $t_1 \in traces[\![P]\!]$ and $t_2 \in traces[\![RUN_s]\!]$ such that

$$tr \frown \langle e \rangle \textbf{ interleaves } (t_1 \frown \langle e \rangle, t_2)$$

As $\mathbf{det}(P \mid\mid\mid RUN_s)$,

$$(tr, \{e\}) \notin failures[\![P \mid\mid\mid RUN_s]\!]$$

As $e \notin s$, $(t_1, \{e\}) \notin failures[\![P]\!]$.

We now prove the $\Leftarrow$ direction. Assume $\mathbf{nondiv}(P)$. It follows that RUN_s is nondivergent. Therefore, $\mathbf{nondiv}(P \mid\mid\mid RUN_s)$.

Assume $t_1 \frown \langle e \rangle \in traces[\![P \mid\mid\mid RUN_s]\!]$. There are two cases to consider: $e \in s$ and $e \notin s$.

Case 1 Assume $e \in s$. Here, $(t_1, \{e\}) \notin failures[\![P \mid\mid\mid RUN_s]\!]$ as $e \in s$ and, as such, can never be refused by $P \mid\mid\mid RUN_s$.

Case 2 Assume $e \notin s$. There must be traces $t \in traces[\![P]\!]$ and $u \in \mathrm{seq}\, s$ such that

$$t_1 \frown \langle e \rangle \textbf{ interleaves } (t \frown \langle e \rangle, u)$$

Given that

$$t \frown \langle e \rangle \in traces[\![P]\!] \wedge e \notin s \Rightarrow (t, \{e\}) \notin failures[\![P]\!]$$

and, following the observation of any event $a \in s$, the behaviour of P with respect to those events contained in the set $\Sigma - s$ is the same as it would have been had a not occurred, it follows that $(t_1, \{e\}) \notin failures[\![P \mid\mid\mid RUN_s]\!]$.

By Definition 1.1, $\mathbf{det}(P \mid\mid\mid RUN_s)$. □

Note how this corresponds to our informal interpretation of lazy independence — if some event e from s occurs, then the interface available to those events not in s is identical to what it would have been had e not occurred.

The following propositions are corollaries to the above. Whilst being trivial, they play an important role in the following and are stated as such.

Proposition 3.2 $\forall P : PROC;\ s : \mathbb{P}\,\Sigma \mid s \cap \alpha P = \varnothing \bullet \mathbf{det}(P) \Rightarrow \mathbf{det}(P \,|||\, RUN_s)$

Proof. Trivial. □

Proposition 3.3 $\forall P : PROC;\ s : \mathbb{P}\,\Sigma \bullet \mathbf{nondiv}(P) \Rightarrow \mathbf{det}(P \,|||\, RUN_{\alpha P \cup s})$

Proof. Trivial. □

We use the above to establish that if s_2 and s_3 are both non-interfering with s_1 in any process $P \in PROC$, then s_1 is protected from the union of s_2 and s_3 in that process.

$$\frac{s_1 \not\Leftarrow s_2\ [P] \qquad s_1 \not\Leftarrow s_3\ [P]}{s_1 \nleftarrow s_2 \cup s_3\ [P]}$$

Finally, if the set of events s_3 is non-interfering with both s_1 and s_2 in the process P, then the union of s_1 and s_2 is protected from s_3 in P.

$$\frac{s_1 \not\Leftarrow s_3\ [P] \qquad s_2 \not\Leftarrow s_3\ [P]}{s_1 \cup s_2 \nleftarrow s_3\ [P]}$$

3.2 Algebraic Properties

In this section, we shall prove that our property of protection is not transitive, reflexive or symmetric. We start by proving that $\nleftarrow$ is not transitive.

Proposition 3.4 $\nleftarrow$ is not transitive.

Proof. The relation $\nleftarrow$ is transitive if and only if

$$\forall P : PROC;\ s_1, s_2, s_3 : \mathbb{P}\,\Sigma \bullet s_1 \nleftarrow s_2\ [P] \wedge s_2 \nleftarrow s_3\ [P] \Rightarrow s_1 \nleftarrow s_3\ [P]$$

To disprove this, it suffices to exhibit a process P, together with disjoint subsets of events s_1, s_2, and s_3 such that

$$s_1 \nleftarrow s_2\ [P] \wedge s_2 \nleftarrow s_3\ [P] \wedge s_1 \not\leftarrow s_3\ [P]$$

Assuming the sets of events

$$A = \{AlanRead, AlanWrite\}$$
$$B = \{BillRead, BillWrite\}$$
$$C = \{CharlieRead, CharlieWrite\}$$

P_8 is such a process.

$$\begin{aligned} P_8 \mathrel{\widehat{=}} & \mathit{CharlieRead} \rightarrow P_8' \\ & \Box \\ & \mathit{CharlieWrite} \rightarrow P_8'' \\ & \Box \\ & \mathit{BillRead} \rightarrow P_8 \\ & \Box \\ & \mathit{AlanRead} \rightarrow P_8 \\ & \Box \\ & \mathit{AlanWrite} \rightarrow P_8 \end{aligned}$$

$$P_8' \mathrel{\widehat{=}} \mathit{CharlieRead} \rightarrow P_8' \Box \mathit{BillRead} \rightarrow P_8' \Box \mathit{AlanRead} \rightarrow P_8$$

$$P_8'' \mathrel{\widehat{=}} \mathit{CharlieWrite} \rightarrow P_8'' \Box \mathit{BillRead} \rightarrow P_8'' \Box \mathit{AlanWrite} \rightarrow P_8$$

Here, $A \dashv\!\!+ B\ [P_8]$, $B \dashv\!\!+ C\ [P_8]$, and $A \longleftarrow C\ [P_8]$. Therefore, $\dashv\!\!+$ is not transitive. □

Proposition 3.5 $\dashv\!\!+$ is not reflexive.

Proof. The relation $\dashv\!\!+$ is reflexive if and only if

$$\forall P : \mathit{PROC};\ s : \mathbb{P}\,\Sigma \bullet s \dashv\!\!+ s\ [P]$$

From Proposition 2.7,

$$s_1 \cap s_2 \neq \varnothing \Rightarrow s_1 \longleftarrow s_2\ [P]$$

Hence,

$$s \neq \varnothing \Rightarrow s \longleftarrow s\ [P]$$

Therefore, $\dashv\!\!+$ is not reflexive. □

Proposition 3.6 $\dashv\!\!+$ is not symmetric.

Proof. The relation $\dashv\!\!+$ is symmetric if and only if

$$\forall P : \mathit{PROC};\ s_1, s_2 : \mathbb{P}\,\Sigma \bullet s_1 \dashv\!\!+ s_2\ [P] \Rightarrow s_2 \dashv\!\!+ s_1\ [P]$$

As a counter-example, take process P_9.

$$P_9 \mathrel{\widehat{=}} \mathit{AlanRead} \rightarrow P_9 \Box \mathit{BillRead} \rightarrow \mathit{AlanRead} \rightarrow P_9$$

Here, $A \dashv\!\!+ B\ [P_9]$ and $B \longleftarrow A\ [P_9]$. Therefore, $\dashv\!\!+$ is not symmetric. □

It can also be shown that $\dashv\!\!+$ is not anti-symmetric or asymmetric.

We can also attempt to establish properties for information flow. We start by proving that, counter-intuitively, our notion of information flow is not transitive.

Proposition 3.7 $\longleftarrow$ is not transitive.

Proof. The relation $\longleftarrow$ is transitive if and only if

$$\forall P : PROC;\ s_1, s_2, s_3 : \mathbb{P}\,\Sigma \bullet s_1 \longleftarrow s_2\ [P] \wedge s_2 \longleftarrow s_3\ [P] \Rightarrow s_1 \longleftarrow s_3\ [P]$$

Take as a counter-example, process P_{10}, which is defined by

$$P_{10} \mathrel{\widehat{=}} AlanRead \rightarrow P_{10} \,\Box\, BillRead \rightarrow P_{10}$$

together with the sets of events

$$\begin{aligned} View_{low} &= \{AlanRead\} \\ View_{mid} &= \{AlanRead, BillRead\} \\ View_{high} &= \{BillRead\} \end{aligned}$$

As $View_{low}$ and $View_{mid}$ are not disjoint, by Definitions 2.5 and 2.6, $View_{low} \longleftarrow View_{mid}\ [P_{10}]$. For similar reasons, $View_{mid} \longleftarrow View_{high}\ [P_{10}]$. However, $View_{low} \dashv\!\!+ View_{high}\ [P_{10}]$. Therefore, $\longleftarrow$ is not transitive. □

Proposition 3.8 $\longleftarrow$ is not reflexive.

Proof. The relation $\longleftarrow$ is reflexive if and only if

$$\forall P : PROC;\ s : \mathbb{P}\,\Sigma \bullet s \longleftarrow s\ [P]$$

By Definitions 2.5 and 2.6, $s \longleftarrow s$ if and only if

$$\nexists\, s' : \mathbb{P}\,\Sigma \bullet (s' \cup s) \Leftarrow ((\alpha P - (s' \cup s)) \cup s)$$

If $s = \varnothing$ and **nondiv**(P), then, by Proposition 3.3, we can exhibit such a set s' — the empty set. As such, the above fails to hold. Therefore, $\longleftarrow$ is not reflexive. □

Some notions of information flow are symmetric (e.g., that of [Sut86]). We establish that ours is not in the following way.

Proposition 3.9 $\longleftarrow$ is not symmetric.

Proof. The relation $\longleftarrow$ is symmetric if and only if

$$\forall P : PROC;\ s_1, s_2 : \mathbb{P}\,\Sigma \bullet s_1 \longleftarrow s_2\ [P] \Rightarrow s_2 \longleftarrow s_1\ [P]$$

As a counter-example, $B \longleftarrow A\ [P_9]$ and $A \dashv\!\!+ B\ [P_9]$. Therefore, $\longleftarrow$ is not symmetric. □

It can also be shown that $\longleftarrow$ is not anti-symmetric or asymmetric.

3.3 A Proof System

In this section, we define a number of laws for protection. We start by introducing a law for the event prefixing operator, $\rightarrow$.

Event Prefixing

Our non-interference law for the event prefixing operator can be stated in the following way. If the events of s_2 are non-interfering with those of s_1 in the process P and $e \notin s_2$, then s_2 is non-interfering with s_1 in $e \rightarrow P$.

$$\frac{s_1 \nLeftarrow s_2\ [P] \qquad e \notin s_2}{s_1 \nLeftarrow s_2\ [e \rightarrow P]}$$

Alternatively, if $e \in s_2$, then non-interference holds only if the behaviour of $e \rightarrow P$ with respect to the events of s_1 is identical to what it would have been if e had not been observed. Such a law is defined in terms of external choice. However, we first introduce a law for internal choice.

Internal Choice

The only condition under which a set of events s_2 is non-interfering with another set of events s_1 in a (nondivergent) process of the form $P \sqcap Q$ is if the sets are disjoint and if every event from this process's alphabet is contained in s_2. This intuition is formalised in the following law.

$$\frac{s_1 \cap s_2 = \varnothing \qquad \alpha(P \sqcap Q) \subseteq s_2 \qquad \mathbf{nondiv}(P \sqcap Q)}{s_1 \nLeftarrow s_2\ [P \sqcap Q]}$$

External Choice

In this section, we introduce a number of proof rules for protection for external choice.

$$\frac{s_1 \nLeftarrow s_2\ [P] \qquad s_1 \nLeftarrow s_2\ [Q] \qquad \forall\, e : \Sigma \bullet e \in \mathit{inits}(P) \cap \mathit{inits}(Q) \Rightarrow P \textbf{ after } \langle e \rangle = Q \textbf{ after } \langle e \rangle}{s_1 \nLeftarrow s_2\ [P \,\Box\, Q]}$$

$$\frac{s_1 \nLeftarrow s_2\ [P] \qquad e \notin \mathit{inits}(P)}{s_1 \nLeftarrow s_2\ [P \,\Box\, e \rightarrow P]}$$

$$\frac{s_1 \not\Leftarrow s_2\ [P] \quad s_1 \not\Leftarrow s_2\ [Q] \quad x, y \notin s_2 \quad x \neq y}{s_1 \not\Leftarrow s_2\ [x \to P \,\Box\, y \to Q]}$$

The above can be generalised to deal with indexed external choice in the following ways.

$$\frac{s_1 \not\Leftarrow s_2\ [P] \quad X \cap \mathit{inits}(P) = \varnothing}{s_1 \not\Leftarrow s_2 [(\Box_{x \in X}\, x \to P) \,\Box\, P]}$$

$$\frac{s_1 \not\Leftarrow s_2\ [P] \quad X \cap s_2 = \varnothing}{s_1 \not\Leftarrow s_2 [\Box_{x \in X}\, x \to P]}$$

Partial Interleaving

Assuming two processes P and Q, we let $\overline{\mathit{inits}}((P, Q)_A \textbf{ after } (t_1, t_2))$ denote the set of events which may be refused by $P \,[\![A]\!]\, Q$ after P has performed trace t_1 and Q has performed trace t_2. This function is defined in the following way.

$$\overline{\mathit{inits}}((\_,\_)\_ \textbf{ after } (\_,\_)) : \mathit{PROC} \times \mathit{PROC} \times \mathbb{P}\,\Sigma \times \operatorname{seq}\Sigma \times \operatorname{seq}\Sigma \nrightarrow \mathbb{P}\,\Sigma$$

$$\begin{aligned}
&\forall P, Q : \mathit{PROC};\ t_1, t_2 : \operatorname{seq}\Sigma;\ A : \mathbb{P}\,\Sigma \mid \\
&\quad t_1 \in \mathit{traces}\,[\![P]\!] \land t_2 \in \mathit{traces}\,[\![Q]\!] \land t_1 \upharpoonright A = t_2 \upharpoonright A \bullet \\
&\qquad \overline{\mathit{inits}}((P, Q)_A \textbf{ after } (t_1, t_2)) = \overline{\mathit{inits}}(P \textbf{ after } t_1) \cap A \\
&\qquad\qquad \cup \\
&\qquad\qquad \overline{\mathit{inits}}(Q \textbf{ after } t_2) \cap A \\
&\qquad\qquad \cup \\
&\qquad\qquad \overline{\mathit{inits}}(P \textbf{ after } t_1) \cap \overline{\mathit{inits}}(Q \textbf{ after } t_2)
\end{aligned}$$

In the special case of interleaving, we define

$$\overline{\mathit{inits}}((P, Q) \textbf{ after } (t_1, t_2)) = \overline{\mathit{inits}}((P, Q)_{\varnothing} \textbf{ after } (t_1, t_2))$$

We also introduce the following.

$$\_ [\![\_]\!] \_ : \operatorname{seq}\Sigma \times \mathbb{P}\,\Sigma \times \operatorname{seq}\Sigma \to \mathbb{P}(\operatorname{seq}\Sigma)$$

$$\begin{aligned}
t_1 \,[\![A]\!]\, t_2 &= t_2 \,[\![A]\!]\, t_1 \\
t_1 \,[\![A]\!]\, \langle\rangle &= \{t_1\} && \textbf{if } t_1 \upharpoonright A = \langle\rangle \\
\langle a_1 \rangle \frown t_1 \,[\![A]\!]\, \langle a_1 \rangle \frown t_2 &= \{tr : t_1 \,[\![A]\!]\, t_2 \bullet \langle a_1 \rangle \frown tr\} && \textbf{if } a_1 \in A \\
\langle a_1 \rangle \frown t_1 \,[\![A]\!]\, \langle a_2 \rangle \frown t_2 &= \{tr : (\langle a_1 \rangle \frown t_1) \,[\![A]\!]\, t_2 \bullet \langle a_2 \rangle \frown tr\} && \textbf{if } a_1 \in A \textbf{ and } a_2 \notin A \\
\langle a_1 \rangle \frown t_1 \,[\![A]\!]\, \langle a_2 \rangle \frown t_2 &= \{tr : (\langle a_1 \rangle \frown t_1) \,[\![A]\!]\, t_2 \bullet \langle a_2 \rangle \frown tr\} && \textbf{if } a_1, a_2 \notin A \\
&\quad \cup \\
&\quad \{tr : t_1 \,[\![A]\!]\, (\langle a_2 \rangle \frown t_2) \bullet \langle a_1 \rangle \frown tr\}
\end{aligned}$$

The above definitions are required for the proof of our non-interference law for partial interleaving, which is stated thus:

$$\frac{s_1 \nleftarrow s_2\ [P] \quad s_1 \nleftarrow s_2\ [Q] \quad \mathbf{det}(P \,[\![\, A \,]\!]\, Q)}{s_1 \nleftarrow s_2\ [P \,[\![\, A \,]\!]\, Q]}$$

Parallel Composition

$$\frac{s_1 \nleftarrow s_2\ [P] \quad s_1 \nleftarrow s_2\ [Q]}{s_1 \nleftarrow s_2\ [P \parallel Q]}$$

We define the following laws in terms of the above. In the first, if s_2 is non-interfering with s_1 in P, s_2 contains no events from the alphabet of Q and Q is deterministic, then s_2 is non-interfering with s_1 in the parallel composition $P \parallel Q$.

$$\frac{s_1 \nleftarrow s_2\ [P] \quad \alpha Q \subseteq s_1 \quad \mathbf{det}(Q)}{s_1 \nleftarrow s_2\ [P \parallel Q]}$$

The following states that if s_2 is non-interfering with s_1 in P, s_1 contains no events from αQ and Q is nondivergent, then s_2 is non-interfering with s_1 in the parallel composition $P \parallel Q$.

$$\frac{s_1 \nleftarrow s_2\ [P] \quad \alpha Q \subseteq s_2 \quad \mathbf{nondiv}(Q)}{s_1 \nleftarrow s_2\ [P \parallel Q]}$$

Interleaving

$$\frac{s_1 \nleftarrow s_2\ [P] \quad s_1 \nleftarrow s_2\ [Q] \quad \mathbf{det}(P \,|||\, Q)}{s_1 \nleftarrow s_2\ [P \,|||\, Q]}$$

Sequential Composition

Non-interference holds in a process of the form $P \fatsemi Q$ if it holds in both P and Q, and information flow cannot be introduced by $\checkmark$.

$$\frac{\begin{array}{c} s_1 \nLeftarrow s_2\ [P] \\ s_1 \nLeftarrow s_2\ [Q] \\ \forall\, t_1 : \mathit{traces}\,[\![P]\!];\ t_2 : \mathit{traces}\,[\![Q]\!] \mid t_1 \frown t_2 \in \mathit{traces}\,[\![P]\!] \land t_1 \frown t_2 \in \mathit{traces}\,[\![P \fatsemi Q]\!] \bullet \\ (\Sigma - s_2) \cap \overline{\mathit{inits}}(P \textbf{ after } t_1 \frown t_2) = (\Sigma - s_2) \cap \overline{\mathit{inits}}(Q \textbf{ after } t_2) \\ \forall\, tr : \mathrm{seq}\,\Sigma;\ e : \Sigma \bullet \\ tr \frown \langle\checkmark\rangle \in \mathit{traces}\,[\![P]\!] \land \langle e\rangle \in \mathit{traces}\,[\![Q]\!] \land e \notin s_2 \Rightarrow \\ (tr, \{e\}) \notin \mathit{failures}\,[\![P]\!] \end{array}}{s_1 \nLeftarrow s_2\ [P \fatsemi Q]}$$

Concealment

Recall from Chapter 2, that eager independence was based on the concealment of events, lazy independence was based on the interleaving of events, and strong independence was based on a combination of both properties. In [Ros95], a number of properties concerning the relationship between the different notions of abstraction are introduced. We reproduce one of those results in Appendix A to define the following law for concealment.

$$\frac{\begin{array}{c} s_1 \nLeftarrow s_2 \cup X\ [P] \\ \textbf{nondiv}(P \setminus X) \end{array}}{s_1 \nLeftarrow s_2\ [P \setminus X]}$$

Event Renaming

If s_2 is non-interfering with s_1 in the process P, then it follows that $f(s_2)$ is non-interfering with $f(s_1)$ in $f(P)$, provided that the function f is a total injection.

$$\frac{\begin{array}{c} s_1 \nLeftarrow s_2\ [P] \\ f \in \alpha P \rightarrowtail \Sigma \end{array}}{f(s_1) \nLeftarrow f(s_2)\ [f(P)]}$$

Interrupts

$$\frac{\begin{array}{c} s_1 \nLeftarrow s_2\ [P] \\ s_1 \nLeftarrow s_2\ [Q] \\ \forall\, t_1 : \mathit{traces}\,[\![P]\!];\ t_2 : \mathit{traces}\,[\![Q]\!] \mid t_1 \frown t_2 \in \mathit{traces}\,[\![P]\!] \bullet \\ \overline{\mathit{inits}}(Q \textbf{ after } t_2) \cap (\Sigma - s_2) = \overline{\mathit{inits}}(P \textbf{ after } t_1 \frown t_2) \cap (\Sigma - s_2) \\ \forall\, t_1 : \mathit{traces}\,[\![P]\!];\ e : \Sigma \mid \langle e\rangle \in \mathit{traces}\,[\![Q]\!] \bullet \\ e \notin s_2 \Rightarrow (t_1, \{e\}) \notin \mathit{failures}\,[\![P]\!] \end{array}}{s_1 \nLeftarrow s_2\ [P \triangle Q]}$$

after

$$\frac{\begin{array}{c} tr \in traces\,[\![P]\!] \\ s_1 \cap s_2 = \varnothing \\ \mathbf{nondiv}(P) \\ \forall\, t_1, t_2 : \mathrm{seq}\,\Sigma;\ e : \Sigma \bullet \\ t_1 \frown t_2 \in traces\,[\![P \textbf{ after } tr]\!] \wedge t_1 \frown \langle e \rangle \frown t_2 \in traces\,[\![P \textbf{ after } tr]\!] \wedge e \in s_2 \Rightarrow \\ (\Sigma - s_2) \cap \overline{inits}(P \textbf{ after } tr \frown t_1 \frown \langle e \rangle \frown t_2) = (\Sigma - s_2) \cap \overline{inits}(P \textbf{ after } tr \frown t_1 \frown t_2) \\ \quad \forall\, t_1, t_2 : \mathrm{seq}\,\Sigma;\ e : \Sigma \bullet \\ \quad t_1 \frown t_2 \in traces\,[\![P \textbf{ after } tr]\!] \wedge t_1 \frown \langle e \rangle \frown t_2 \in traces\,[\![P \textbf{ after } tr]\!] \wedge e \notin s_2 \Rightarrow \\ \quad (t_1, \{e\}) \notin failures\,[\![P \textbf{ after } tr]\!] \end{array}}{s_1 \Leftarrow s_2\ [P \textbf{ after } tr]}$$

3.4 An Example

As an illustration of our proof system, we take the following simple system.

A bank has a number of customers, drawn from the set

$$C = \{c_1, \ldots, c_n\}$$

An abstract specification of a customer's account is given by the process $ACCT$, which has the alphabet

$$\begin{array}{l} \alpha ACCT(c, n) = \{m : \mathbb{N} \bullet deposit.c.m\} \\ \quad \cup \\ \quad \{m : \mathbb{N} \bullet withdraw.c.m\} \\ \quad \cup \\ \quad \{m : \mathbb{N};\ x : \{mgr, c\} \bullet enquire.x.c.m\} \end{array}$$

At any time, any customer $c \in C$ can check the status of his account, deposit money, or withdraw any amount up to the value deposited in the account. The bank manager can also check the status of each account.

$$\begin{array}{l} ACCT(c, n) \mathrel{\hat{=}} deposit.c?m \rightarrow ACCT(c, n + m) \\ \quad \Box \\ \quad \Box_{m \in 0 \ldots n}\ withdraw.c.m \rightarrow ACCT(c, n - m) \\ \quad \Box \\ \quad enquire.mgr.c!n \rightarrow ACCT(c, n) \\ \quad \Box \\ \quad enquire.c.c!n \rightarrow ACCT(c, n) \end{array}$$

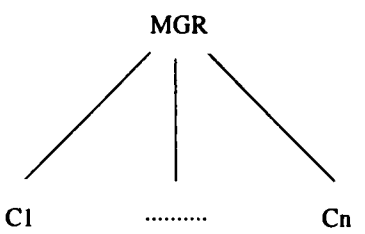

Figure 3.1: The bank.

Given any customer $c \in C$, we define c's interface by

$$\begin{aligned} \alpha c = \; & \{n : \mathbb{N} \bullet \mathit{withdraw.c.n}\} \\ & \cup \\ & \{n : \mathbb{N} \bullet \mathit{deposit.c.n}\} \\ & \cup \\ & \{n : \mathbb{N} \bullet \mathit{enquire.c.c.n}\} \end{aligned}$$

The bank manager can enquire about the state of an account at any time.

$$\begin{aligned} \alpha \mathit{MGR} &= \{n : \mathbb{N};\ c : C \bullet \mathit{enquire.mgr.c.n}\} \\ \mathit{MGR} &\mathrel{\hat{=}} \mathit{enquire.mgr?c?n} \rightarrow \mathit{MGR} \end{aligned}$$

We define the bank thus:

$$\mathit{Bank} \mathrel{\hat{=}} (\|_{c \in C} \mathit{ACCT}(c, 0)) \parallel \mathit{MGR}$$

We require that any specification of such a system should provide protection with respect to the partial order of Figure 3.1.

Proposition 3.10 $\forall\, c : C \bullet \alpha c \dashv\!\!+ \alpha \mathit{MGR}\ [\mathit{Bank}]$

Proof. Trivially,

$$\forall\, c : C \bullet \alpha \mathit{MGR} \subseteq \alpha c \cup (\alpha \mathit{Bank} - \alpha c) \wedge \alpha c \cap (\alpha \mathit{Bank} - \alpha c) = \varnothing$$

Trivially, $\mathbf{det}(\mathit{MGR} \mid\mid\mid \mathit{RUN}_{\alpha \mathit{Bank} - \alpha c})$. Therefore, by Definition 2.5,

$$\forall\, c : C \bullet \alpha c \not\dashv\!\!= \alpha \mathit{Bank} - \alpha c\ [\mathit{MGR}]$$

Once again by observation,

$$\forall\, c : C \bullet \alpha \mathit{ACCT}(c, 0) \subseteq \alpha c \cup (\alpha \mathit{Bank} - \alpha c) \wedge \alpha c \cap (\alpha \mathit{Bank} - \alpha c) = \varnothing$$

Trivially, $\mathbf{det}(\mathit{ACCT}(c, 0) \mid\mid\mid \mathit{RUN}_{\alpha \mathit{Bank} - \alpha c})$. Therefore, by Definition 2.5,

$$\forall\, c : C \bullet \alpha c \not\dashv\!\!= \alpha \mathit{Bank} - \alpha c\ [\mathit{ACCT}(c, 0)]$$

Given any $c' \in C$ such that $c' \neq c$, by observation,

$$\forall\, c : C \bullet \alpha ACCT(c', 0) \subseteq \alpha c \cup (\alpha Bank - \alpha c) \wedge \alpha c \cap (\alpha Bank - \alpha c) = \varnothing$$

By Proposition 3.3, $\mathbf{det}(ACCT(c', 0) \;|||\; RUN_{\alpha Bank - \alpha c})$. Therefore, by Definition 2.5,

$$\forall\, c, c' : C \mid c \neq c' \bullet \alpha c \not\Leftarrow \alpha Bank - \alpha c\ [ACCT(c', 0)]$$

By several applications of Law A.14,

$$\forall\, c : C \bullet \alpha c \not\Leftarrow \alpha Bank - \alpha c\ [Bank]$$

By Proposition 2.4,

$$\forall\, c : C \bullet \alpha c \nleftarrow \alpha Bank - \alpha c\ [Bank]$$

As $\alpha MGR \subseteq \alpha Bank - \alpha c$, by Law A.2,

$$\forall\, c : C \bullet \alpha c \nleftarrow \alpha MGR\ [Bank]$$

□

Proposition 3.11 $\forall\, c, c' : C \mid c \neq c' \bullet \alpha c \nleftarrow \alpha c'\ [Bank]$

Proof. We have already established that

$$\forall\, c : C \bullet \alpha c \nleftarrow \alpha Bank - \alpha c\ [Bank]$$

Given any $c' \in C$ such that $c' \neq c$, $\alpha c' \subseteq \alpha Bank - \alpha c$. Therefore, by Law A.2,

$$\forall\, c, c' : C \mid c \neq c' \bullet \alpha c \nleftarrow \alpha c'\ [Bank]$$

□

3.5 The Preservation of Protection

The property of lazy independence is shown to be preserved by refinement in [Wul94]. In this section, we show that our property of non-interference is similarly preserved.

We first introduce the following lemma.

Lemma 3.1 $\forall\, P, Q, R : PROC \bullet (\mathbf{det}(P \;|||\; Q) \wedge Q \sqsubseteq R) \Rightarrow (\mathbf{det}(P \;|||\; R))$

Proof. By virtue of the fact that failures-divergences refinement is preserved 'within context'.
□

Proposition 3.12

$$\forall P, Q : PROC;\ s_1, s_2 : \mathbb{P}\,\Sigma \bullet (s_1 \not\Leftarrow s_2\ [P] \land P \sqsubseteq Q) \Rightarrow s_1 \not\Leftarrow s_2\ [Q]$$

Proof. Assume $s_1 \not\Leftarrow s_2\ [P]$. By Definition 2.5,

$$\alpha P \subseteq s_1 \cup s_2 \land s_1 \cap s_2 = \varnothing \land \mathbf{det}(P \,|||\, RUN_{s_2})$$

Assume $P \sqsubseteq Q$. By Lemma 3.1,

$$\alpha P \subseteq s_1 \cup s_2 \land s_1 \cap s_2 = \varnothing \land \mathbf{det}(Q \,|||\, RUN_{s_2})$$

As $P \sqsubseteq Q$, it follows that $\alpha Q \subseteq \alpha P$. As such,

$$\alpha Q \subseteq s_1 \cup s_2 \land s_1 \cap s_2 = \varnothing \land \mathbf{det}(Q \,|||\, RUN_{s_2})$$

Finally, by Definition 2.5, $s_1 \not\Leftarrow s_2\ [Q]$. □

By Proposition 2.4, we can establish that protection is also preserved under refinement.

Having developed a proof system for protection and illustrated that it is preserved by refinement, we shall now endeavour to apply this property to the analysis and specification of a number of safety-critical concepts.

Chapter 4

Non-interference and Safety-critical Systems

In this chapter, we introduce a number of safety-critical properties which may be defined in terms of protection. We also present a number of examples as a means of illustrating these properties.

Work described in this chapter has previously appeared as [SW96] and [Sim96].

4.1 Mission-safety Independence

Requirements capture is one of the most important elements of safety-critical system development. To quote [Lev86],

> "Determining the requirements for software ... is a major source of software problems and may be the most important with respect to safety."

It is often the case that requirements for a safety-critical system may be divided into two categories: *mission* (or *functional*) requirements and *safety* requirements. The former describe how a system should behave in the absence of failures and "focus on what the system is supposed to achieve in terms of function, timeliness, and some dependability requirements," [dLSA92] while the latter describe how a system should behave in the presence of failures and "focus on the elimination and control of hazards, and the limitation of damage in the case of a disaster." [dLSA92]

Given a system which satisfies its mission and safety requirements, we might wish to prove that the implementation of mission requirements cannot have an adverse affect upon the implementation of safety requirements in that system. A property, definable in terms of protection, which we refer to as *mission-safety independence*, allows us to reason about such non-interference.

As an example, assume a set of mission requirements, M, and a set of safety requirements, S. A suitable implementation of these requirements might be a process P, such that

$$(\forall\, m : M \bullet m \sqsubseteq P) \land (\forall\, s : S \bullet s \sqsubseteq P)$$

We start by formalising what we mean by an *implementation.*

4.1.1 Property Checking with FDR

FDR verifies that a property P holds of a specification S by checking that the latter is a refinement of the former.

As an example, suppose that we have a sensor which measures the level of water in a given system. If the water level is measured as *high*, then an alarm is sounded immediately and the alarm continues to be sounded until the water level is measured as *mid* or *low*, at which point the alarm is silenced. This may be represented in CSP in terms of the following process, which corresponds to the finite state automaton of Figure 4.1.

$$\begin{aligned} SwitchAlarmOn \mathrel{\hat{=}} & \; alarm.on \rightarrow AlarmOn \\ & \Box \\ & waterLevel.low \rightarrow WaterMeasure \\ & \Box \\ & waterLevel.mid \rightarrow WaterMeasure \\ & \Box \\ & waterLevel.high \rightarrow SwitchAlarmOn \end{aligned}$$

$$\begin{aligned} AlarmOn \mathrel{\hat{=}} & \; waterLevel.low \rightarrow SwitchAlarmOff \\ & \Box \\ & waterLevel.mid \rightarrow SwitchAlarmOff \\ & \Box \\ & waterLevel.high \rightarrow AlarmOn \end{aligned}$$

$$\begin{aligned} SwitchAlarmOff \mathrel{\hat{=}} & \; alarm.off \rightarrow WaterMeasure \\ & \Box \\ & waterLevel.low \rightarrow SwitchAlarmOff \\ & \Box \\ & waterLevel.mid \rightarrow SwitchAlarmOff \\ & \Box \\ & waterLevel.high \rightarrow AlarmOn \end{aligned}$$

Note that we assume that the alarm might not be able to respond immediately to the detection of a high water level. As such, we offer the external choice between further detection and the alarm being switched on. However, in a CSP model with priorities,[1] priority might be given to communications on the channel *alarm*, which would ensure that if it was possible for the alarm to be switched on, then this would occur in preference to future observations on the channel *WaterMeasure*. We discuss this issue later in this chapter.

A functional requirement of our specification is that the alarm is correctly modelled, i.e., we cannot have two consecutive occurrences of *alarm.on* or *alarm.off*.

$$MissionReq \mathrel{\hat{=}} alarm.on \rightarrow alarm.off \rightarrow MissionReq$$

[1] See, for example, [Low93].

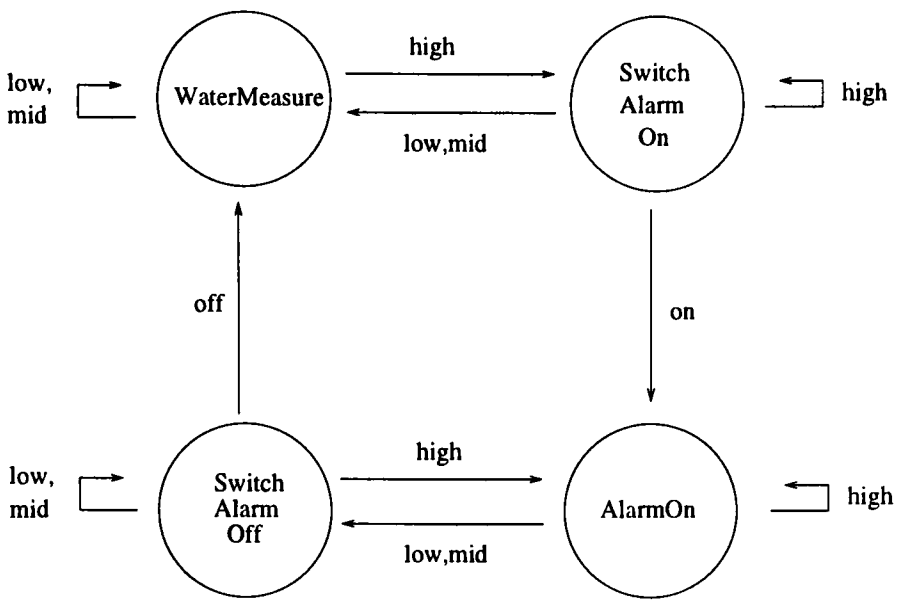

Figure 4.1: The process *WaterMeasure.*

To verify that the process *WaterMeasure* satisfies this requirement, we need to ensure that

$$MissionReq \,[\![\, \{alarm.on, alarm.off\} \,]\!]\, CHAOS_\Sigma \sqsubseteq WaterMeasure$$

This process insists that communications on the channel *alarm* alternate, while imposing no conditions upon other events from the alphabet of *WaterMeasure.* If this refinement relation holds, then we say that *WaterMeasure* implements *MissionReq.*

Definition 4.1 A process Q *implements* another process P (equivalently, P is implemented by Q), denoted Q **imp** P, if and only if

$$P \,[\![\, \alpha P \,]\!]\, CHAOS_\Sigma \sqsubseteq Q$$

□

That is, Q implements P if and only if the requirement described by P holds of the relevant communications in Q. This condition holds of *WaterMeasure* with respect to *MissionReq.*

Returning to our water measuring sensor, a safety requirement is stated thus:

> If the water level is measured as *high* and the alarm is *off*, then the alarm should be sounded.

$$\begin{aligned} SafetyReq_{off} \mathrel{\hat{=}} & \; alarm.off \rightarrow SafetyReq_{off} \\ & \Box \\ & alarm.on \rightarrow SafetyReq_{on} \\ & \Box \\ & waterLevel.high \rightarrow alarm.on \rightarrow SafetyReq_{on} \end{aligned}$$

$$\begin{aligned} SafetyReq_{on} \mathrel{\widehat{=}} & \; alarm.off \rightarrow SafetyReq_{off} \\ & \Box \\ & alarm.on \rightarrow SafetyReq_{on} \\ & \Box \\ & waterLevel.high \rightarrow SafetyReq_{on} \end{aligned}$$

Note that we assume that the alarm may be sounded independently of water measurement — the requirement does not state that the alarm is concerned exclusively with water measurement. This property fails to hold of *WaterMeasure*, as we assume that the alarm may not be capable of being sounded which means that a further occurrence of *waterLevel.high* is possible without *alarm.on* being observed. However, if priority is given to communications on the channel *alarm*, then it does hold.

Given the above requirements, together with our specification, one might attempt to prove that, in addition to implementing both *MissionReq* and $SafetyReq_{off}$, the former is non-interfering with the latter in the process *WaterMeasure*. In accordance with this, we define our property of mission-safety independence in the following way.

Definition 4.2 A process P has *mission-safety independence* with respect to mission requirements M and safety requirements S, denoted $MS(P, M, S)$, if and only if

$$(\forall\, m : M \bullet P \;\mathbf{imp}\; m \wedge \forall\, s : S \bullet P \;\mathbf{imp}\; s) \Rightarrow S - \mathcal{M} \nrightarrow \mathcal{M} - S \; [P]$$

where S and $\mathcal{M}$ are defined by

$$\begin{aligned} S &= \{e : \Sigma;\; s : S \mid e \in \alpha s \bullet e\} \\ \mathcal{M} &= \{e : \Sigma;\; m : M \mid e \in \alpha m \bullet e\} \end{aligned}$$

□

That is, mission-safety independence holds of a process P with respect to mission requirements M and safety requirements S if and only if each of the requirements being satisfied by P implies that all of those events which are *exclusively* concerned with safety are protected from those that are *exclusively* concerned with functionality in P.

In the above example, $SafetyReq_{off}$ is our safety requirement and *MissionReq* is our mission requirement. We define the sets S and $\mathcal{M}$ accordingly.

$$\begin{aligned} S &= \{waterLevel.high, alarm.on, alarm.off\} \\ \mathcal{M} &= \{alarm.on, alarm.off\} \end{aligned}$$

By Definition 4.2, mission-safety independence holds of *WaterMeasure* if and only if

$$\begin{aligned} &(WaterMeasure \;\mathbf{imp}\; SafetyReq_{off} \wedge WaterMeasure \;\mathbf{imp}\; MissionReq) \Rightarrow \\ &\quad \varnothing \nrightarrow \{waterLevel.high\}\; [WaterMeasure] \end{aligned}$$

As *WaterMeasure* fails to implement its safety requirement, this holds trivially.

4.1.2 A New Form of Protection

In our above definition of mission-safety independence, we considered non-disjoint sets of events for the first time. Rather than subtracting elements from sets in the manner described above, we define a new property, denoted $s \nleftarrow\!\!\!\!\!\!+\!\!\!\!+ s'$, which describes protection between sets of events which are not disjoint.

Definition 4.3 $\forall P : PROC;\ s_1, s_2 : \mathbb{P}\Sigma \bullet s_1 \nleftarrow\!\!\!\!\!\!+\!\!\!\!+ s_2\ [P] \Leftrightarrow s_1 - s_2 \nleftarrow\!\!\!+ s_2 - s_1\ [P]$ □

The following proposition holds trivially.

Proposition 4.1 $\forall P : PROC;\ s_1, s_2 : \mathbb{P}\Sigma \bullet s_1 \nleftarrow\!\!\!+ s_2\ [P] \Rightarrow s_1 \nleftarrow\!\!\!\!\!\!+\!\!\!\!+ s_2\ [P]$

Proof. Assume $s_1 \nleftarrow\!\!\!+ s_2\ [P]$. By Definition 2.6,

$$\begin{aligned}&\exists s : \mathbb{P}\Sigma \bullet \\ &\quad \alpha P \subseteq (s \cup s_1) \cup ((\alpha P - (s \cup s_1)) \cup s_2) \wedge \\ &\quad (s \cup s_1) \cap ((\alpha P - (s \cup s_1)) \cup s_2) = \varnothing \wedge \\ &\quad \mathbf{det}(P \,|||\, RUN_{(\alpha P - (s \cup s_1)) \cup s_2})\end{aligned}$$

As $(s \cup s_1) \cap ((\alpha P - (s \cup s_1)) \cup s_2) = \varnothing$, s_1 and s_2 must be disjoint, i.e.,

$$s_1 - s_2 = s_1 \wedge s_2 - s_1 = s_2$$

Therefore, by substitution, $s_1 - s_2 \nleftarrow\!\!\!+ s_2 - s_1\ [P]$ and by Definition 4.3, $s_1 \nleftarrow\!\!\!\!\!\!+\!\!\!\!+ s_2\ [P]$. □

We take the following simple system as a further example of mission-safety independence.

In a warehouse, there is an automated vehicle which delivers products to a loading bay. One of the system's safety requirements is that if an obstacle is detected in front of the vehicle, then the vehicle's brakes should be applied immediately. If we let the events *detect* and *brake* represent the detection of an object and the subsequent immediate application of brakes respectively, then we state this requirement in the following way.

$$SafetyReq \mathrel{\widehat{=}} detect \rightarrow brake \rightarrow SafetyReq \,\Box\, brake \rightarrow SafetyReq$$

Note that we assume that occurrences of *brake* may occur independently of those of *detect*.

Our vehicle also has a flashing light which is under computer control. One of the system's mission requirements is that this light should flash every second. We represent the passing of a second and a flash of the light by the events *tock* and *flash* respectively.

$$MissionReq \mathrel{\widehat{=}} tock \rightarrow flash \rightarrow MissionReq$$

It follows that any process P which implements *SafetyReq* ensures that an occurrence of *detect* is followed by one of *brake*. In addition, if P implements *MissionReq*, then occurrences of *tock* and *flash* will alternate.

As these are our only stated requirements, it follows that any process implements these requirements should be acceptable to us as a specification. Consider the process *Vehicle*.

$$Vehicle \mathrel{\widehat{=}} tock \rightarrow flash \rightarrow Vehicle'$$
$$Vehicle' \mathrel{\widehat{=}} detect \rightarrow brake \rightarrow Vehicle \,\Box\, brake \rightarrow Vehicle$$

This process satisfies both of our requirements. However, it is clearly a nonsensical implementation, as the vehicle brakes every second. Despite the fact that it implements our safety and mission requirements, it is clearly an unimplementable specification, as the latter has an adverse affect upon the former.

If we were to add the requirement that mission-safety interference should hold of *Vehicle* with respect to $\alpha MissionReq$ and $\alpha SafetyReq$, then we would find that

$$\{detect, brake\} \leftarrow \{tock, flash\}\ [Vehicle]$$

As such, mission-safety independence fails to hold of this process. On the other hand, there is no such information flow in the process *NewVehicle*.

$$NewVehicle \mathrel{\widehat{=}} SafetyReq \,|||\, MissionReq$$

Here, the alphabets of *SafetyReq* and *MissionReq* are disjoint, and observations of *detect* and *brake* will always occur independently of those of *tock* and *flash*. As such,

$$\{detect, brake\} \nleftarrow \{tock, flash\}\ [NewVehicle]$$

Thus, independence between events concerned with safety and those concerned with functionality can be expressed in terms of protection. There is one issue which we have left to address — that of priority. We return to this issue in the next section, in which we illustrate how we might give priority to critical events.

4.2 Dealing with Failures

In Chapter 1, we introduced a number of different failure modes which might be associated with a safety-critical system: such a system may fail-safe, fail-soft, or fail-operational. In Chapter 2, we proposed a general model for safety-critical systems, which included a classification of such a system's observable events. In this section, we describe how our failure modes can be expressed in terms of non-interference properties which are based on these classifications.

Our first task is to define functions which assign classifications to a process's events.

Definition 4.4 The following functions categorise the events of a process according to the classifications of Chapter 2.

$$benign : PROC \rightarrow \mathbb{P}\,\Sigma$$
$$fails : PROC \rightarrow \mathbb{P}\,\Sigma$$
$$hazards : PROC \rightarrow \mathbb{P}\,\Sigma$$
$$recovery : PROC \rightarrow \mathbb{P}\,\Sigma$$
$$safety : PROC \rightarrow \mathbb{P}\,\Sigma$$

Here,

$$\forall P : PROC \bullet benign(P) \subseteq \alpha P \wedge fails(P) \subseteq \alpha P \wedge hazards(P) \subseteq \alpha P \wedge recovery(P) \subseteq \alpha P \wedge safety(P) \subseteq \alpha P$$

□

Note that we do not assume that these functions partition the alphabet of processes.

4.2.1 Fail-safe

It is a basic requirement of any system in which incorrect behaviour might result in a threat to safety that the occurrence of failures must have no affect upon safety-critical events. Nevertheless, in many cases it may be permissible for benign activity to be affected, or even terminated, by these failures. The overall requirement is that if the loss or corruption of a particular activity would have a detrimental affect on the level of safety of the system and its environment, then that activity should be protected from the consequences of failure at all costs. If this requirement is met, then we say that the system *fails-safe*.

For example, a safety requirement of a fire protection system is that, at all times, it should be capable of detecting fires and sounding a fire alarm in the area in which a fire is detected. It is often the case that faults, such as a display lamp being incapable of illumination, will occur. Despite this, the system's safety requirements should still be met.

In the context of non-interference, we shall say that a system fails-safe if and only if its safety-critical events are protected from its failures.

Definition 4.5 A process P *fails-safe* with respect to a set of failures F, denoted $FailSafe(P, F)$, if and only if

$$safety(P) \dashv\!\!+ F\ [P]$$

□

Recall the automated vehicle of the previous section. The two requirements of the system were that it should flash its light every second and it should brake every time an object is detected in its path. Suppose that two further events to which it should react are those of the vehicle travelling either off course or too fast, which we represent by the events *offCourse* and *tooFast* respectively. Following an occurrence of the former, the vehicle's course should be reset. This is represented by the event *reset*. The event *onCourse* represents the successful recovery from this failure. In the latter case, the vehicle's brakes should be applied immediately. Here, the event *slowEnough* represents successful recovery. Finally, if a vehicle travels both too fast and off course, then it will crash. This is a hazard, which is represented by the ghost event $\underline{crash}$. We categorise *tooFast* and *offCourse* as failures, with *brake* and *reset* being the corresponding recovery measures for these failures.

Given these classifications and the above information, we define a hazard analysis for this system by the process *HA*. This hazard analysis corresponds to the fault tree analysis

of Figure 1.2 and event tree analyses of Figure 1.3 and Figure 1.4.

$$
\begin{aligned}
HA \mathrel{\hat{=}} \; & tooFast \rightarrow TooFastFailure \\
& \Box \\
& slowEnough \rightarrow HA \\
& \Box \\
& offCourse \rightarrow OffCourseFailure \\
& \Box \\
& onCourse \rightarrow HA
\end{aligned}
$$

If the vehicle is travelling on course and not too fast, then there is no change in state. Alternatively, if either *tooFast* or *offCourse* are observed, then the next state is described by *TooFastFailure* or *OffCourseFailure* respectively.

$$
\begin{aligned}
TooFastFailure \mathrel{\hat{=}} \; & tooFast \rightarrow TooFastFailure \\
& \Box \\
& slowEnough \rightarrow HA \\
& \Box \\
& offCourse \rightarrow \underline{crash} \rightarrow \bot_{\{tooFast,slowEnough,offCourse,onCourse,\underline{crash}\}} \\
& \Box \\
& onCourse \rightarrow TooFastFailure
\end{aligned}
$$

$$
\begin{aligned}
OffCourseFailure \mathrel{\hat{=}} \; & tooFast \rightarrow \underline{crash} \rightarrow \bot_{\{tooFast,slowEnough,offCourse,onCourse,\underline{crash}\}} \\
& \Box \\
& slowEnough \rightarrow OffCourseFailure \\
& \Box \\
& offCourse \rightarrow OffCourseFailure \\
& \Box \\
& onCourse \rightarrow HA
\end{aligned}
$$

If the vehicle travels off course when in state *TooFastFailure*, then it will crash. There is a similar result if it travels too fast when in state *OffCourseFailure*.

We define our new specification in terms of the vehicle's components. First, the system has two sensors which analyse speed and direction. Here, the events *tooFast* and *slowEnough* can be observed at all times.

$$Speed \mathrel{\hat{=}} tooFast \rightarrow Speed \;\Box\; slowEnough \rightarrow Speed$$

The process *Direction* is defined similarly.

$$Direction \mathrel{\hat{=}} offCourse \rightarrow Direction \;\Box\; onCourse \rightarrow Direction$$

Next, we define a process representing the vehicle's light, which flashes every second.

$$Light \mathrel{\hat{=}} tock \rightarrow flash \rightarrow Light$$

The vehicle's brakes are applied following the detection of an object in its path, or if it is travelling too fast. We define the process *Brakes* accordingly.

$$Brakes \mathrel{\widehat{=}} tooFast \rightarrow Brakes' \;\Box\; detect \rightarrow Brakes'$$
$$Brakes' \mathrel{\widehat{=}} tooFast \rightarrow Brakes' \;\Box\; detect \rightarrow Brakes' \;\Box\; brake \rightarrow Brakes$$

Finally, if the vehicle is travelling off course, then its course is reset.

$$Course \mathrel{\widehat{=}} offCourse \rightarrow Course'$$
$$Course' \mathrel{\widehat{=}} offCourse \rightarrow Course' \;\Box\; reset \rightarrow Course$$

We define our view of the vehicle's overall behaviour by the process *Vehicle*2.

$$Vehicle2 \mathrel{\widehat{=}} (Light \;|||\; Brakes \;|||\; Course) \;[\![\; \{offCourse, tooFast\} \;]\!]\; (Speed \;|||\; Direction)$$

We classify the events of this process in the following way.

$$safety(Vehicle2) = \{detect\}$$
$$benign(Vehicle2) = \{flash, slowEnough, onCourse\}$$
$$fails(Vehicle2) = \{tooFast, offCourse\}$$
$$recovery(Vehicle2) = \{brake, reset\}$$

This system fails-safe with respect to *fails*(*Vehicle*2) if and only if its safety-critical events are protected from its failures, i.e.,

$$safety(Vehicle2) \mathrel{+\!\!\!\vdash} fails(Vehicle2)\ [Vehicle2]$$

This is indeed true.

In addition, we might ensure that our specification cannot lead to a hazard by defining

$$System \mathrel{\widehat{=}} Vehicle2 \;[\![\; \{tooFast, slowEnough, offCourse, onCourse\} \;]\!]\; HA$$

If **nondiv**(*System*), then the process *Vehicle*2 can never lead to *crash*. This property fails to hold of this process. However, as we shall see later, if we redefine this process to give priority to recovery events, then the hazard can be avoided.

4.2.2 Fail-soft

In defining our notion of a process failing safe, we made no mention of whether the actions taken to recover from failures could affect safety-critical events. If we require that recovery measures cannot affect safety-critical events, then we must impose a stronger condition upon processes, requiring that neither failures, nor the measures taken by the system to recover from them, can influence safety-critical activity in any way.

Definition 4.6 A process P *fails-soft* with respect to the set of failures F and the set of recovery events R, denoted $FailSoft(P, F, R)$, if and only if

$$safety(P) \mathrel{+\!\!\!\vdash} F \cup R\ [P]$$

□

Recalling our automated vehicle specification, we find that

$$safety(Vehicle2) \dashv\!\!+ fails(Vehicle2) \cup recovery(Vehicle2)\ [Vehicle2]$$

As such, *Vehicle*2 fails-soft with respect to its failures and recovery events.

4.2.3 Fail-operational

Our definitions of fail-safe and fail-soft both require that safety-critical events are protected from failures (and, in the case of our definition of fail-soft, are also protected from recovery procedures). We have made no mention of possible protection for the system's non-safety-critical activity. There has been an implicit acceptance that failures may interfere with such events, with the possible result of the loss of all non-safety critical activity.

If we require that a process P's benign events, as well as its safety-critical events are unaffected by the occurrence of, and subsequent recovery from, P's failures, then we need to establish

$$safety(P) \cup benign(P) \dashv\!\!+ fails(P) \cup recovery(P)\ [P]$$

If this condition holds of a process, then we say that it *fails-operational.*

Definition 4.7 A process P *fails-operational* with respect to a set of failures F and a set of recovery events R, denoted $FailOp(P, F, R)$, if and only if

$$safety(P) \cup benign(P) \dashv\!\!+ F \cup R\ [P]$$

□

If we denote *fails*(*Vehicle*2) by F and *recovery*(*Vehicle*2) by R, as

$$\alpha Vehicle2 \subseteq (\alpha Vehicle2 - (F \cup R)) \cup (F \cup R) \wedge$$
$$(\alpha Vehicle2 - (F \cup R)) \cap (F \cup R) = \varnothing \wedge$$
$$\mathbf{det}(Vehicle2 \mid\mid\mid RUN_{F \cup R})$$

by Definition 2.5,

$$\alpha Vehicle2 - (F \cup R) \nVdash F \cup R\ [Vehicle2]$$

By Proposition 2.4,

$$\alpha Vehicle2 - (F \cup R) \dashv\!\!+ F \cup R\ [Vehicle2]$$

and by Law A.1,

$$safety(Vehicle2) \cup benign(Vehicle2) \dashv\!\!+ F \cup R\ [Vehicle2]$$

Finally, by Definition 4.7, our motor vehicle specification fails-operational with respect to its failures and recovery events.

As a further example, assume a signal, represented by the process *SignalInit*, which communicates its colour (red or green) to a receiving entity upon request (a safety-critical event).

$$Colour ::= red \mid green$$

The system also emits an audible 'beep' every second (a benign event).

This system is known to be prone to a failure. Upon observation of this failure, the signal immediately switches to a backup system, which is capable of transmitting the colour, but is incapable of emitting the beep.

We start by introducing the following channels.

- *initSignal.c* — *c* is the signal's initial colour.
- *signalColour.c* — *c* is the value transmitted as the signal's colour.
- *signalChange.c* — the signal's colour changes to *c*.
- *backup.c* — *c* is the backup signal's initial colour.

We categorise the events and channels in which this system participates in the following way.

$$\begin{aligned} safety(SignalInit) &= events(signalColour) \cup events(signalChange) \\ benign(SignalInit) &= \{beep\} \cup events(initSignal) \\ fails(SignalInit) &= \{signalFailure\} \\ recovery(SignalInit) &= events(backup) \end{aligned}$$

The behaviour of this system is described by the process *SignalInit*.

$$\begin{aligned} SignalInit \mathrel{\widehat{=}} & \; initSignal?newColour \rightarrow \\ & (Signal(newColour) \triangle (signalFailure \rightarrow Backup)) \end{aligned}$$

After a failure, the signal's colour is received by the backup system.

$$Backup \mathrel{\widehat{=}} backup?newColour \rightarrow BackupSignal(newColour)$$

The 'normal' system behaviour is described by the process *Signal*.

$$\begin{aligned} Signal(c) \mathrel{\widehat{=}} & \; tock \rightarrow Tocked(c) \\ & \Box \\ & signalColour!c \rightarrow Signal(c) \\ & \Box \\ & signalChange?newColour \rightarrow Signal(newColour) \end{aligned}$$

$$\begin{aligned} Tocked(c) \mathrel{\widehat{=}} & \; beep \rightarrow Signal(c) \\ & \Box \\ & signalColour!c \rightarrow Tocked(c) \\ & \Box \\ & signalChange?newColour \rightarrow Tocked(newColour) \end{aligned}$$

In addition, the behaviour of the backup system is described by the process *BackupSignal*.

$$\begin{aligned} BackupSignal(c) \mathrel{\widehat{=}} & \; tock \rightarrow BackupSignal(c) \\ & \Box \\ & signalColour!c \rightarrow BackupSignal(c) \\ & \Box \\ & signalChange?newColour \rightarrow BackupSignal(newColour) \end{aligned}$$

We also define a safety controller, which observes changes in signal aspect and communicates this aspect to the backup system when this value is requested.

$$SafetyControllerInit \mathrel{\widehat{=}} initSignal?newColour \rightarrow SafetyController(newColour)$$

$$\begin{aligned} SafetyController(c) \mathrel{\widehat{=}} & \; signalChange?newColour \rightarrow SafetyController(newColour) \\ & \Box \\ & backup!c \rightarrow SafetyController(c) \end{aligned}$$

The overall system behaviour is described by the process *System*:

$$System \mathrel{\widehat{=}} SignalInit \; [\![\{initSignal.*, signalChange.*, backup.*\}]\!] \; SafetyControllerInit$$

By Definition 4.7, this process fails-operational with respect to its failures and recovery events if and only if

$$safety(SignalInit) \cup benign(SignalInit) \; \leftarrow\!\!\!+ \; fails(SignalInit) \cup recovery(SignalInit) \; [System]$$

Unfortunately, this property fails to hold, as the opportunity to perform *beep* (a benign event) is removed from the system following a failure. Neither does this process fail-safe or fail-soft because of the need to observe a communication on the channel *backup* following an observation of *signalFailure*. However, if we conceal the behaviour of the recovery events, i.e., communications on the channel *backup*, then this process both fails-soft (trivially, as *recovery*(*SignalInit*) is no longer in the alphabet of the process) and fails-safe, i.e.,

$$safety(SignalInit) \; \leftarrow\!\!\!+ \; fails(SignalInit) \; [System \setminus recovery(SignalInit)]$$

We discuss this technique of hiding recovery events later in this section.

4.2.4 Some Implications

Given our above definitions, if a system fails-operational, then it fails-soft. Also, if a system fails-soft, then it fails-safe.

Proposition 4.2 $\forall P : PROC;\ F, R : \mathbb{P}\,\Sigma \bullet FailOp(P, F, R) \Rightarrow FailSoft(P, F, R)$

Proof.

$FailOp(P, F, R)$	[Assumption]	(1)
$safety(P) \cup benign(P) \nrightarrow F \cup R\ [P]$	[1, Definition 4.7]	(2)
$safety(P) \nrightarrow F \cup R\ [P]$	[2, Law A.1]	(3)
$FailSoft(P, F, R)$	[3, Definition 4.6]	(4)

□

Proposition 4.3 $\forall P : PROC;\ F, R : \mathbb{P}\,\Sigma \bullet FailSoft(P, F, R) \Rightarrow FailSafe(P, F)$

Proof.

$FailSoft(P, F, R)$	[Assumption]	(1)
$safety(P) \nrightarrow F \cup R\ [P]$	[1, Definition 4.6]	(2)
$safety(P) \nrightarrow F\ [P]$	[2, Law A.2]	(3)
$FailSafe(P, F)$	[3, Definition 4.5]	(4)

□

4.2.5 Further Approaches

Of course, there are many other possible failure modes possible for a safety-critical system. For example, following a failure, a system *Sys* may stop all activity (assuming that it terminates in a safe state), i.e., it might be defined in the form

$$SafeSys_1 \mathrel{\widehat{=}} Sys \triangle (failure \rightarrow Stop)$$

We say that such a system is a *fail-stop* (q.v.) system.

Furthermore, assuming that its initial state is safe, the system may return to its initial state following a failure, i.e.,

$$SafeSys_2 \mathrel{\widehat{=}} Sys \triangle (failure \rightarrow SafeSys_2)$$

We say that such a system is a *fail-initial* (q.v.) system.

We think of these approaches as being modes of operation, and our interpretations of fail-safe, fail-soft and fail-operational as being properties of systems. For example, the fail-stop system $SafeSys_1$ and the fail-initial system $SafeSys_2$ may fail-safe, fail-soft, or fail-operational, depending on the non-interference properties of $SafeSys_1$ and $SafeSys_2$.

4.2.6 Benign and Catastrophic Failures

There are a number of ways of classifying failures. For example, [Lap89] suggests dividing them into two categories: *benign failures* (q.v.) and *catastrophic failures* (q.v.). In the following, we shall divide sets of failures according to their consequences. For example, if the occurrence of a failure affects safety-critical behaviour, then we shall say that it is catastrophic; if it cannot interfere with critical events, but can disrupt the delivery of proper

service, then we shall say that it is benign; finally, if neither of the above cases holds, i.e., neither safety-critical nor benign activity can be affected by a failure, then we shall say that it is invisible.

We relate these properties to our definitions of fail-safe, fail-soft and fail-operational in the following way.

Definition 4.8 Given a process $P \in PROC$ and a failure $f \in \Sigma$, we say that f is

- *catastrophic* in P, denoted $Catastrophic(P, f)$, if and only if

$$\neg FailSafe(P, \{f\})$$

- *benign* in P, denoted $Benign(P, f)$, if and only if

$$FailSafe(P, \{f\}) \wedge \neg FailOp(P, \{f\}, recovery(P))$$

- *invisible* in P, denoted $Invisible(P, f)$, if and only if

$$FailOp(P, \{f\}, recovery(P))$$

□

The above definition ensures that a failure has exactly one classification in a process P.

Proposition 4.4 Given a process $P \in PROC$ and a failure $f \in \Sigma$, exactly one of the following conditions holds.

- $Catastrophic(P, f)$,
- $Benign(P, f)$, or
- $Invisible(P, f)$.

Proof. By Definition 4.8, we have the following.

$$\begin{aligned} Catastrophic(P, f) &\Leftrightarrow \neg FailSafe(P, \{f\}) \\ Benign(P, f) &\Leftrightarrow FailSafe(P, \{f\}) \wedge \neg FailOp(P, \{f\}, recovery(P)) \\ Invisible(P, f) &\Leftrightarrow FailOp(P, \{f\}, recovery(P)) \end{aligned}$$

By Definitions 4.5, 4.6 and 4.7, the above is equivalent to

$$\begin{aligned} Catastrophic(P, f) &\Leftrightarrow safety(P) \longleftarrow \{f\}\ [P] \\ Benign(P, f) &\Leftrightarrow safety(P) \dashv\!\!\!\!+ \{f\}\ [P] \\ &\quad \wedge \\ &\quad safety(P) \cup benign(P) \longleftarrow \{f\} \cup recovery(P)\ [P] \\ Invisible(P, f) &\Leftrightarrow safety(P) \cup benign(P) \dashv\!\!\!\!+ \{f\} \cup recovery(P)\ [P] \end{aligned}$$

From our understanding of protection, if f is invisible in P, then it cannot be benign or catastrophic. Similarly, if f is benign in P, then it cannot be catastrophic. Finally, if f is neither invisible nor benign in P, then it must be catastrophic. □

4.2.7 The Treatment of Recovery

During this section, we have mentioned two means of representing recovery — one using models of CSP with priorities, and the other by abstraction.

We deal with the priorities approach first. If we were to extend our approach to such a model, then recovery (if available) would occur in preference to other types of event, thus being representative of the notion that if recovery is available, then it must occur.

Such a model of CSP is described in [Low93], with the fundamental extension to the failures-divergences model being that of prioritised choice. As well as the 'normal' external choice operator, $\Box$, this model has an external choice operator with left priority, which we write as $\underset{\leftarrow}{\Box}$, and an external choice operator with right priority, which we write as $\underset{\rightarrow}{\Box}$.

As an example, we take processes P_{11} and P_{12}, which are defined as follows.

$$P_{11} \mathrel{\hat{=}} AlanRead \rightarrow Stop \underset{\leftarrow}{\Box} BillRead \rightarrow Stop$$
$$P_{12} \mathrel{\hat{=}} AlanRead \rightarrow Stop \underset{\rightarrow}{\Box} BillRead \rightarrow Stop$$

Assuming that these processes have not been placed in parallel with other processes which may restrict their behaviour, P_{11} will choose to perform *AlanRead* in preference to *BillRead* before terminating, while P_{12} will give preference to *BillRead*.

As an example of the use of prioritised choice in representing recovery, recall our motor vehicle specification, which was defined by

$$Vehicle2 \mathrel{\hat{=}} (Light \;|||\; Brakes \;|||\; Course) \;[\![\; \{offCourse, tooFast\} \;]\!]\; (Speed \;|||\; Direction)$$

Recall that we also defined

$$System \mathrel{\hat{=}} Vehicle2 \;[\![\; \{tooFast, slowEnough, offCourse, onCourse\} \;]\!]\; HA$$

and that, although *Vehicle2* fails-safe, we could not guarantee the absence of hazards.

If we were to rewrite the processes *Brakes* and *Course*, so that priority is given to recovery events, we find that not only does the process *Vehicle2* avoid its hazard, but it still fails-safe.

We redefine the processes *Brakes* and *Course* thus:

$$Brakes \mathrel{\hat{=}} tooFast \rightarrow Brakes' \;\Box\; detect \rightarrow Brakes'$$
$$Brakes' \mathrel{\hat{=}} (tooFast \rightarrow Brakes' \;\Box\; detect \rightarrow Brakes') \underset{\rightarrow}{\Box} brake \rightarrow Brakes$$
$$Course \mathrel{\hat{=}} offCourse \rightarrow Course'$$
$$Course \mathrel{\hat{=}} offCourse \rightarrow Course \underset{\rightarrow}{\Box} reset \rightarrow Course$$

As a further example of the use of prioritised choice, recall how our water measure failed to meet its safety requirement, $SafetyReq_{off}$. By redefining *WaterMeasure* so that priority is given to communications on the channel *alarm*, this process not only satisfies its safety requirement, but also guarantees mission-safety independence.

As well as giving priority to safety-critical events, the other non-interference caveat that we have encountered in this section is that of abstracting recovery events by concealment.

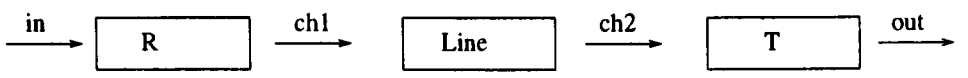

Figure 4.2: A fault-tolerant buffer.

This acknowledges the fact that there may be occasions when recovery events are the only type of event which may be observed, and that we verify that this recovery has not interfered with the rest of the system after it has been concluded, rather than during its execution. In the next section, we model fault-tolerance along similar lines.

4.3 Fault-tolerance

Suppose that we are aware that a given system is prone to a particular set of *faults* (q.v.), and that we have prescribed means of preventing such faults from occurring, e.g., by redundancy. We might wish that both safety-critical and benign components of this system are protected from both the set of faults and the measures which we take to avoid them, i.e., that the system is *fault-tolerant* (q.v.).

We define fault-tolerance in terms of protection in the following way.

Definition 4.9 Assume a process P, with a faulty communication ch, the behaviour of which is described by the process F. Assume further the set R_{ch}, which is the set of communications which aid in the prevention of, and recovery from the faults associated with ch. We shall say that P is *fault-tolerant* with respect to ch, denoted $FT(P, ch)$, if and only if

$$benign(P) \cup safety(P) \nleftarrow (\{ch\} \cup R_{ch})\ [(P \parallel F) \setminus \{ch\} \cup R_{ch}]$$

□

Note that this definition involves concealment, as well as interleaving in its guise of protection. As such, this property is similar to eager independence in its treatment of fault-tolerance.

In addition, we define the following function.

Definition 4.10 The function

$$faults : PROC \rightarrow \mathbb{P}\,\Sigma$$

such that

$$\forall\, P : PROC \bullet faults(P) \subseteq \alpha P$$

defines the set of faults associated with a particular process. □

4.3.1 Fault-tolerance by Redundancy

Assume a faulty buffer which transmits bits. The buffer is known to corrupt a certain percentage of bits, although it is guaranteed that no more than one in every three consecutive transmissions will ever be corrupted.

The behaviour of the system is illustrated by Figure 4.2. Initially, a bit is received on the channel *in*. This bit is input to the buffer from the receiver over channel ch_1, and then output from the buffer to the transmitter over channel ch_2. Finally, the bit is output on channel *out*.

We start by introducing the set *Data*

$$Data = \{0, 1\}$$

together with the channels *in*, ch_1, ch_2 and *out*, such that

$$\begin{aligned} events(in) &= \{d : Data \bullet in.d\} \\ events(ch_1) &= \{d : Data \bullet ch_1.d\} \\ events(ch_2) &= \{d : Data \bullet ch_2.d\} \\ events(out) &= \{d : Data \bullet out.d\} \end{aligned}$$

The faulty behaviour of channel ch_2 is modelled by the process $Fault_1$.

$$\begin{aligned} Fault_1 \mathrel{\widehat{=}} ch_1?d_1 \rightarrow\ & ch_2!d_1 \rightarrow Fault_1 \\ & \sqcap \\ & ch_2!\neg d_1 \rightarrow ch_1?d_2 \rightarrow ch_2!d_2 \rightarrow ch_1?d_3 \rightarrow ch_2!d_3 \rightarrow Fault_1 \end{aligned}$$

Either the bit is transmitted successfully over channel ch_2, or the negated value is transmitted. If the latter case is true, then we know that the next two transmissions will be transmitted successfully.

Following an initial receipt of data, the received bit is transmitted three times over channel ch_1 to combat the fault.

$$R_1 \mathrel{\widehat{=}} in?d \rightarrow ch_1!d \rightarrow ch_1!d \rightarrow ch_1!d \rightarrow R_1$$

As channel ch_1 is used as a means of overcoming the faulty behaviour of ch_2, we define

$$R_{ch_2} = events(ch_1)$$

When the process $Line_1$ receives data on channel ch_1, it is subsequently transmitted over channel ch_2. Remember that if a fault occurs, then the bit transmitted will be the negation of d. Thus, a correct modelling of this communication allows either the correct or the corrupted value of d to be transmitted. The choice will be resolved by the process $Fault_1$.

$$Line_1 \mathrel{\widehat{=}} ch_1?d \rightarrow (ch_2!d \rightarrow Line_1 \ \Box\ ch_2!\neg d \rightarrow Line_1)$$

Following the receipt of three bits over channel ch_2, the majority of the three values is output over channel *out*.

$$T_1 \mathrel{\widehat{=}} ch_2?d_1 \rightarrow ch_2?d_2 \rightarrow ch_2?d_3 \rightarrow out!(maj\{d_1, d_2, d_3\}) \rightarrow T_1$$

We define our overall specification by $Comm_1$.

$$Comm_1 \mathrel{\widehat{=}} (R_1 \ [\![\ events(ch_1)\ ]\!]\ Line_1) \ [\![\ events(ch_2)\ ]\!]\ T_1$$

The system's overall behaviour with respect to its fault is described by $FaultyComm_1$.

$$FaultyComm_1 \mathrel{\widehat{=}} Comm_1 \llbracket events(ch_1) \cup events(ch_2) \rrbracket Fault_1$$

Finally, we conceal the events which are internal to the buffer, and define the process Sys_1.

$$Sys_1 \mathrel{\widehat{=}} FaultyComm_1 \setminus (events(ch_1) \cup events(ch_2))$$

If we denote $events(ch_1)$ by ch_1, $events(ch_2)$ by ch_2, etc., then by Definition 4.9, this buffer is fault-tolerant with respect to ch_2 if and only if

$$in \cup out \mathrel{+\!\!\!-} ch_1 \cup ch_2 \; [Sys_1]$$

Proposition 4.5 $in \cup out \mathrel{+\!\!\!-} ch_1 \cup ch_2 \; [Sys_1]$

Proof. By observation,

$$\alpha Sys_1 \subseteq (in \cup out) \cup (ch_1 \cup ch_2) \wedge (in \cup out) \cap (ch_1 \cup ch_2) = \varnothing$$

FDR confirms that Sys_1 is deterministic. As

$$(ch_1 \cup ch_2) \cap \alpha Sys_1 = \varnothing$$

by Proposition 3.2, $\mathbf{det}(Sys_1 \;|||\; RUN_{ch_1 \cup ch_2})$. By Definition 2.5,

$$in \cup out \not\mathrel{+\!\!\!-} ch_1 \cup ch_2 \; [Sys_1]$$

Finally, by Proposition 2.4,

$$in \cup out \mathrel{+\!\!\!-} ch_1 \cup ch_2 \; [Sys_1]$$

□

4.3.2 Fault-tolerance by Retransmission

The above example is an illustration of the use of redundancy in providing fault-tolerance. The following is an example of fault-tolerance by retransmission.

Our system is faulty in that messages can become scrambled, although it is guaranteed that two consecutive messages will never be scrambled. We introduce the set S_Data accordingly.

$$S\_Data = \{0, 1, scrambled\}$$

We also redefine $events(ch_2)$.

$$events(ch_2) = \{d : S\_Data \bullet ch_2.d\}$$

We model this fault in terms of the following process. If, following the receipt of data on ch_1, the transmission on ch_2 is scrambled, then the retransmission will be successful. Alternatively, the initial transmission might be successful.

$$Fault_2 \mathrel{\hat{=}} ch_1?d_1 \rightarrow (ch_2!d_1 \rightarrow Fault_2 \sqcap ch_2!scrambled \rightarrow Fault_2'(d_1))$$

$$\begin{aligned} Fault_2'(d_1) \mathrel{\hat{=}}\ & ch_2!d_1 \rightarrow Fault_2 \\ & \Box \\ & ch_1?d_2 \rightarrow ch_2!d_1 \rightarrow ch_2!d_2 \rightarrow Fault_2 \\ & \qquad\qquad\qquad\qquad \sqcap \\ & \qquad\qquad\qquad\qquad ch_2!scrambled \rightarrow Fault_2'(d_2) \end{aligned}$$

Note that we assume that the need for retransmission will not disrupt the ordering of messages.

We redefine the processes representing the buffer's behaviour according to the necessity of overcoming the faulty behaviour described above.

Following the receipt of data on channel *in*, the relevant value is transmitted over channel ch_1.

$$R_2 \mathrel{\hat{=}} in?d \rightarrow ch_1!d \rightarrow R_2$$

After receiving data on channel ch_1, either the actual value of d or a scrambled version is transmitted over ch_2.

$$Line_2 \mathrel{\hat{=}} ch_1?d \rightarrow (ch_2!d \rightarrow Line_2'(d) \Box ch_2!scrambled \rightarrow Line_2'(d))$$

In a similar fashion to the first example, the choice will be resolved by $Fault_2$.

If confirmation that the value was received correctly is forthcoming (given by *ok*), then the process can again receive on channel ch_1. On the other hand, if the value was scrambled, then the receiver requests a retransmission (represented by the event *retransmit*).

$$\begin{aligned} Line_2'(d) \mathrel{\hat{=}}\ & ok \rightarrow Line_2 \\ & \Box \\ & retransmit \rightarrow (ch_2!d \rightarrow Line_2'(d) \Box ch_2!scrambled \rightarrow Line_2'(d)) \end{aligned}$$

The actions of the process T_2 after receiving a bit depend upon whether it is scrambled or not. If the value is scrambled, then it requests a retransmission; alternatively, confirmation of a successful receipt is sent.

$$\begin{aligned} T_2 \mathrel{\hat{=}} ch_2?d \rightarrow\ & \textbf{if} \quad d = scrambled \\ & \textbf{then}\ retransmit \rightarrow T_2 \\ & \textbf{else}\ ok \rightarrow out!d \rightarrow T_2 \end{aligned}$$

We define $R_{ch_2} = \{ok, retransmit\}$ and compose our system by

$$Comm_2 \mathrel{\hat{=}} (T_2 \,[\![\, ch_2 \cup \{ok, retransmit\} \,]\!]\, Line_2) \,[\![\, ch_1 \,]\!]\, R_2$$

The system's behaviour with respect to the above fault is described by

$$FaultyComm_2 \mathrel{\widehat{=}} Comm_2 \,[\![\, ch_1 \cup ch_2 \,]\!]\, Fault_2$$

Finally, we define

$$Sys_2 \mathrel{\widehat{=}} FaultyComm_2 \setminus (ch_1 \cup ch_2 \cup \{ok, retransmit\})$$

Proposition 4.6 $ch_1 \cup in \cup out \dashv\!\!+ ch_2 \cup \{ok, retransmit\}\ [Sys_2]$

Proof. By observation,

$$\alpha Sys_2 \subseteq (ch_1 \cup in \cup out) \cup (ch_2 \cup \{ok, retransmit\}) \wedge$$
$$(ch_1 \cup in \cup out) \cap (ch_2 \cup \{ok, retransmit\}) = \varnothing$$

FDR confirms that Sys_2 is deterministic. As

$$\alpha Sys_2 \cap (ch_2 \cup \{ok, retransmit\}) = \varnothing$$

by Proposition 3.2, $\mathbf{det}(Sys_2 \,|||\, RUN_{ch_2 \cup \{ok, retransmit\}})$. By Definition 2.5,

$$ch_1 \cup in \cup out \not\dashv\!\!+ ch_2 \cup \{ok, retransmit\}\ [Sys_2]$$

By Proposition 2.4,

$$ch_1 \cup in \cup out \dashv\!\!+ ch_2 \cup \{ok, retransmit\}\ [Sys_2]$$

□

4.3.3 Fault-tolerance by Timeout

As a final example, we examine the possibility of data becoming 'lost'. Returning to our buffer, we require that if more than 1 time unit (which we represent by the event *tock*) passes between the receipt of data on the channel *in* and the output of data on *out*, then it is assumed that the data has been lost by the buffer. Again, retransmission can be used as a means of overcoming this type of fault.

As with our previous examples, we define

$$Data = \{0, 1\}$$

We shall assume that no more than one out of every two transmissions can be lost in the manner described above. As such, we define the process $Fault_3$.

$$\begin{aligned} Fault_3 \mathrel{\widehat{=}}\ & ch_1?d \rightarrow (ch_2!d \rightarrow Fault_3 \sqcap tock \rightarrow ch_1?d \rightarrow ch_2!d \rightarrow Fault_3) \\ & \Box \\ & tock \rightarrow Fault_3 \end{aligned}$$

The event *tock* can occur independently of communications on the channels ch_1 and ch_2.

Data is received by the process R_3 every second, and is subsequently transmitted over channel ch_1.

$$R_3 \mathrel{\widehat{=}} in?d \rightarrow ch_1!d \rightarrow R_3'(d)$$

The process R_3' waits for acknowledgment that the value d has been successfully received (represented by the event *received*). Alternatively, if a time unit passes without such a communication, then the value is retransmitted.

$$R_3'(d) \mathrel{\widehat{=}} received \rightarrow tock \rightarrow R_3 \,\Box\, tock \rightarrow ch_1!d \rightarrow R_3'(d)$$

As we delay the receipt of new data on the channel ch_1 until the current value has been received by ch_2, the event *tock* is considered to be a preventative measure. As such, we define

$$R_{ch_2} = \{tock, received\}$$

After receiving the value d on channel ch_1, the process $Line_3$ should pass on this value on channel ch_2. Again, we model the possibility of faulty behaviour, in the case, the loss by $Line_3$ of d.

$$\begin{aligned} Line_3 &\mathrel{\widehat{=}} ch_1?d \rightarrow Line_3'(d) \,\Box\, tock \rightarrow Line_3 \\ Line_3'(d) &\mathrel{\widehat{=}} ch_2!d \rightarrow Line_3 \,\Box\, tock \rightarrow Line_3 \end{aligned}$$

If a value is received on channel ch_2, then acknowledgement is given.

$$T_3 \mathrel{\widehat{=}} ch_2?d \rightarrow received \rightarrow out!d \rightarrow T_3$$

We compose our system in a similar fashion to our previous examples.

$$Comm_3 \mathrel{\widehat{=}} (R_3 \,[\![\, ch_1 \cup \{tock\} \,]\!]\, Line_3) \,[\![\, ch_2 \cup \{received\} \,]\!]\, T_3$$

Furthermore, we define Sys_3 thus:

$$Sys_3 \mathrel{\widehat{=}} (Comm_3 \,[\![\, ch_1 \cup ch_2 \cup \{tock\} \,]\!]\, Fault_3) \setminus (ch_1 \cup ch_2 \cup \{received, tock\})$$

Proposition 4.7 $in \cup out \dashv\!\!+ ch_1 \cup ch_2 \cup \{received, tock\}$

Proof. By observation,

$$\begin{aligned} &\alpha Sys_3 \subseteq (in \cup out) \cup (ch_1 \cup ch_2 \cup \{received, tock\}) \wedge \\ &(in \cup out) \cap (ch_1 \cup ch_2 \cup \{received, tock\}) = \varnothing \end{aligned}$$

FDR confirms that Sys_3 is deterministic. As

$$\alpha Sys_3 \cap (ch_1 \cup ch_2 \cup \{received, tock\}) = \varnothing$$

by Proposition 3.2, $\mathbf{det}(Sys_3 \,|||\, RUN_{ch_1 \cup ch_2 \cup \{received, tock\}})$. By Definition 2.5,

$$in \cup out \not\dashv\!\!+ ch_1 \cup ch_2 \cup \{received, tock\}\ [Sys_3]$$

By Proposition 2.4,

$$in \cup out \rightarrowtail ch_1 \cup ch_2 \cup \{received, tock\}\ [Sys_3]$$

□

4.3.4 A General Approach to Safety

Together, our approaches to failure modes and fault-tolerance have helped emphasise the distinction which our model provides between faults, failures, and hazards. A *fault* is a known system deficiency for which we can prescribe means of avoidance, which we term *preventative measures.* A *failure*, the possibility of which may or may not be known in advance, may occur as a consequence of an unprevented fault or may occur independently. We have seen how different failure modes might deal with such failures. If the possibility of a failure is known in advance, then we can attempt to limit its damage by means of *recovery measures.* If we cannot provide a sufficient means of recovery from a failure and the system's safety-critical activity is affected, then a *hazard* might occur — this can be viewed as a 'point of no return' as the behaviour of the system is no longer predictable and a threat to safety may result.

Given the above, we say that a process P is *safe*, denoted $Safe(P)$, if and only if none of its failures are catastrophic. In addition, we say that P is *dependable*, denoted $Dep(P)$, if and only if P is fault-tolerant with respect to each of its faults.

Definition 4.11 A process P is *safe*, denoted $Safe(P)$, if and only if

$$\forall f : \Sigma \mid f \in fails(P) \bullet \neg Catastrophic(P, \{f\})$$

□

Definition 4.12 A process P is *dependable*, denoted $Dep(P)$, if and only if

$$\forall f : \mathbb{P}\,\Sigma \mid f \in faults(P) \bullet FT(P, f)$$

□

4.4 Safety Cores

Suppose that we wished to design a system in such a way that safety-critical behaviour was controlled by a single dedicated component. In addition to being dependable and safe, we might require that the system's safety-critical activity is protected from all other activity that is not so classified. We term such a dedicated component a *safety core.*

An example of such a system is the safety-critical software in the Darlington nuclear power generating station [Par95]. Here, as with all Canadian nuclear power stations, three control systems are required — the reactivity control system is responsible for power adjustments under normal working conditions, while the other two, known as safety or shutdown systems have the sole task of shutting down the reactor following a failure: "Canadian regulations forbid assigning any functions that are not essential for the shutdown task to a shutdown system." [Par95]

Definition 4.13 A process P has a *safety core* if and only if P can be written as an equivalent process $Core \parallel NonSafety$ such that $Core$ is concerned solely with the system's safety-critical behaviour, i.e.,

$$safety(P) \subseteq \alpha Core \land benign(P) \cap \alpha Core = \varnothing$$

□

A property associated with this concept, termed *safety core protection*, is defined as follows.

Definition 4.14 A process P is *safety core protected* if and only if P has a safety core and

$$\alpha Core \dashv\!\!+ \alpha P - \alpha Core\ [P]$$

□

Take, as an example, a valve which can be in one of two states (open or closed) at any time. If the valve is left open for three or more seconds, then a threat to safety might arise — we consider this to be a hazard. Thus, one of this system's safety requirement is that the valve must never be open for two or more seconds — this gives a 'margin for error' before a hazard can occur. We consider the valve being open for two or more seconds to be a failure.

We model the system hazard by the process *HA*. Following an opening of the valve, the system behaves as the process *Open*.

$$HA \mathrel{\hat{=}} valveOpen \rightarrow Open \,\Box\, tock \rightarrow HA$$

If two seconds pass before the valve is closed, then a failure is observed.

$$\begin{aligned} Open \mathrel{\hat{=}}\ & valveClosed \rightarrow HA \\ & \Box \\ & tock \rightarrow (valveClosed \rightarrow HA \,\Box\, tock \rightarrow \underline{failure} \rightarrow Failure) \end{aligned}$$

If the valve is not closed within a further second, then a hazard is observed.

$$\begin{aligned} Failure \mathrel{\hat{=}}\ & valveClosed \rightarrow \underline{recovery} \rightarrow HA \\ & \Box \\ & tock \rightarrow \underline{hazard} \rightarrow \bot_{\{valveOpen, valveClosed, tock, \underline{failure}, \underline{hazard}, \underline{recovery}\}} \end{aligned}$$

The system's required behaviour is given by the process *Core*. Here, no more than one second may pass between the valve opening and closing.

$$\begin{aligned} Core \mathrel{\hat{=}}\ & valveOpen \rightarrow (tock \rightarrow valveClosed \rightarrow Core \,\Box\, valveClosed \rightarrow Core) \\ & \Box \\ & tock \rightarrow Core \end{aligned}$$

We model the behaviour of the system with respect to its safety requirement by the process *Valve*.

$$Valve \mathrel{\hat{=}} HA \,[\![\, \{valveOpen, valveClosed, tock\} \,]\!]\, Core$$

If it is possible for the system hazard to occur, then nondeterminism will be brought about by $\bot$. It follows that if

$$\{valveOpen, valveClosed, tock\} \nplus \{\underline{failure}, \underline{recovery}, \underline{hazard}\}\ [Valve]$$

then this system is safety-core protected. Via FDR, together with our definition of protection, we can verify that this property holds.

As a further example, we take a chemical reactor system, which consists of three tanks and is represented by the process *Tanks*.

$$Tanks \mathrel{\widehat=} Tank_1 \ |||\ Tank_2 \ |||\ Tank_3$$

Furthermore, this reactor deals with three chemical processes, which may be carried out in any tank.

$$Processes \mathrel{\widehat=} Process_A \ |||\ Process_B \ |||\ Process_C$$
$$Reactor \mathrel{\widehat=} Processes \parallel Tanks$$

The safety-critical events associated with *Reactor* are observed by the process *Controller*, which is the system's safety core. As such, we define

$$System \mathrel{\widehat=} Reactor \parallel Controller$$

By Definition 4.14, this process is safety-core protected if

$$\alpha Controller \nplus \alpha System - \alpha Controller\ [System]$$

4.5 Firewalling

In defining our notion of a safety core, we suggested that all safety-critical components of a system might be incorporated into one 'module', and that the behaviour of this module should be unaffected by the behaviour of the rest of the system.

An alternative view is that we should divide a system into smaller subcomponents, thus factoring out the safety-critical behaviour between different subprocesses; we could then attempt to prove that a failure in one subprocess cannot have an adverse affect upon the behaviour of the others. If such a property holds of a process, then we shall say that it is 'firewalled' [McD93].

In addition to being an example of a safety core, the Darlington nuclear power generating station which was cited in the previous section is also an example of firewalling — the two shutdown systems should be independent of one another and a failure in one should not affect the other.

We define our property of firewalling in the following way.

Definition 4.15 Given a process P, composed of subprocesses $S = \{s_1, s_2, \ldots, s_n\}$, we say that P is *firewalled* if and only if

$$\forall\, s, s' : S \mid s \neq s' \bullet safety(s) \nplus fails(s')\ [P]$$

□

4.6 Dependability

4.6.1 Some Definitions

It is postulated in [Lap89] that there are two main measures of dependability (q.v.): *reliability* (q.v.) and *availability* (q.v.). These are defined as "a measure of the continuous delivery of proper service — or equivalently, of the time to failure"; and "a measure of the delivery of proper service with respect to the alternation of proper and improper service" respectively. Furthermore, reliability is defined in [Lev95] as being "the probability that a piece of equipment or component will perform its intended function satisfactorily for a prescribed time and under stipulated environmental conditions." Moreover, [Lev95] relates reliability and failure by defining unreliability to be the probability of the latter.[2]

Thus, dependability in general, is a central issue with regard to safety. Below are three distinct definitions of dependability. The first is given in terms of failures, the second in terms of faults, and the final definition is given in terms of reliability. Being defined in these ways, we can use protection to represent each interpretation.

"Dependable means safe with respect to the non-occurrence of catastrophic failures."

We have already seen how our approach can be applied to reason about dependability in this sense — we attempt to prove that no trace of a system specification can ever lead to a (known) hazard. Recall from Chapter 2 that we achieve this by developing a hazard analysis *HA* in which hazardous states are represented by $\perp$, and composing it with our specification *Spec*. If **nondiv**(*Spec* || *HA*), then we conclude that a hazard can never occur in *Spec*.

Recall also we say that *Safe*(*P*) holds of a process *P* if none of *P*'s failures can interfere with its critical events. As such, our first definition of dependability is as follows.

Definition 4.16 A process *P* has *type 1 dependability*, denoted $Dep_1(P)$, if and only if *Safe*(*P*). □

"Dependable means secure with respect to the avoidance, or tolerance, of deliberate faults."

Recalling our definition of the property *Dep*, this definition of dependability can be recast in terms of our definition of fault-tolerance in the following way.

Definition 4.17 A process *P* has *type 2 dependability*, denoted $Dep_2(P)$, if and only if *Dep*(*P*). □

"Dependable means reliable with respect to the continuity of service."

If we think of continuity of service in terms of failures non-interfering with non-critical events, then protection provides a means of interpreting this kind of dependability in a CSP context as follows.

[2] See also [McD94], in which a "loss and risk-based" definition of dependability is proposed.

Definition 4.18 A process P has *type 3 dependability*, denoted $Dep_3(P)$, if and only if

$$FailOp(P, fails(P), recovery(P))$$

□

4.6.2 Further Notions of Dependability

In this section, we provide simple CSP representations of three further measures of dependability: least time to failure, least time between failures, and greatest time to recover. Our representation of time (by means of *tock*) places a restriction on the measures of dependability that we can reason about. It should be noted that in a timed probabilistic model, it might be possible to reason about further types of dependability, e.g., mean time to failure, etc.

Least Time To Failure

Given a process P, we might wish to determine the length of time that the system is guaranteed to make progress without failing. That is, we might wish to determine its *least time to failure*. To this end, we define the following monitor process.

$$LTF(F, t) \mathrel{\widehat{=}} (Wait_t \triangle (f : F \rightarrow \bot_{F \cup \{tock\}})) \mathbin{⨟} Stop$$

Here, F is a set of failures, and we wish to guarantee that t time units pass before some failure $f \in F$ occurs. The process *Wait* is defined as follows.

$$Wait_t \mathrel{\widehat{=}} \text{if } t = 0 \text{ then } Skip \text{ else } tock \rightarrow Wait_{t-1}$$

If P fails before time t, then $P \parallel LTF(fails(P), t)$ will lead to $\bot_{F \cup \{tock\}}$. We define this notion of dependability in terms of the function *LTF*.

Definition 4.19 Assuming the function

$$\begin{array}{|l} LTF : PROC \rightarrow \mathbb{N} \\ \hline \forall P : PROC \bullet LTF(P) = \mu\, n : \mathbb{N} \mid \mathbf{nondiv}(P \parallel LTF(fails(P), n)) \wedge \\ \qquad\qquad\qquad\qquad \forall n' > n \bullet \neg \mathbf{nondiv}(P \parallel LTF(fails(P), n')) \end{array}$$

the *least time to failure* of a process $P \in PROC$ is given by $LTF(P)$. □

Least Time Between Failures

Instead of guaranteeing that a certain amount of time may pass before a failure can occur, we might wish to establish that a certain amount of time is guaranteed to pass between each failure. That is, we might wish to establish the *least time between failures*.

We again start by defining a monitor process.

$$LBF(F, t) \mathrel{\widehat{=}} tock \rightarrow LBF(F, t) \mathbin{\Box} \Box_{f \in F} f \rightarrow LBF'(F, t)$$

Here, either *tock* is observed, in which case there is no change in state, or a failure $f \in F$ is observed. In this case, LBF' is invoked. This process is defined thus:

$$LBF'(F, t) \mathrel{\widehat{=}} (Wait_t \triangle (f : F \rightarrow \bot_{F \cup \{tock\}})) \fatsemi LBF(F, t)$$

We define the least time between failures of a process in the following way.

Definition 4.20 Assuming the function

$$LBF : PROC \rightarrow \mathbb{N}$$

$$\forall P : PROC \bullet LBF(P) = \mu\, n : \mathbb{N} \mid \mathbf{nondiv}(P \parallel LBF(fails(P), n)) \land \forall\, n' > n \bullet \neg \mathbf{nondiv}(P \parallel LBF(fails(P), n'))$$

the *least time between failures* of a process $P \in PROC$ is given by $LBF(P)$. □

Greatest Time To Recover

Assume a process P. If we need to guarantee that no more than a certain amount of time can elapse between some failure $f \in fails(P)$ occurring and an observation of the subsequent recovery from that failure, r, then we would need to reason about the *greatest time to recover* from t.

We start by introducing the partial function

$$Rec : PROC \times \Sigma \nrightarrow \mathbb{P}\, \Sigma$$

Here, given a process $P \in PROC$ and a failure $f \in fails(P)$, $Rec(P, f) \subseteq recovery(P)$ defines the set of events denoting recovery from f in P.

We define the process *GTR* thus:

$$GTR(f, t, R) \mathrel{\widehat{=}} tock \rightarrow GTR(f, t, R) \,\Box\, f \rightarrow GTR'(f, t, R, 0)$$

Here, if *tock* is observed, then there is no change in state. Alternatively, if the failure f is observed, then the process GTR' is invoked with the appropriate values.

$$\begin{aligned} GTR'(f, t, R, count) \mathrel{\widehat{=}} \; & \mathbf{if} \quad count > t \\ & \mathbf{then} \; \bot_{R \cup \{f, tock\}} \\ & \mathbf{else} \; \Box_{r \in R}\, r \rightarrow GTR(f, t, R) \\ & \quad \Box \\ & \quad tock \rightarrow GTR'(f, t, R, count + 1) \end{aligned}$$

Given the above, we define the *greatest time to recover* for a process P in the following way.

Definition 4.21 Assuming the function

$$
\begin{array}{|l}
GTR : PROC \rightarrow \mathbb{N} \\
\hline
\forall P : PROC \bullet GTR(P) = \mu\, n : \mathbb{N} \mid \forall f : fails(P) \bullet \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{nondiv}(GTR(f, n, recovery(P, f)) \parallel P) \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad \forall n' > n \bullet \exists f : fails(P) \bullet \\
\qquad\qquad\qquad\qquad\qquad\qquad \neg\mathbf{nondiv}(GTR(f, n', recovery(P, f)) \parallel P)
\end{array}
$$

the *greatest time to recover* of a process $P \in PROC$ is given by $GTR(P)$. □

4.7 Human Factors

Recall from Section 1.1 that safety is not just a software or hardware matter: it involves the system as a whole, including those humans incorporated within the system. As humans are notoriously difficult entities to reason about, it follows that developing systems around humans is an extremely complex task.

It can be argued that, paradoxically, *human factors* (q.v.) become increasingly important as society becomes more automated. In recent years, the study of cognitive, rather than the more traditional physical factors, has driven such research.

Most hazard analyses which incorporate the 'human element' tend to concentrate on human operators and their possible mistakes. In this respect, humans can play one of three roles within the safety-critical environment [Lev95]: 'partner', 'monitor', or 'backup'. However, as well as those dangers which may result from an operator's mistakes, human interaction can pose a further threat to safety: that of the *malicious threat.*

It is in some ways naïve to suggest, as we did in Chapter 2, that any threat from safety-critical systems comes solely from the failure of that system to the environment. For, just as the system's human components may pose a threat to security, so they may pose a threat to safety by contributing to the possibility of failures. For example, an office worker might disable an overactive smoke alarm which, of course, poses a threat to the safety of others who should be protected by that system.

Looking once again to the world of security for inspiration, it is not uncommon for security-critical processes to be classified as *trusted*, *benign*, or *malicious* [Gas88]. It is our belief that human interaction with safety-critical systems can be classified in a similar way.

4.7.1 Three Classes of Behaviour

Trusted Behaviour

We think of trusted users as corresponding to the notion of the ideal user of [Cla93]:

> "An ideal user is someone who understands the engineering constraints, the scope of functionality and the full extent of the 'requirements'. He or she probably does not want to do anything with the system other than make it work to its specification, so will not seek to use it to meet some external objectives such as production targets, return on investment, or military advantage."

For example, we would describe pilots in this fashion, as they play an integral role in the safety-critical behaviour of aeroplanes. Another example is a train driver, or, for that matter, any operator of safety-critical equipment.

Neutral Behaviour

Humans who interact with a system might be described as having a neutral interface if their behaviour cannot increase or decrease the safety level of that system in any way. For example, a passenger in an aeroplane cannot influence the system's safety-critical behaviour, but can fashion its benign behaviour (perhaps by interacting with the in-flight entertainment component of a system). A further example of neutral human behaviour is that of an observer of safety-critical equipment. Note that even though this person's safety might be directly reliant upon the safe behaviour of the system, there might be no circumstances under which they can influence the system's safety-critical behaviour.

Malicious Behaviour

The final category of human behaviour which we reason about is that of the malicious interface. This involves a person who willfully causes damage to the system, perhaps by physical means, or by altering safety-critical code in such a way that the level of safety of that system is reduced. For example, acts of vandalism or malicious interference with software are such interactions.

Rather than giving classifications to individuals who interact with a safety-critical system, we assign classifications to interfaces. This is because an obvious problem with assigning classifications to individuals is that we would not be able to prevent trusted users from carrying out malicious actions.

As such, we define three interfaces corresponding to the above classes of behaviour, each with distinct sets of events. It follows that the trusted interface consists of interactions carried out at a safety-critical level; the neutral interface consists of interactions carried out at a benign level; and the malicious interface consists of interactions which must be either prevented or tolerated at all costs.

4.7.2 The Human Factors Model

Our first attempt at a suitable non-interference model for human factors is given by Figure 4.3.

Here, our system and its trusted interface are seen to co-exist as one, with information flowing freely between the two. In addition, the system is protected from malicious behaviour, although information can flow (for example, by preventative measures), in the opposite direction. Finally, there is no information flow either to or from the neutral interface. We also assume that there is no information flow between malicious and neutral interfaces.

This model is sufficient as a means of providing an overview of how information should flow between a safety-critical system and the outside world, but is not sophisticated enough to take into account the different types of events in which a safety-critical system may participate. For example, in Figure 4.3, we see that there is no flow of information to and from the neutral interface, whereas it might be perfectly acceptable for it to flow between this

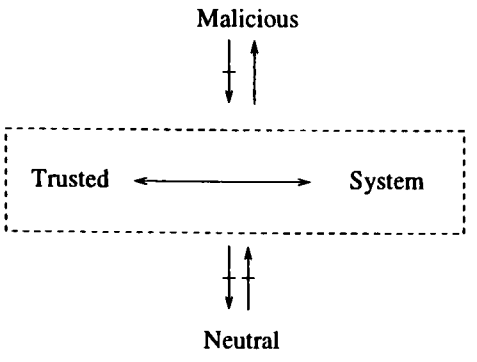

Figure 4.3: Human factors (model 1).

interface and the system's benign events. As we already have in place a general model for the description of safety-critical systems, we expand upon this to provide a model of human interactions.

As before, we place the events in which a safety-critical system may participate into four classes: *safety*, *benign*, *fails*, and *recovery*, such that *fails* includes faults and hazards and *recovery* includes fault prevention.

Before constructing our model, we first need to establish between which sets we want protection to hold, and where we want information to flow freely. To this end, we make the following observations.

- Neither malicious behaviour nor random failures can be controlled by our system, and, as such, we cannot prevent information flowing between them.
- There must be information flow between malicious behaviour and failures, and trusted behaviour and recovery events in order to prevent and recover from such activity.
- The only interface possible for neutral behaviour involves information flowing to and from benign system behaviour.
- We assume a fail-soft approach, and allow information flow relevant to this property.
- We assume that there is no information flow between any of the external interfaces.

Our model is given by Figure 4.4.

With the exception of insisting that the system fails-safe with respect to malicious behaviours and its failures, the only other non-interference properties which we insist upon are those restricting information flow between the interfaces.

4.7.3 Protection and Human Factors

The Trusted Interface

When considering *trusted* users, we do not restrict information flow either to or from the system's safety-critical or benign components. However, we do need to ensure that the neutral interface is protected from the trusted interface.

Definition 4.22 A process P with trusted interface $T \subseteq \alpha P$ and neutral interface $N \subseteq \alpha P$ is *trusted interface protected* if and only if $N \nrightarrow T\ [P]$. □

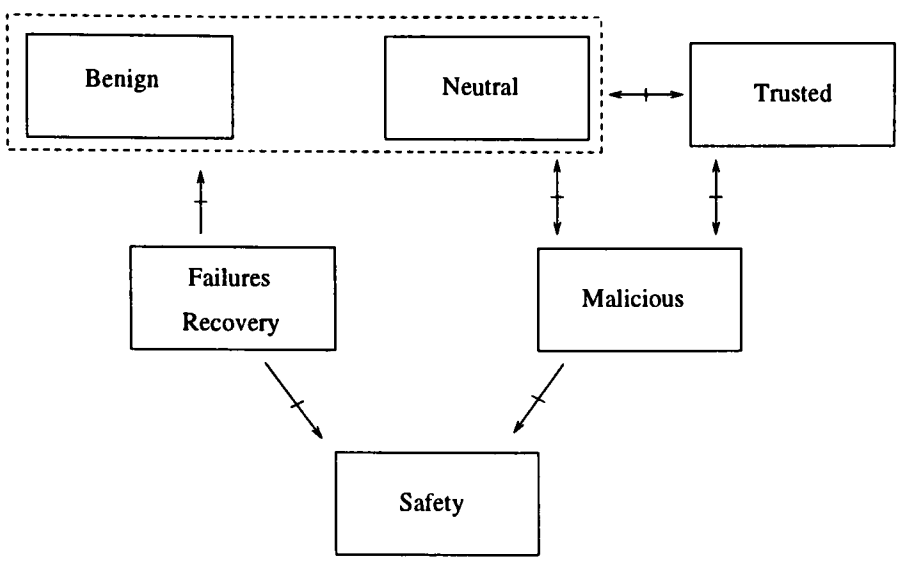

Figure 4.4: Human factors (model 2).

That is, a process is trusted interface protected if and only if the interfaces of users at the neutral interface are unaffected by the behaviour of users at the trusted interface.

The Neutral Interface

Given our understanding of neutral interaction, we require that the only communication such events have with a system is with its benign behaviour.

Definition 4.23 A process P with neutral interface $N \subseteq \alpha P$ is *neutral interface protected* if and only if

$$N \cup benign(P) \nrightarrow \alpha P - (N \cup benign(P))\ [P]$$
$$\wedge$$
$$\alpha P - (N \cup benign(P)) \nrightarrow N \cup benign(P)\ [P]$$

□

Given the above, if P is neutral interface protected, then, by implication, it is also trusted interface protected.

Proposition 4.8 If a process P is neutral interface protected, then it is trusted interface protected.

Proof. Assume P is neutral interface protected. By Definition 4.23,

$$N \cup benign(P) \nrightarrow \alpha P - (N \cup benign(P))\ [P]$$
$$\wedge$$
$$\alpha P - (N \cup benign(P)) \nrightarrow N \cup benign(P)\ [P]$$

As $T \subseteq \alpha P - (N \cup \mathit{benign}(P))$, by Laws A.1 and A.2,

$$N \cup \mathit{benign}(P) \nrightarrow T \ [P]$$
$$\wedge$$
$$T \nrightarrow N \cup \mathit{benign}(P) \ [P]$$

Again, by Law A.1, $N \nrightarrow T \ [P]$. Therefore, by Definition 4.22, P is trusted interface protected. □

The Malicious Interface

We would obviously wish safety-critical systems to be unaffected by malicious behaviour. If we think of such an interface as failures, then we would wish our system to fail-safe with respect to such behaviour. We again provide a suitable definition for such a non-interference property in terms of protection.

Definition 4.24 A process P with malicious interface $M \subseteq \alpha P$ is *malicious interface protected* if and only if $\mathit{safety}(P) \nrightarrow M \ [P]$. □

Given the above, we define human factors protection in the following way.

Definition 4.25 A process P with trusted interface $T \subseteq \alpha P$, neutral interface $N \subseteq \alpha P$, and malicious interface $M \subseteq \alpha P$ is *human factors protected* if and only if

- $\forall x, x' : \{M, N, T\} \mid x \neq x' \bullet x \nrightarrow x' \ [P]$,
- P is neutral interface protected, and
- P is malicious interface protected.

□

We have seen how our approach to modelling non-interference with CSP has allowed us to describe a number of important safety-critical concepts. In the next chapter, we present a case study in which we apply some of these concepts, and analyse a number of non-interference properties of a specification of a simplified safety-critical system.

Chapter 5

A Case Study: Automatic Train Protection

In this chapter, we put the theory of the previous three chapters into practice by specifying, and proving non-interference properties of, a simplified Automatic Train Protection (ATP) system. The main task of such a system is to provide a 'safety envelope' which protects against the consequences of driver error. For example, if an ATP-fitted train's actual speed exceeds its required speed, then the system warns the driver accordingly. If this warning is ignored, then the train's emergency brakes are applied automatically, which brings the train to a safe halt.

We start by providing an overview of automatic train protection, and propose a number of requirements which are desirable of such a system. Some of these requirements are characterised in terms of non-interference.

Work described in this chapter has previously appeared as [Sim94] and [Sim95a].

5.1 Automatic Train Protection

5.1.1 The Background

A system known as AWS (Automatic Warning System) has been implemented on approximately 70,000 miles of British railways, which equates to 70% of the total network [Hal92]. The purpose of the system is to inform the train driver at appropriate times that he needs to slow down or stop. For example, if the next signal to be passed is displaying a proceed signal, then no action need be taken by the driver. On the other hand, if the next signal is displaying a stop signal and the train is travelling too fast towards that signal, then a warning is sounded. If the driver fails to acknowledge such a warning, then the train's brakes are applied automatically within two to three seconds.

Despite the fact that AWS has undoubtedly helped in reducing the number of collisions on British railways, a major drawback of the system is that it does not monitor the driver's response to a warning. This presents the possibility of a driver pressing the acknowledgement button and then failing to brake sufficiently. In particular, the following point is made in [BRB94].

> "A number of accidents have occurred as a result of a driver acknowledging the AWS warning but not applying the brakes to prevent the train overshooting the

signal at danger."

For example, a signal passed at danger at Purley in South London in 1989, "would have been prevented if ATP [had] been installed at the locations and on the trains concerned." [BRB94]

5.1.2 ATP Structure

An ATP system consists of three components: wayside equipment, trainborne equipment, and a track-to-train transmission system.

Wayside Equipment Wayside equipment is located near signals and generates data, such as signal aspects and maximum permitted speeds, which is relevant to that section of track.

Trainborne Equipment Trainborne equipment handles information received from the track, and includes the driver's interface and the train's brakes interface.

Track-to-train Transmission The track-to-train transmission transmits information from the wayside equipment to the train. This can be accomplished either by continuous or intermittent transmission. The former continuously updates and transmits information regarding the line ahead to a train, while the latter transmits data to a train when it passes either individual devices in the track at the signals or 'fill-in' devices between signals. In the following, we assume intermittent transmission from beacons.

5.1.3 System Aims

The main aim of an ATP system is to provide a high level of train safety by means of supervising the driver's alertness. It automatically protects trains against the consequences of driver error and automatically applies the train's emergency brakes whenever necessary. Such a system is implemented in the following way.

When a train passes a beacon located in the track, information relevant to that section is transmitted to that train. This information is then relayed to the trainborne computer. It is on the basis of this information, together with fixed information such as the train's length and braking performance, and information such as the train's current position and speed which are received from the tachometer, that braking curves are calculated. An onboard computer continuously checks the actual speed of the train against the braking curve which it has calculated and gives a warning if it finds that the actual speed exceeds that specified by the braking curve. If necessary, this is followed by an automatic application of the train's brakes. The maximum permitted line speed is also safeguarded by the system by continuously checking the train's actual speed against its maximum permitted speed. If the former exceeds the latter, then the ATP system will intervene, which should result in speed being reduced sufficiently.

We summarise the requirements of our simplified ATP system below.

1. If the driver is driving below the required speed limit, then no action is necessary.
2. If a signal displaying red is passed, then the emergency brakes should be applied immediately.

3. If a signal displaying yellow is passed, then the required braking curve should be calculated in anticipation of the next signal displaying red.
4. If the train is travelling too fast towards a signal displaying red, then a warning should be given. If this warning goes unheeded, then the emergency brakes should be applied immediately.
5. If the train is travelling too fast towards a proceed signal, then the service brakes should be applied automatically to reduce speed.
6. If a failure of the ATP system is detected (e.g., no message is received when a train approaching a signal displaying red passes over a beacon), then the emergency brakes should be applied immediately.
7. If the emergency brakes have been applied, then they should be released only after the train has been brought to a complete halt.

The reader is referred to [Sim94] for a CSP specification which satisfies these requirements. In the following, we build upon that work, and develop a specification which satisfies a number of non-interference properties. These properties are described in the next section.

5.2 Non-interference and ATP

In this section, we introduce a number of ATP requirements which can be characterised in terms of non-interference and analysed in terms of our protection-based safety-critical properties.

Fault-tolerance We shall assume that our link from the transmission system to trains is faulty, and that no more than 4 out of every 5 consecutive messages are guaranteed to be received correctly. As such, we attempt to develop a fault-tolerant specification which combats this fault by means of redundancy.

Fail-operational If we characterise the event of a train's actual speed exceeding either its required speed or its permitted speed as a failure, and the sounding of an automatic warning together with a subsequent reduction in speed as the corresponding recovery procedure for this failure, then we might attempt to prove that such a sequence of events affects neither the safety-critical nor the non-critical behaviour of the system. In this sense, the system fails-operational with respect to the detection of excess speed.

Fail-safe We might deem the passing of a signal at danger to be a failure. In this case, our recovery measure is the automatic application of the train's brakes, which should result in the train being brought to a halt in the overlap section in advance of the signal. If this property holds, then the system fails-safe with respect to a signal being passed at danger.

A Safety Core One of the keys to the safe operation of a system such as ATP is that nothing should interfere with the system's ability to receive new information with regard to signal aspects and permitted line speeds — we endeavour to prove that such communications are protected from all other observable events.

Firewalling Our description of the wayside and transmission systems should take into account the fact that both systems deal with many trains at once. As such, we attempt to prove that the system is firewalled in the sense that the behaviour of the onboard equipment of one train is non-interfering with the onboard equipment of all others.

5.3 A Specification

5.3.1 Basic Types

We shall assume that all signals are *three-aspect signals*, i.e., they may display one of three aspects at any one time. We define the free type *Aspect* accordingly.

$$Aspect ::= red \mid yellow \mid green$$

In addition, we define the free type *Speed*.

$$Speed ::= high \mid mid \mid low$$

We also assume given types of beacons and trains.[1]

$$[Beacon, Train]$$

5.3.2 Events and Channels

The wayside communicates with the transmission system via events contained in the set

$$track2trans = \{b : Beacon;\ a : Aspect;\ s : Speed \bullet track2trans.b.a.s\}$$

Here, the communication $track2trans.b.a.s$ represents the transmission from beacon b that the signal in advance of b is currently displaying aspect a, and the maximum permitted speed for the section of track is s.

In turn, the transmission system communicates with trains on the channel *trans2trains*.

$$trans2trains = \{t : Train;\ a : Aspect;\ s : Speed \bullet trans2trains.t.a.s\}$$

Here, the communication $trans2trains.t.a.s$ represents the transmission to train t that the next signal is displaying aspect a, and the maximum permitted speed for the current section of track is s.

A communication on the channel *passedBeacon* indicates that train $t \in Train$ has passed beacon $b \in Beacon$.

$$passedBeacon = \{b : Beacon;\ t : Train \bullet passedBeacon.b.t\}$$

Trains can also request data from beacons via the channel *initState*.

$$initState = \{b : Beacon;\ t : Train \bullet initState.b.t\}$$

[1] In actuality, we limit the cardinality of these sets to reduce the complexity of mechanical verification.

Here, *initState.b.t* represents a request by train t for information from beacon b.

A number of channels are specific to trains. These are interpreted for any $t \in Train$, $b \in Beacon$, $a \in Aspect$, and $s \in Speed$ in the following way.

- *location.t.b* — b is the closest beacon to t's initial position.
- *passedSignal.t* — t passes a signal.
- *readAspect.t.a* — a is the aspect of the signal in advance of t.
- *readPermittedSpeed.t.s* — s is t's permitted speed.
- *readRequiredSpeed.t.s* — s is t's required speed.
- *readActualSpeed.t.p.s* — s is t's actual speed, read against its permitted speed.
- *readActualSpeed.t.r.s* — s is t's actual speed, read against its required speed.
- *tooFast.t.p* — t's actual speed exceeds its permitted speed.
- *tooFast.t.r* — t's actual speed exceeds its required speed.
- *slowEnough.t.p* — t's actual speed is within its permitted speed.
- *slowEnough.t.r* — t's actual speed is within its required speed.
- *emergencyBrakesOn.t* — t applies its emergency brakes.
- *emergencyStop.t* — t comes to an emergency stop.
- *emergencyBrakesOff.t* — t releases its emergency brakes.

Furthermore, updates of data relevant to a beacon are represented by communications on the channels *newAspect* and *newSpeed*.

$$newAspect = \{b : Beacon;\ a : Aspect \bullet newAspect.b.a\}$$
$$newSpeed = \{b : Beacon;\ s : Speed \bullet newSpeed.b.s\}$$

Finally, we assume two independent clocks for each train, and introduce the channel *tock* accordingly.

$$tock = \{t : Train;\ x : \{p, r\} \bullet tock.t.x\}$$

As an example, *tock.t.p* denotes the passing of a time unit on the clock which is associated with checking the permitted speed of train t, while *train.t.r* denotes the passing of a time unit on the clock which is associated with checking the required speed of t.

The Top Level

We take a top-down approach to this specification, and define our ATP system in terms of the track-to-train transmission system, trainborne equipment, and wayside equipment.

$$ATP \mathrel{\widehat{=}} Transmission \llbracket initState \cup initState \cup track2trans \cup trans2trains \rrbracket (Trains \,|||\, Wayside)$$

The processes *Transmission*, *Trains*, and *Wayside* are defined below. The connection diagram of Figure 5.1 might help to illuminate the relationships between these processes.

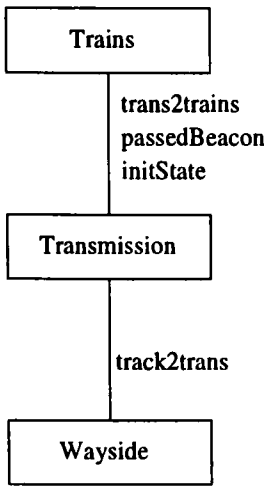

Figure 5.1: ATP connection diagram.

Trainborne Equipment

Given that our system deals with the set of trains *Train*, we define the process *Trains* accordingly.

$$Trains \mathrel{\widehat{=}} |||_{t \in Train}\, TrainborneInit(t)$$

After receiving the permitted speed of the current section of track and the aspect of the next signal, which we represent by a communication on the channel $trans2trains.t$, the trainborne equipment behaves as the parallel operation of the three safety checks with which we concern ourselves: the comparison of the train's actual speed against its required speed, the comparison of the train's actual speed against its permitted speed, and ensuring that the train is brought to a halt if it passes a signal at danger.

$$\begin{aligned} TrainborneInit(t) \mathrel{\widehat{=}}\ & location.t?b \rightarrow initState.b.t \rightarrow \\ & trans2trains.t?a?s \rightarrow Trainborne(t, a, s) \end{aligned}$$

Following the receipt of the aspect of the next signal, a, and the permitted speed, s, the process $Trainborne(t)$ behaves as the composition of five processes: $DataStorage(t, a, s)$, which continuously stores and updates relevant parameters; $SignalCheck(t)$, which checks the aspect of any signal passed by train t; $PermittedSpeedCheck(t)$, which checks t's actual speed against its permitted speed; $RequiredSpeedCheck(t)$, which checks t's actual speed against its required speed; and $PassBeacons(t)$, which observes when t passes a beacon.

$$\begin{aligned} Trainborne(t, a, s) \mathrel{\widehat{=}}\ & \\ & DataStorage(t, a, s) \\ & [\![\ \{trans2trains.t.* .*\} \cup \{readAspect.t.*\} \cup \{readPermittedSpeed.t.*\}\ ]\!] \\ & (SignalCheck(t) \ ||| \ PermittedSpeedCheck(t) \\ & ||| \\ & RequiredSpeedCheck(t) \ ||| \ PassBeacons(t)) \end{aligned}$$

The process *DataStorage* offers the choice between updating the train's parameters following a receipt of new data on the channel *trans2trains.t*, or the output of requested data.

$$\begin{aligned} DataStorage(t, a, s) \mathrel{\hat{=}} & \; trans2trains.t?a'?s' \rightarrow DataStorage(t, a', s') \\ & \Box \\ & readAspect.t!a \rightarrow DataStorage(t, a, s) \\ & \Box \\ & readPermittedSpeed.t!s \rightarrow DataStorage(t, a, s) \end{aligned}$$

If train t passes a signal displaying red, then the process *PassedSignalAtDanger* is invoked, which applies t's emergency brakes.

$$\begin{aligned} SignalCheck(t) \mathrel{\hat{=}} & \; passedSignal.t \rightarrow readAspect.t?a \rightarrow \\ & \textbf{if} \quad a = red \\ & \textbf{then} \; PassedSignalAtDanger(t) \\ & \textbf{else} \; SignalCheck(t) \end{aligned}$$

The process *PassedSignalAtDanger* assumes the successful application of emergency brakes. This is in the sense that if emergency brakes are applied, then the train will be brought to a halt in the overlap section in advance of the signal. As such, we define this process in the following way.

$$\begin{aligned} PassedSignalAtDanger(t) \mathrel{\hat{=}} & \; emergencyBrakesOn.t \rightarrow emergencyStop.t \rightarrow \\ & emergencyBrakesOff.t \rightarrow SignalCheck(t) \end{aligned}$$

Our second safety check concerns the comparison between the train's actual speed and its permitted speed. We assume that after each occurrence of *tock.t.p*, the two speeds are compared. We define the following process accordingly.

$$\begin{aligned} PermittedSpeedCheck(t) \mathrel{\hat{=}} & \; tock.t.p \rightarrow \\ & readActualSpeed.t.p?actual \rightarrow \\ & readPermittedSpeed.t?permitted \rightarrow \\ & \textbf{if} \quad actual > permitted \\ & \textbf{then} \; \underline{tooFast.t.p} \rightarrow TrainFasterThanPermitted(t) \\ & \textbf{else} \; \underline{slowEnough.t.p} \rightarrow PermittedSpeedCheck(t) \end{aligned}$$

If the train is travelling too fast, then it is assumed that appropriate action is taken to reduce speed — we do not concern ourselves with this action here. The train's speed is continually monitored until it has been reduced sufficiently.

$$\begin{aligned} TrainFasterThanPermitted(t) \mathrel{\hat{=}} & \; tock.t.p \rightarrow TrainFasterThanPermitted(t) \\ & \Box \\ & \underline{tooFast.t.p} \rightarrow TrainFasterThanPermitted(t) \\ & \Box \\ & \underline{slowEnough.t.p} \rightarrow PermittedSpeedCheck(t) \end{aligned}$$

The train's actual speed is compared with its required speed in a similar manner. Again, we do not concern ourselves with the action taken to reduce speed — we simply assume that it will be reduced sufficiently.

$$\begin{aligned} \mathit{RequiredSpeedCheck}(t) \mathrel{\hat{=}} \; & \mathit{tock.t.r} \rightarrow \\ & \mathit{readActualSpeed.t.r?actual} \rightarrow \\ & \mathit{readRequiredSpeed.t?required} \rightarrow \\ & \textbf{if} \quad \mathit{actual} > \mathit{required} \\ & \textbf{then} \; \underline{\mathit{tooFast.t.r}} \rightarrow \mathit{TrainFasterThanRequired}(t) \\ & \textbf{else} \; \underline{\mathit{slowEnough.t.r}} \rightarrow \mathit{RequiredSpeedCheck}(t) \end{aligned}$$

Here, the train's required speed is continuously monitored against its actual speed — when $\underline{\mathit{slowEnough.t.r}}$ is observed, this process returns to its previous state.

$$\begin{aligned} \mathit{TrainFasterThanRequired}(t) \mathrel{\hat{=}} \; & \mathit{tock.t.r} \rightarrow \mathit{TrainFasterThanRequired}(t) \\ & \Box \\ & \underline{\mathit{tooFast.t.r}} \rightarrow \mathit{TrainFasterThanRequired}(t) \\ & \Box \\ & \underline{\mathit{slowEnough.t.r}} \rightarrow \mathit{RequiredSpeedCheck}(t) \end{aligned}$$

Finally, the process $\mathit{PassBeacons}(t)$ always allows observations corresponding to train t passing a beacon.

$$\mathit{PassBeacons}(t) \mathrel{\hat{=}} \mathit{passedBeacon?b.t} \rightarrow \mathit{PassBeacons}(t)$$

The Wayside Equipment

Given that our ATP system is concerned with the set of beacons, *Beacon*, we define the process *Wayside* by

$$\mathit{Wayside} \mathrel{\hat{=}} |||_{b \in \mathit{Beacon}} \mathit{BeaconInit}(b)$$

Each beacon receives its initial values before it may transmit any values to trains. We assume that these values may be received in either order.

$$\begin{aligned} \mathit{BeaconInit}(b) \mathrel{\hat{=}} \; & \mathit{newAspect.b?a'} \rightarrow \mathit{newSpeed.b?s'} \rightarrow \mathit{Beacon}(b, a', s') \\ & \Box \\ & \mathit{newSpeed.b?s'} \rightarrow \mathit{newAspect.b?a'} \rightarrow \mathit{Beacon}(b, a', s') \end{aligned}$$

After beacon b's values have been initialised, they may be transmitted or updated *ad infinitum*.

$$\begin{aligned} \mathit{Beacon}(b, a, s) \mathrel{\hat{=}} \; & \mathit{newAspect.b?a'} \rightarrow \mathit{Beacon}(b, a', s) \\ & \Box \\ & \mathit{newSpeed.b?s'} \rightarrow \mathit{Beacon}(b, a, s') \\ & \Box \\ & \mathit{track2trans.b!a!s} \rightarrow \mathit{Beacon}(b, a, s) \end{aligned}$$

The Track-to-Train Transmission System

The transmission system transmits relevant data to a train $t \in Train$ whenever t requests initial data or passes a beacon in the track. The data is transmitted on a first-come, first-served basis, and initially there are no requests for data.

$$Transmission \mathrel{\widehat{=}} Trans_1(\langle\rangle, \langle\rangle)$$

If data is required by train t when in the section controlled by beacon b, then the process's parameters are updated. This process maintains queues of outstanding requests for data and deals with each in turn. To reduce the complexity of verification, there is a maximum queue size of 3 requests — if this limit is reached, then data is transmitted.

$$\begin{array}{l}
Trans_1(bqueue, tqueue) \mathrel{\widehat{=}} \\
\quad \textbf{if} \quad bqueue = \langle\rangle \\
\quad \textbf{then} \; initState?b?t \rightarrow Trans_1(\langle b\rangle, \langle t\rangle) \\
\qquad \Box \\
\qquad passedBeacon?b?t \rightarrow Trans_1(\langle b\rangle, \langle t\rangle) \\
\quad \textbf{else} \; \textbf{if} \quad \#bqueue \geq 3 \\
\qquad \textbf{then} \; track2trans.hd(bqueue)?a?s \rightarrow Trans_2(bqueue, tqueue, a, s) \\
\qquad \textbf{else} \; initState?b?t \rightarrow Trans_1(bqueue \frown \langle b\rangle, tqueue \frown \langle t\rangle) \\
\qquad\qquad \Box \\
\qquad\qquad passedBeacon?b?t \rightarrow Trans_1(bqueue \frown \langle b\rangle, tqueue \frown \langle t\rangle) \\
\qquad\qquad \Box \\
\qquad\qquad track2trans.hd(bqueue)?a?s \rightarrow Trans_2(bqueue, tqueue, a, s)
\end{array}$$

The option to receive further requests for data is taken away from this process only if the queue size has reached its maximum limit. In this case, transmission of data to the train at the head of the queue is enforced.

$$\begin{array}{l}
Trans_2(bqueue, tqueue, a, s) \mathrel{\widehat{=}} \\
\quad \textbf{if} \quad \#bqueue \geq 3 \\
\quad \textbf{then} \; trans2trains.hd(tqueue)!a!s \rightarrow Trans_1(tl(bqueue), tl(tqueue)) \\
\quad \textbf{else} \; initState?b?t \rightarrow Trans_2(bqueue \frown \langle b\rangle, tqueue \frown \langle t\rangle, a, s) \\
\qquad \Box \\
\qquad passedBeacon?b?t \rightarrow Trans_2(bqueue \frown \langle b\rangle, tqueue \frown \langle t\rangle, a, s) \\
\qquad \Box \\
\qquad trans2trains.hd(tqueue)!a!s \rightarrow Trans_1(tl(bqueue), tl(tqueue))
\end{array}$$

A Fault

Recall from Section 5.2 that we can guarantee that only 4 out of every 5 consecutive messages from the transmission system to the train will be transmitted correctly. In this section, we define a process which models this fault.

The process *TransmissionFault* is defined in terms of the unsynchronised parallel composition of faults over each of the channels $trans2trains.t$ for any $t \in Train$.

$$TransmissionFault \mathrel{\hat{=}} |||_{t \in Train} FaultInit(t)$$

The process *FaultInit* is defined for each $t \in Train$ in the following way. Following a fault, the data to be transmitted might be scrambled.

$$FaultInit(t) \mathrel{\hat{=}} initState?b.t \rightarrow Fault(t, b) \;\Box\; passedBeacon?b.t \rightarrow Fault(t, b)$$

$$\begin{aligned} Fault(t, b) \mathrel{\hat{=}} track2trans.b?a?s \rightarrow\ & trans2trains.t!a!s \rightarrow FaultInit(t) \\ & \sqcap \\ & fault.t \rightarrow Fault'(t) \end{aligned}$$

If a fault does occur, then any value could be transmitted.

$$Fault'(t) \mathrel{\hat{=}} \sqcap_{a' \in A;\, s' \in S} trans2trains.t!a'!s' \rightarrow CorrectBehaviour(t, 4)$$

Following an incorrect transmission, it is guaranteed that the next four communications will be transmitted correctly.

$$\begin{aligned} CorrectBehaviour(t, n) \mathrel{\hat{=}}\ & \textbf{if} \quad n = 0 \\ & \textbf{then}\ FaultInit(t) \\ & \textbf{else}\ initState?b.t \rightarrow TransmitCorrectly(t, b, n) \\ & \quad \Box \\ & \quad passedBeacon?b.t \rightarrow TransmitCorrectly(t, b, n) \end{aligned}$$

$$\begin{aligned} TransmitCorrectly(t, b, n) \mathrel{\hat{=}}\ & track2trans.b?a?s \rightarrow \\ & trans2trains.t!a!s \rightarrow CorrectBehaviour(t, n - 1) \end{aligned}$$

As a preventative measure to guard against this fault, we rewrite the process $Trans_2$, so that the relevant value is transmitted three times. However, we must ensure that this repeated transmission does not introduce interference by blocking further requests for data.

$$\begin{aligned} & Trans_2(bqueue, tqueue, a, s) \mathrel{\hat{=}} \\ & \textbf{if} \quad \#bqueue \geq 3 \\ & \textbf{then}\ trans2trains.hd(tqueue)!a!s \rightarrow Repeat_2(bqueue, tqueue, a, s, 0) \\ & \textbf{else}\ initState?b?t \rightarrow Trans_2(bqueue \frown \langle b \rangle, tqueue \frown \langle t \rangle, a, s) \\ & \quad \Box \\ & \quad passedBeacon?b?t \rightarrow Trans_2(bqueue \frown \langle b \rangle, tqueue \frown \langle t \rangle, a, s) \\ & \quad \Box \\ & \quad trans2trains.hd(tqueue)!a!s \rightarrow Repeat_2(bqueue, tqueue, a, s, 0) \end{aligned}$$

The process $Repeat_2$ is defined thus:

$$
\begin{array}{l}
Repeat_2(bqueue, tqueue, a, s, num) \mathrel{\hat{=}} \\
\textbf{if} \quad num \geq 2 \\
\textbf{then}\ Trans_1(tl(bqueue), tl(tqueue)) \\
\textbf{else}\ \textbf{if} \quad \#bqueue \geq 3 \\
\qquad \textbf{then}\ trans2trains.hd(tqueue)!a!s \rightarrow Repeat_2(bqueue, tqueue, a, s, num + 1) \\
\qquad \textbf{else}\ initState?b?t \rightarrow Repeat_2(bqueue \frown \langle b \rangle, tqueue \frown \langle t \rangle, a, s, num) \\
\qquad\qquad \Box \\
\qquad\qquad passedBeacon?b?t \rightarrow Repeat_2(bqueue \frown \langle b \rangle, tqueue \frown \langle t \rangle, a, s, num) \\
\qquad\qquad \Box \\
\qquad\qquad trans2trains.hd(tqueue)!a!s \rightarrow Repeat_2(bqueue, tqueue, a, s, num + 1)
\end{array}
$$

We also rewrite *DataStorage* in the following way.

$$
\begin{array}{l}
DataStorage(t, a, s) \mathrel{\hat{=}} trans2trains.t?a'?s' \rightarrow DataStorage'(t, a, s, \langle (a's') \rangle) \\
\qquad\qquad \Box \\
\qquad\qquad readAspect.t!a \rightarrow DataStorage(t, a, s) \\
\qquad\qquad \Box \\
\qquad\qquad readPermittedSpeed.t!s \rightarrow DataStorage(t, a, s)
\end{array}
$$

This version of *DataStorage* differs from the original in that the process waits for data to be received three times, with the majority vote of the three values being taken as the correct one. This majority vote is calculated by the process *DataStorage'*.

$$
\begin{array}{l}
DataStorage'(t, a, s, values) \mathrel{\hat{=}} \\
\quad \textbf{if} \quad \#values = 3 \\
\quad \textbf{then}\ \textbf{if} \quad values[1] = values[2] \vee values[1] = values[3] \\
\qquad \textbf{then}\ DataStorage(t, fst(values[1]), snd(values[1])) \\
\qquad \textbf{else}\ DataStorage(t, fst(values[2]), snd(values[2])) \\
\quad \textbf{else}\ readAspect.t!a \rightarrow DataStorage'(t, a, s, values) \\
\qquad\qquad \Box \\
\qquad\qquad readPermittedSpeed.t!s \rightarrow DataStorage'(t, a, s, values) \\
\qquad\qquad \Box \\
\qquad\qquad trans2trains.t?a'?s' \rightarrow DataStorage'(t, a, s, values \frown \langle (a', s') \rangle)
\end{array}
$$

Note that this process insists that the option to output aspects and speeds is not removed while the correct value is calculated.

Given the above, we define

$$
\begin{array}{l}
FaultyATP \mathrel{\hat{=}} (ATP \\
\qquad [\![\ initState \cup passedBeacon \cup track2trans \cup trans2trains\ ]\!] \\
\qquad TransmissionFault) \setminus \{fault.*\}
\end{array}
$$

5.4 Mechanical Verification

In this section, we describe how the property of protection, together with FDR, allows us to verify that our specification satisfies its non-interference requirements.

5.4.1 Fault-tolerance

To prove fault-tolerance of our specification, we need to ensure that, for every $t \in Train$, $fault.t$ does not have an adverse affect upon the behaviour of the system as a whole. As $trans2trains$ plays a preventative role in this process, we consider this channel to be a preventative measure.

Our formulation of fault-tolerance with respect to this fault and this preventative measure shall require that the channels $readAspect.t$ and $readPermittedSpeed.t$ are protected from $fault.t$ in the process $FaultyATP \setminus trans2trains$. Thus, we state fault-tolerance in the following way.

Proposition 5.1

$$\forall\, t : Train \bullet \{readAspect.t.*, readPermittedSpeed.t.*\} \nrightarrow \{fault.t\} \; [FaultyATP \setminus trans2trains]$$

Proof. FDR confirms that $FaultyATP \setminus trans2trains$ is deterministic. As

$$\forall\, t : Train \bullet \{fault.t\} \cap (\alpha(FaultyATP \setminus trans2trains)) = \varnothing$$

by Proposition 3.2,

$$\forall\, t : Train \bullet \mathbf{det}(FaultyATP \setminus trans2trains \;|||\; RUN_{\{fault.t\}})$$

By observation,

$$\forall\, t : Train \bullet \alpha(FaultyATP \setminus trans2trains) \subseteq \alpha(FaultyATP \setminus trans2trains)) \cup \{fault.t\} \land \alpha(FaultyATP \setminus trans2trains)) \cap \{fault.t\} = \varnothing$$

By Definition 2.5,

$$\forall\, t : Train \bullet \alpha(FaultyATP \setminus trans2trains) \nLeftarrow \{fault.t\} \; [FaultyATP \setminus trans2trains]$$

By Proposition 2.4,

$$\forall\, t : Train \bullet \alpha(FaultyATP \setminus trans2trains) \nrightarrow \{fault.t\} \; [FaultyATP \setminus trans2trains]$$

By observation,

$$\forall\, t : Train \bullet \{readAspect.t.*, readPermittedSpeed.t.*\} \subseteq \alpha(FaultyATP \setminus trans2trains)$$

Therefore, by Law A.1,

$$\forall\, t : \mathit{Train} \bullet \{\mathit{readAspect}.t.*, \mathit{readPermittedSpeed}.t.*\} \nleftarrow \{\mathit{fault}.t\} \; [\mathit{FaultyATP} \setminus \mathit{trans2trains}]$$

□

5.4.2 Fail-operational

Recall that we wish to ensure that our system fails-operational with respect to a train exceeding either its required speed or its permitted speed. We state this requirement in terms of the following proposition.

Proposition 5.2

$$\forall\, t : \mathit{Train} \bullet \{\mathit{passedSignal}.t, \mathit{trans2trains}.t.*.*\} \nleftarrow \{\underline{\mathit{tooFast}.t.*}\} \; [\mathit{TrainborneInit}(t)]$$

Proof. Consider the set of events

$$\mathit{FailOp}_t = \{\underline{\mathit{tooFast}.t.*}, \underline{\mathit{slowEnough}.t.*}, \mathit{readActualSpeed}.t.*.*, \mathit{readPermittedSpeed}.t.*, \mathit{readRequiredSpeed}.t.*, \mathit{tock}.t.*\}$$

By their definitions,

$$\forall\, t : \mathit{Train} \bullet \alpha\mathit{PermittedSpeedCheck}(t) \subseteq \mathit{FailOp}_t \wedge$$
$$\forall\, t : \mathit{Train} \bullet \alpha\mathit{RequiredSpeedCheck}(t) \subseteq \mathit{FailOp}_t$$

As both $\mathit{PermittedSpeedCheck}(t)$ and $\mathit{RequiredSpeedCheck}(t)$ are nondivergent, by Proposition 3.3,

$$\forall\, t : \mathit{Train} \bullet \mathbf{det}(\mathit{PermittedSpeedCheck}(t) \;|||\; \mathit{RUN}_{\mathit{FailOp}_t}) \wedge$$
$$\forall\, t : \mathit{Train} \bullet \mathbf{det}(\mathit{RequiredSpeedCheck}(t) \;|||\; \mathit{RUN}_{\mathit{FailOp}_t})$$

Therefore, by Definition 2.5,

$$\alpha\mathit{TrainborneInit}(t) - \mathit{FailOp}_t \not\Leftarrow \mathit{FailOp}_t \; [\mathit{PermittedSpeedCheck}(t)] \wedge$$
$$\alpha\mathit{TrainborneInit}(t) - \mathit{FailOp}_t \not\Leftarrow \mathit{FailOp}_t \; [\mathit{RequiredSpeedCheck}(t)]$$

FDR confirms that

$$\forall\, t : \mathit{Train};\; a : \mathit{Aspect};\; s : \mathit{Speed} \bullet \mathbf{det}(\mathit{DataStorage}(t, a, s) \;|||\; \mathit{RUN}_{\mathit{FailOp}_t})$$

Again, by Definition 2.5,

$$\forall\, t : \mathit{Train};\; a : \mathit{Aspect};\; s : \mathit{Speed} \bullet \alpha\mathit{TrainborneInit}(t) - \mathit{FailOp}_t \not\Leftarrow \mathit{FailOp}_t \; [\mathit{DataStorage}(t, a, s)]$$

We can similarly establish

$$\forall\, t : \mathit{Train} \bullet \alpha\mathit{TrainborneInit}(t) - \mathit{FailOp}_t \Leftarrow \mathit{FailOp}_t\ [\mathit{PassBeacons}(t)] \wedge$$
$$\forall\, t : \mathit{Train} \bullet \alpha\mathit{TrainborneInit}(t) - \mathit{FailOp}_t \Leftarrow \mathit{FailOp}_t\ [\mathit{SignalCheck}(t)]$$

By several applications of Laws A.14 and A.17,

$$\forall\, t : \mathit{Train} \bullet \alpha\mathit{TrainborneInit}(t) - \mathit{FailOp}_t \Leftarrow \mathit{FailOp}_t\ [\mathit{Trainborne}(t)]$$

As

$$\{\mathit{location}.* .*, \mathit{initState}.* .*, \mathit{trans2trains}.* .* .*\} \cap \alpha\mathit{FailOp}_t = \varnothing$$

by three applications of Law A.6,

$$\forall\, t : \mathit{Train} \bullet \alpha\mathit{TrainborneInit}(t) - \mathit{FailOp}_t \Leftarrow \mathit{FailOp}_t\ [\mathit{TrainborneInit}(t)]$$

By Proposition 2.4,

$$\forall\, t : \mathit{Train} \bullet \alpha\mathit{TrainborneInit}(t) - \mathit{FailOp}_t \dashv\!\!\!\vdash \mathit{FailOp}_t\ [\mathit{TrainborneInit}(t)]$$

Finally, by Law A.3,

$$\forall\, t : \mathit{Train} \bullet \{\mathit{passedSignal}.t, \mathit{trans2trains}.t.* .*\} \dashv\!\!\!\vdash \{\underline{\mathit{tooFast}.t.*}\}\ [\mathit{TrainborneInit}(t)]$$

□

5.4.3 Fail-safe

A signal being passed at danger is considered a failure, with the corresponding recovery measure being the automatic application of emergency brakes. In this context, given any train $t \in \mathit{Train}$, the events $\mathit{emergencyBrakesOn}.t$, $\mathit{emergencyStop}.t$, and $\mathit{emergencyBrakesOff}.t$ all introduce non-interference. As such, we shall assume that these events all occur successfully, and we abstract the possibility of their occurrence by means of hiding.

Therefore, to prove that this system fails-safe for any train $t \in \mathit{Train}$, we define the set of events

$$\mathit{EMERGENCY}_t = \{\mathit{emergencyBrakesOn}.t, \mathit{emergencyStop}.t, \mathit{emergencyBrakesOff}.t\}$$

together with the process

$$\mathit{Train}'(t) \mathrel{\widehat{=}} \mathit{TrainborneInit}(t) \setminus \mathit{EMERGENCY}_t$$

and prove the following proposition.

Proposition 5.3

$$\forall\, t : \mathit{Train} \bullet$$
$$\alpha\mathit{TrainborneInit}(t) - \{\mathit{readAspect}.t.\mathit{red}\} \dashv\!\!\!\vdash \{\mathit{readAspect}.t.\mathit{red}\}\ [\mathit{Train}'(t)]$$

Proof. Via FDR, we can verify that

$$\forall\, t : Train \bullet \mathbf{det}(Train'(t) \;|||\; RUN_{\{readAspect.t.red\}})$$

By observation,

$$\forall\, t : Train \bullet$$
$$\alpha Train'(t) \subseteq (\alpha TrainborneInit(t) - \{readAspect.t.red\}) \cup \{readAspect.t.red\} \wedge$$
$$(\alpha TrainborneInit(t) - \{readAspect.t.red\}) \cap \{readAspect.t.red\} = \varnothing$$

Therefore, by Definition 2.5 and Proposition 2.4,

$$\forall\, t : Train \bullet$$
$$\alpha TrainborneInit(t) - \{readAspect.t.red\} \nleftarrow \{readAspect.t.red\}\ [Train'(t)]$$

□

5.4.4 A Safety Core

We state our safety core property for this specification in terms of the following proposition.

Proposition 5.4

$$\forall\, t : Train;\ a : Aspect;\ s : Speed \bullet$$
$$\{trans2trains.t.\ *.*\} \nleftarrow \alpha Trainborne(t, a, s) - \{trans2trains.t.\ *.*\}\ [Trainborne(t, a, s)]$$

Proof. By observation,

$$\forall\, t : Train;\ a : Aspect;\ s : Speed \bullet$$
$$\alpha DataStorage(t, a, s) \subseteq$$
$$\{trans2trains.t.\ *.*\} \cup (\alpha Trainborne(t, a, s) - \{trans2trains.t.\ *.*\})$$
$$\wedge\ \{trans2trains.t.\ *.*\} \cap (\alpha Trainborne(t, a, s) - \{trans2trains.t.\ *.*\}) = \varnothing$$

Trivially,

$$\forall\, t : Train;\ a : Aspect;\ s : Speed \bullet$$
$$\mathbf{det}(DataStorage(t, a, s) \;|||\; RUN_{\alpha Trainborne(t,a,s) - \{trans2trains.t.*.*\}})$$

By Definition 2.5,

$$\forall\, t : Train;\ a : Aspect;\ s : Speed \bullet$$
$$\{trans2trains.t.\ *.*\} \nleftarrow\!\!\!\!\!\!= \alpha Trainborne(t, a, s) - \{trans2trains.t.\ *.*\}\ [DataStorage(t, a, s)]$$

By their definitions,

$$\forall\, t : Train;\ a : Aspect;\ s : Speed \bullet$$
$$\alpha SignalCheck(t) \subseteq \alpha Trainborne(t, a, s) - \{trans2trains.t.\ *.*\} \wedge$$
$$\alpha PermittedSpeedCheck(t) \subseteq \alpha Trainborne(t, a, s) - \{trans2trains.t.\ *.*\} \wedge$$
$$\alpha RequiredSpeedCheck(t) \subseteq \alpha Trainborne(t, a, s) - \{trans2trains.t.\ *.*\} \wedge$$
$$\alpha PassBeacons(t) \subseteq \alpha Trainborne(t, a, s) - \{trans2trains.t.\ *.*\}$$

As all of these processes are nondivergent, by Proposition 3.3,

$$\begin{aligned}
&\forall\, t : \mathit{Train};\ a : \mathit{Aspect};\ s : \mathit{Speed} \bullet \\
&\quad \{\mathit{trans2trains}.t.* .*\} \not\Vdash \alpha\mathit{Trainborne}(t,a,s) - \{\mathit{trans2trains}.t.* .*\} \\
&\qquad\qquad [\mathit{SignalCheck}(t)] \\
&\quad \wedge\ \{\mathit{trans2trains}.t.* .*\} \not\Vdash \alpha\mathit{Trainborne}(t,a,s) - \{\mathit{trans2trains}.t.* .*\} \\
&\qquad\qquad [\mathit{PermittedSpeedCheck}(t)] \\
&\quad \wedge\ \{\mathit{trans2trains}.t.* .*\} \not\Vdash \alpha\mathit{Trainborne}(t,a,s) - \{\mathit{trans2trains}.t.* .*\} \\
&\qquad\qquad [\mathit{RequiredSpeedCheck}(t)] \\
&\quad \wedge\ \{\mathit{trans2trains}.t.* .*\} \not\Vdash \alpha\mathit{Trainborne}(t,a,s) - \{\mathit{trans2trains}.t.* .*\} \\
&\qquad\qquad [\mathit{PassBeacons}(t)]
\end{aligned}$$

By Laws A.14 and A.17,

$$\begin{aligned}
&\forall\, t : \mathit{Train};\ a : \mathit{Aspect};\ s : \mathit{Speed} \bullet \\
&\{\mathit{trans2trains}.t.* .*\} \not\Vdash \alpha\mathit{Trainborne}(t,a,s) - \{\mathit{trans2trains}.t.* .*\} \\
&\qquad\qquad [\mathit{Trainborne}(t,a,s)]
\end{aligned}$$

By Proposition 2.4,

$$\begin{aligned}
&\forall\, t : \mathit{Train};\ a : \mathit{Aspect};\ s : \mathit{Speed} \bullet \\
&\{\mathit{trans2trains}.t.* .*\} \not\vdash \alpha\mathit{Trainborne}(t,a,s) - \{\mathit{trans2trains}.t.* .*\} \\
&\qquad\qquad [\mathit{Trainborne}(t,a,s)]
\end{aligned}$$

□

Note that this property does not hold in $\mathit{TrainborneInit}(t)$, because the relevant communications are not available initially.

5.4.5 Firewalling

Given our definition of our firewalling property, we require the following to hold our specification.

$$\forall\, t, t' : \mathit{Train} \mid t \neq t' \bullet \alpha\mathit{TrainborneInit}(t) \not\vdash \alpha\mathit{TrainborneInit}(t')\ [\mathit{ATP}]$$

Unfortunately, this can never hold of our process ATP because the track-to-train transmission system has a maximum buffer size of three messages. This means that a request for data from one train $t \in \mathit{Train}$ may have to be met before another train $t' \in \mathit{Train}$ can be dealt with, thus affecting the latter's interface. However, if we define the process $\mathit{ATP}2$ to be ATP with the restriction of the buffer limit removed in the process *Transmission*, which we call *Transmission*2, then we can prove the following.

Proposition 5.5

$$\forall\, t, t' : \mathit{Train} \mid t \neq t' \bullet \alpha\mathit{TrainborneInit}(t) \not\vdash \alpha\mathit{TrainborneInit}(t')\ [\mathit{ATP}2]$$

Proof. By observation,

$$\forall\, t : Train \bullet \\ \alpha TrainborneInit(t) \subseteq \alpha TrainborneInit(t) \cup (\alpha ATP2 - \alpha TrainborneInit(t)) \land \\ \alpha TrainborneInit(t) \cap (\alpha ATP2 - \alpha TrainborneInit(t)) = \varnothing$$

As $TrainborneInit(t)$ is deterministic, by Proposition 3.2,

$$\forall\, t : Train \bullet \mathbf{det}(TrainborneInit(t) \;|||\; RUN_{\alpha ATP2 - \alpha TrainborneInit(t)})$$

By Definition 2.5,

$$\forall\, t : Train \bullet \alpha TrainborneInit(t) \Leftarrow\!\!= \alpha ATP2 - \alpha TrainborneInit(t) \\ [TrainborneInit(t)]$$

By Proposition 3.3,

$$\forall\, t, t : Train \mid t \neq t' \bullet \mathbf{det}(TrainborneInit(t') \;|||\; RUN_{\alpha ATP2 - \alpha TrainborneInit(t)})$$

By Definition 2.5,

$$\forall\, t, t : Train \mid t \neq t' \bullet \alpha TrainborneInit(t) \Leftarrow\!\!= \alpha ATP2 - \alpha TrainborneInit(t) \\ [TrainborneInit(t')]$$

By Law A.14,

$$\forall\, t : Train \bullet \alpha TrainborneInit(t) \Leftarrow\!\!= \alpha ATP2 - \alpha TrainborneInit(t)\ [Trains]$$

We can show that

$$\forall\, t : Train \bullet \alpha TrainborneInit(t) \Leftarrow\!\!= \alpha ATP2 - \alpha TrainborneInit(t)\ [Transmission2] \\ \land \\ \forall\, t : Train \bullet \alpha TrainborneInit(t) \Leftarrow\!\!= \alpha ATP2 - \alpha TrainborneInit(t)\ [Wayside]$$

Therefore, by Laws A.14 and A.17,

$$\forall\, t : Train \bullet \alpha TrainborneInit(t) \Leftarrow\!\!= \alpha ATP2 - \alpha TrainborneInit(t)\ [ATP2]$$

Finally, by Law A.2,

$$\forall\, t, t' : Train \mid t \neq t' \bullet \alpha TrainborneInit(t) \leftarrow\!\!\!+ \alpha TrainborneInit(t')\ [ATP2]$$

□

5.5 Conclusions

We have seen how a number of properties of a specification of a simplified safety-critical system can be defined in terms of non-interference, and how such properties can be mechanically verified with FDR. In the next chapter, we apply FDR to the analysis of a specific form of non-interference — that of mutual exclusion.

Chapter 6

Mutual Exclusion

Mutual exclusion is one of the fundamental problems of concurrent computing. In the base case, the mutual exclusion problem is stated thus: Given two programs P_1 and P_2 such that both have 'critical regions', can we guarantee that no more than one of these programs will ever be in its critical region at any time? That is, can we guarantee that the critical regions are *mutually exclusive*? This requirement can be enforced by the application of a *mutual exclusion algorithm*. In addition to guaranteeing mutual exclusion, such algorithms should be fair, i.e., if a program wishes to enter its critical section, then it cannot repeatedly be denied entry while the other continues to enter and leave its critical section. It should also be free from the possibility of livelock (which we term *starvation*) and deadlock.

Some solutions to the mutual exclusion problem involve multiple resources, while others rely upon some form of hardware assistance. Some solutions are more efficient than others, while some are more elegant. Some are unsatisfactory: they might enforce mutual exclusion, but they can lead to deadlock or starvation. Furthermore, some 'solutions' do not even enforce mutual exclusion. We do not aim to present a novel solution to the mutual exclusion problem; rather, we illustrate how CSP may be used to model the behaviour of distributed processes and programming constructs for this form of non-interference.

CCS interpretations of a number of mutual exclusion algorithms are described in [Wal88, Wal89a], with automated analysis provided by the Concurrency Workbench [CPS89a, CPS89b]. As such, further goals of this chapter are to illustrate how FDR may be used to explore and prove properties of distributed algorithms, and to investigate how our approach compares with that of the Concurrency Workbench. We also investigate how certain algorithms fair in the absence of atomicity — an assumption upon which the correctness of many solutions to the mutual exclusion problem rely heavily. Furthermore, we investigate the problem of fault-tolerant mutual exclusion.

Work described in this chapter has previously appeared as [DS96] and [Sim96].

6.1 Mutual Exclusion and Non-interference

The mutual exclusion problem is one of non-interference — if we can guarantee that no more than one program is in its critical section at any moment in time and that our mutual exclusion algorithm is fair, then at every stage of the system's execution, each program's

critical section is non-interfering with those of the others.

Perhaps the clearest parallel between non-interference and mutual exclusion is provided by non-interference in its guise of safety-critical transaction non-interference. In this sense, the possible behaviour of the system's non-critical events following the execution of one program's critical region should be identical to what it was before the critical region was entered. That is, guaranteeing mutual exclusion can be thought of as demonstrating that the transactions representing critical regions are non-interfering with the rest of the system. For example, a critical region may involve non-critical events, as well as those which are categorised as critical. As such, it is not non-interference from critical events that we are concerned with, but non-interference from specific transactions which are so categorised.

6.2 The Mutual Exclusion Problem

In the general case, the mutual exclusion problem is concerned with n programs ($n \geq 2$), $P_1, \ldots, P_n$, whose access to some shared resource takes place only within a critical region. Any algorithm proposing to enforce mutual exclusion algorithm should satisfy the following conditions.

- **Safety.** No more than one of the programs may be inside its critical region at any time.
- **Liveness.** The algorithm should not introduce deadlocks. If a program halts outside its critical region, then other programs should not be prevented from proceeding.
- **Fairness.** If two programs are waiting to enter their critical regions, then it should be impossible for one program to be granted access infinitely many times in preference to the other.
- **Freedom from starvation.** If at least one program wishes to enter its critical region, then it should not be possible for all programs to remain in their non-critical regions *ad infinitum.*

Note that we do not insist that the system never deadlocks — we insist only that a mutual exclusion algorithm must not introduce deadlocks where they are unnecessary.

6.3 A CSP Approach to Mutual Exclusion

Given the base case of 2 programs, we shall write mutual exclusion algorithms in the following form.

```
{P1}
  WHILE true DO
    BEGIN
      ...
      ...
      <critical region 1>;
      ...
      ...
    END;
```

```
{P2}
  WHILE true DO
    BEGIN
      ...
      ...
      <critical region 2>;
      ...
      ...
    END;
```

6.3.1 Critical Regions

Despite the fact that we do not concern ourselves with their details, we denote the critical regions for programs P1 and P2 by `<critical region 1>` and `<critical region 2>` respectively.

In the general case, we assume a set

$$Id = \{1, \ldots, n\}$$

such that mutual exclusion should hold with respect to programs $P_1, \ldots, P_n$. We mark the entry and exit from a program's critical region by communications on the channels *enter* and *leave*.

$$enter = \{i : Id \bullet enter.i\}$$
$$leave = \{i : Id \bullet leave.i\}$$

For each $i \in Id$, we define the set of critical events of program P_i by

$$critical_i = \{enter.i, leave.i\}$$

Furthermore, the set of all critical events are given by

$$Critical = \bigcup_{i \in Id} critical_i$$

We say that any events not contained in this set are non-critical.

We also define the free type *Bool*.

$$Bool ::= true \mid false$$

6.3.2 Dealing with Variables

Atomicity

All mutual exclusion algorithms involve the reading of, and writing to, variables. To guarantee the required properties, some mutual exclusion algorithms make the assumption that variable access is atomic. In reality, such assumptions cannot be enforced. Therefore, in the following, we make no such assumptions about variable access.

Flag variables

In several of the algorithms described in the following, there is a *flag variable* associated with each program. A flag may be in one of two states — true or false — at any time. In general, if a program's flag is true, then that program wishes to enter its critical region. It is often the case that a program can write to its own flag only. However, we shall not impose this assumption via our modelling of variables — this will be enforced by the mutual exclusion algorithms themselves. Furthermore, any program can read the value of any flag.

We introduce the following sets of communications accordingly.

$$writeFlag = \{i, j : Id;\ b : Bool \bullet writeFlag.i.j.b\}$$
$$readFlag = \{i, j : Id;\ b : Bool \bullet readFlag.i.j.b\}$$

Here, $writeFlag.i.j.b$ represents the action of program P_i writing the value b to program P_j's flag, and $readFlag.i.j.b$ represents the action of program P_i reading the value b of program P_j's flag.

Given the above, we represent the storage of program P_i's flag by the process *Flag*. This is defined thus:

$$\begin{aligned} Flag(i, b) \mathrel{\hat{=}}\ & writeFlag?p.i?c \rightarrow Flag(i, c) \\ & \Box \\ & readFlag?p.i!b \rightarrow Flag(i, b) \end{aligned}$$

Here, the value of P_i's flag can always be read or written to by any program.

If we were to assume that each flag has the initial value *false*, then we might define the process *FlagsInit* to represent the initial state of our overall view of the flag variables.

$$FlagsInit \mathrel{\hat{=}} Flag(1, false) \ |||\ Flag(2, false)$$

Turn variables

In several solutions to the mutual exclusion problem, there is a variable which represents the notion of some program having the next *turn*. This provides a means of alternating priority between the programs. Again, as far as our CSP representation is concerned, any program is permitted to write to, or read from, this variable. We start by introducing the channels *writeTurn* and *readTurn*.

$$writeTurn = \{i, j : Id \bullet writeTurn.i.j\}$$
$$readTurn = \{i, j : Id \bullet readTurn.i.j\}$$

Here, $writeTurn.i.j$ represents the action of program P_i writing the value j to the turn variable, and $readTurn.i.j$ represents the action of program P_i reading the value j of the turn variable.

We define the process *Turn* accordingly.

$$\begin{aligned} Turn(i) \mathrel{\hat{=}}\ & writeTurn?p?j \rightarrow Turn(j) \\ & \Box \\ & readTurn?p!i \rightarrow Turn(i) \end{aligned}$$

If we were to assume that the initial value of *Turn* is 1, then we might define

$$TurnInit \mathrel{\hat{=}} Turn(1)$$

We shall typically define the overall view of variable storage by a process of the form

$$Vars \mathrel{\hat{=}} FlagsInit \ |||\ TurnInit$$

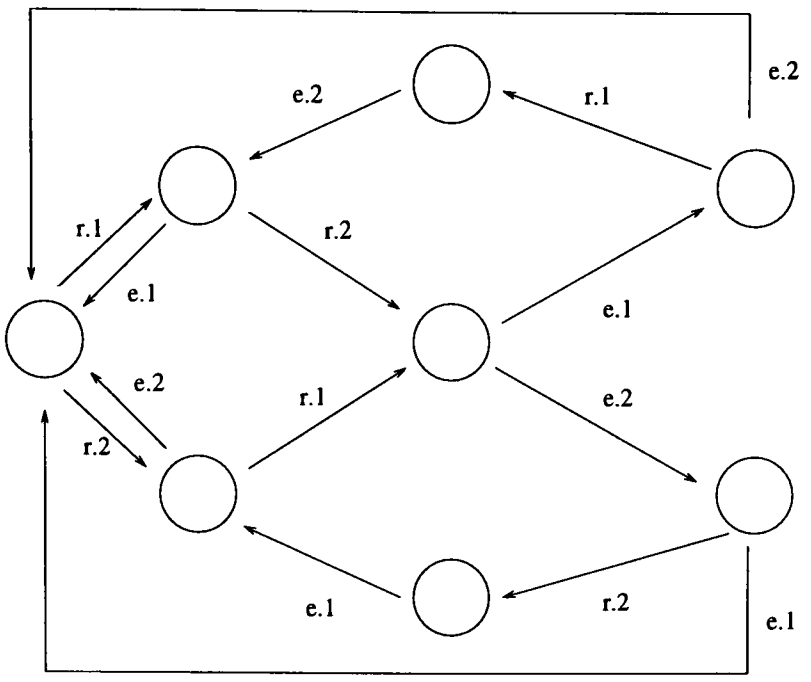

Figure 6.1: Fairness.

6.3.3 Mutual Exclusion

Given the channels *enter* and *leave*, we define our mutual exclusion requirement in terms of the process *MutualExclusionCheck*.

$$MutualExclusionCheck \mathrel{\widehat{=}} enter?i \rightarrow leave.i \rightarrow MutualExclusionCheck$$

Here, if program P_i enters its critical region, then it must perform a *leave.i* before another program can enter its critical region.

We define the satisfaction of mutual exclusion in the following way.

Definition 6.1 A mutual exclusion algorithm *Alg* is *safe* if and only if

$$Alg \textbf{ imp } MutualExclusionCheck$$

□

6.3.4 Liveness

Any process which is a refinement of $CHAOS_\Sigma$ will always be willing to participate in at least one event — it can never deadlock. As such, we define liveness in the following way.

Definition 6.2 A mutual exclusion algorithm *Alg* is *live* if and only if

$$CHAOS_\Sigma \sqsubseteq Alg$$

□

6.3.5 Fairness

In the following CSP descriptions of mutual exclusion algorithms, it will be apparent that the moment at which a program wishes to engage in critical activity is represented by a unique event. We shall represent the observation that a program wishes to engage in its critical activity by a communication on the channel *ready*.

$$ready = \{i : Id \bullet ready.i\}$$

Assuming for the moment $n = 2$, we introduce the process *FairnessCheck*, which corresponds to the state machine of Figure 6.1, in which $r.1$ represents the event $ready.1$, $e.1$ represents the event $enter.1$, etc.

Initially, either program may perform a *ready*.

$$\begin{aligned} FairnessCheck \mathrel{\hat{=}} & \; ready.1 \rightarrow FairnessCheck_1 \\ & \Box \\ & \; ready.2 \rightarrow FairnessCheck_2 \end{aligned}$$

Note that the subscript 1 denotes the state of this requirement when program P_1 is ready to enter its critical region, the subscript 2 denotes the fact that program P_2 is ready, and 12 is representative of the fact that both wish to enter.

If program P_1 has performed $ready.1$, then either $enter.1$ or $ready.2$ may be performed next. If it is the former, then we return to the initial state; if it is the latter, then both programs are ready to enter their critical regions.

$$\begin{aligned} FairnessCheck_1 \mathrel{\hat{=}} & \; enter.1 \rightarrow FairnessCheck \\ & \Box \\ & \; ready.2 \rightarrow FairnessCheck_{12} \end{aligned}$$

The process $Fairness_2$ is the complement of $Fairness_1$.

$$\begin{aligned} FairnessCheck_2 \mathrel{\hat{=}} & \; ready.1 \rightarrow FairnessCheck_{12} \\ & \Box \\ & \; enter.2 \rightarrow FairnessCheck \end{aligned}$$

Finally, if both programs have performed a *ready*, then we need to ensure that neither P_1 nor P_2 can enter its critical region twice without the other one entering its critical region.

$$\begin{aligned} FairnessCheck_{12} \mathrel{\hat{=}} & \; enter.1 \rightarrow \; enter.2 \rightarrow FairnessCheck \\ & \qquad\qquad\quad \Box \\ & \qquad\qquad\quad ready.1 \rightarrow enter.2 \rightarrow FairnessCheck_1 \\ & \Box \\ & \; enter.2 \rightarrow \; enter.1 \rightarrow FairnessCheck \\ & \qquad\qquad\quad \Box \\ & \qquad\qquad\quad ready.2 \rightarrow enter.1 \rightarrow FairnessCheck_2 \end{aligned}$$

The above guarantees that if both programs wish to enter their critical regions, then there cannot be two consecutive *enter*s performed by the same program without the other one

being able to enter its critical region. The justification for this formulation is that if one program cannot be denied entry twice, then it cannot be denied entry infinitely many times. Of course, this process can easily be generalised to the case of n programs.

Given the above process, we define fairness in the following way.

Definition 6.3 A mutual exclusion algorithm *Alg* is fair if and only if

$$Alg \textbf{ imp } FairnessCheck$$

□

6.3.6 Starvation

A mutual exclusion algorithm *Alg* introduces starvation if it is possible for there to be an infinite sequence of non-critical events, thus denying all programs the possibility of entry into their critical regions. As such, we define starvation-freedom in the following way.

Definition 6.4 A mutual exclusion algorithm *Alg* is *starvation-free* if and only if

$$\textbf{nondiv}(Alg \setminus (\alpha Alg - Critical))$$

□

6.4 The Mechanical Verification of Mutual Exclusion Algorithms

The first published solution to the mutual exclusion problem was that of Dijkstra [Dij65].[1] Other proposed solutions which we investigate in this section are those of Dekker (with our version being taken from [PS85]), Hyman [Hym66], Knuth [Knu66], Peterson [Pet81], and Lamport [Lam86]. CCS interpretations of these algorithms are given in [Wal88, Wal89b], with automated analysis provided by the Concurrency Workbench [CPS89a, CPS89b]. In this section, we describe CSP translations of these CCS agents, with automated analysis being provided by FDR.

6.4.1 Dijkstra's Algorithm

The first published solution to the mutual exclusion problem was Dijkstra's algorithm, which was presented in [Dij65]. The version given in Table 6.1 is taken from [Wal88].

This algorithm involves two arrays of Boolean variables, b and c, and a further variable, k, which is of type *Id*. All Boolean variables are initially *true*, while k can take any initial value.[2] The behaviour of program P_i, for any $i \in Id$, is as follows.

The processes representing the variables b and c are defined in terms of process *Flag* of Section 6.3.2, while the process representing variable k is defined in terms of process *Turn*.

We have seen many times during the course of this thesis that conditionals can be modelled in CSP by processes of the form

$$\textbf{if } b \textbf{ then } P \textbf{ else } Q$$

[1] See [Ray86] and [Jac95] for collections of mutual exclusion algorithms.

[2] However, for the sake of completeness, we choose the initial value 1.

```
VAR b : array [1..n] of Boolean
FOR i := 1 TO n DO b[i] := true;

VAR c : array [1..n] of Boolean
FOR i := 1 TO n DO c[i] := true;

VAR k : integer
k := 1;

{Pi}
VAR j : integer;
WHILE true DO
  BEGIN
    b[i] := false;
    L1: IF k <> i THEN
      BEGIN
        c[i] := true;
        IF b[k] THEN k := i;
        GOTO L1
      END
    ELSE
      BEGIN
        c[i] := false;
        FOR j := 1 TO n DO
          IF (j <> i AND c[j] = false) THEN GOTO L1
      END;
    <critical region i>;
    c[i] := true;
    b[i] := true
END;
```

Table 6.1: Dijkstra's algorithm.

We now consider how assignment and for-loops, which both play an important role in the following, can be modelled in terms of CSP processes.

In Dijkstra's algorithm, the first action program `Pi` performs when it wishes to enter its critical section is the assignment

```
b[i] := false
```

We represent such an assignment in our mutual exclusion model by

$$writeB.i.i!false \rightarrow Skip$$

In addition, Dijkstra's algorithm involves the for-loop

```
FOR j := 1 TO n DO
  IF (j <> i AND c[j] = false) THEN GOTO L1
```

In the general case, a for-loop of the form

```
FOR count := 1 TO m DO Body
```

can be represented by the CSP process $ForLoop(1, m)$, where

$$\begin{aligned} ForLoop(count, m) \mathrel{\hat{=}} \textbf{if}\ & count > m \\ \textbf{then}\ & Skip \\ \textbf{else}\ & Body \fatsemi ForLoop(count + 1, m) \end{aligned}$$

It follows that our CSP interpretation of the above algorithm, for any program `Pi`, is given by the process $Dijkstra_0(i)$.

Initially, if program P_i wishes to enter its critical section, then it writes the value of $b[i]$ to *false*. The process then behaves as $Dijkstra_1$, which corresponds to line `L1` of the algorithm.

$$Dijkstra_0(i) \mathrel{\hat{=}} ready.i \rightarrow writeB.i.i!false \rightarrow Dijkstra_1(i)$$

The process $Dijkstra_1$ is defined thus:

$$\begin{aligned} Dijkstra_1(i) \mathrel{\hat{=}} readK.i?j \rightarrow \textbf{if}\ & j = i \\ \textbf{then}\ & writeC.i.i!false \rightarrow ForLoop(i, 1) \\ \textbf{else}\ & Dijkstra_2(i) \end{aligned}$$

Here, the value of k is checked: if $k = i$, then the value of $c[i]$ is set to *false*, and a for-loop is entered; if $k \neq i$, then the process $Dijkstra_2$ is invoked.

$$\begin{aligned} Dijkstra_2(i) \mathrel{\hat{=}}\ & writeC.i.i!true \rightarrow \\ & readK.i?x \rightarrow readB.i.x.true \rightarrow writeK.i!i \rightarrow Dijkstra_1(i) \\ & \Box \\ & readB.i.x.false \rightarrow Dijkstra_1(i) \end{aligned}$$

$$
\begin{array}{l}
ForLoop(i, count) \mathrel{\widehat{=}} \\
\quad \textbf{if} \quad count > n \\
\quad \textbf{then} \; Critical(i) \\
\quad \textbf{else} \; \textbf{if} \quad count \neq i \\
\qquad \textbf{then} \; readC.i.count.false \rightarrow Dijkstra_1(i) \\
\qquad\qquad \Box \\
\qquad\qquad readC.i.count.true \rightarrow ForLoop(i, count + 1) \\
\qquad \textbf{else} \; ForLoop(i, count + 1)
\end{array}
$$

Finally, after program P_i has left its critical region, the values of both $c[i]$ and $b[i]$ are written to *true*.

$$
\begin{array}{l}
Critical(i) \mathrel{\widehat{=}} enter.i \rightarrow leave.i \rightarrow \\
\qquad\qquad writeC.i.i!true \rightarrow writeB.i.i!true \rightarrow Dijkstra_0(i)
\end{array}
$$

We define

$$
DijkstraAlg \mathrel{\widehat{=}} DijkstraVars \; [\![\{ \begin{array}{l} readK.*.*, readB.*.*.*, readC.*.*.*, \\ writeK.*.*, writeB.*.*.*, writeC.*.*.* \} \end{array}]\!] \; |||_{i \in Id} Dijkstra_0(i)
$$

such that *DijkstraVars* is defined thus:

$$
DijkstraVars \mathrel{\widehat{=}} K(1) \; ||| \; |||_{i \in Id} (B(i, true) \; ||| \; C(i, true))
$$

Via FDR, it can be verified in a matter of seconds (the process representing the algorithm for $n = 2$ composed with processes representing the storage of variables consists of only 254 states) that this algorithm is safe, live, and starvation-free. However, it is not fair.

As an illustration of how this process fails to be fair, consider the following behaviour. Initially, program P_1 writes `b[1]` to false, and continues its execution uninterrupted until it performs *readC*.1.2.*true*. At this point, program P_2 performs *writeB*.2.2.*false*. This is followed by P_1 entering and leaving its critical region. Following the algorithm around again, it is possible for program P_1 to enter its critical region again without an observation of *enter*.2. Therefore, we conclude that this algorithm is not fair.

6.4.2 Dekker's Algorithm

The algorithm described in this section is due to Thomas Dekker (see [PS85]), and the formulation (for the case of $n = 2$) given in Table 6.2 is a correction of that described in [Jac95]. This algorithm involves two flag variables, both of which have initial value *false*, and a turn variable which has an arbitrary initial value.[3]

This algorithm involves a number of 'simple' while-loops, i.e., loops of the form

```
WHILE flag2 DO nothing
```

[3] Again, for the sake of completeness, we choose the initial value 1.

```
VAR flag1, flag2 : Boolean
flag1 := false; flag2 := false;

VAR turn : Id
turn := 1;
```

```
{P1}
  WHILE true DO
    BEGIN
      flag1 := true;
      IF turn = 1 THEN
        WHILE flag2 DO
          nothing
      ELSE
        BEGIN
          flag1 := false;
          WHILE turn = 2 DO
            nothing;
          flag1 := true;
          WHILE flag2 DO
            nothing
        END;
    <critical region 1>;
    turn := 2;
    flag1 := false
END
```

```
{P2}
  WHILE true DO
    BEGIN
      flag2 := true;
      IF turn = 2 THEN
        WHILE flag1 DO
          nothing
      ELSE
        BEGIN
          flag2 := false;
          WHILE turn = 1 DO
            nothing;
          flag2 := true;
          WHILE flag1 DO
            nothing
        END;
    <critical region 2>;
    turn := 1;
    flag2 := false
END
```

Table 6.2: Dekker's algorithm.

As observations of the true case for such conditionals are of no consequence to us, we choose to model such loops by processes of the form

$$readFlag.1.2.false \rightarrow Skip$$

It follows that Dekker's algorithm can be coded in CSP in the following way.

As $n = 2$, each program needs to keep track of the value of the 'other' program — j represents this value.

Initially, program P_i writes its flag to *true*.

$$Dekker(i,j) \mathrel{\hat{=}} ready.i \rightarrow writeFlag.i.i!true \rightarrow CheckTurn(i,j)$$

Next, the value of the turn variable is checked. If it is program P_i's turn, then it waits for program P_j's flag to become *false* (represented by the process *WaitFlag*) before entering its critical region. Alternatively, the process *WaitTurn* is invoked.

$$\begin{aligned} CheckTurn(i,j) \mathrel{\hat{=}}\ & readTurn.i.i \rightarrow WaitFlag(i,j) \\ & \Box \\ & readTurn.i.j \rightarrow WaitTurn(i,j) \end{aligned}$$

The processes *WaitFlag* and *WaitTurn* are defined as follows.

$$WaitFlag(i,j) \mathrel{\hat{=}} readFlag.i.j.false \rightarrow Critical(i,j)$$

$$\begin{aligned} WaitTurn(i,j) \mathrel{\hat{=}}\ & writeFlag.i.i!false \rightarrow readTurn.i.i \rightarrow \\ & writeFlag.i.i!true \rightarrow readFlag.i.j.false \rightarrow Critical(i,j) \end{aligned}$$

Following an exit from the critical region, the turn and flag variables are updated accordingly.

$$\begin{aligned} Critical(i,j) \mathrel{\hat{=}}\ & enter.i \rightarrow leave.i \rightarrow \\ & writeTurn.i!j \rightarrow writeFlag.i.i!false \rightarrow Dekker(i,j) \end{aligned}$$

It can be verified easily that this algorithm is safe, fair, live and starvation-free, with the process in question consisting of only 84 states for the case of $n = 2$.

6.4.3 Hyman's Algorithm

Hyman's algorithm [Hym66] was proposed as a simplification of Dijkstra's algorithm. Unfortunately, it is something of an over-simplification, as it fails to guarantee mutual exclusion. The version given in Table 6.3 is adapted from that of [Wal89b], which in turn was taken from [PS85].

Hyman's algorithm for program P1 includes the while-loop

```
WHILE turn <> 1 DO
  BEGIN
    ...
  END
```

```
VAR flag1, flag2 : Boolean
flag1 := false; flag2 := false;

VAR turn : Id
turn := 1;
```

```
{P1}
 WHILE true DO
  BEGIN
   flag1 := true;
   WHILE turn <> 1 DO
    BEGIN
     WHILE flag2 DO
      nothing;
     turn := 1
    END;
   <critical region 1>;
   flag1 := false
  END;
```

```
{P2}
 WHILE true DO
  BEGIN
   flag2 := true;
   WHILE turn <> 2 DO
    BEGIN
     WHILE flag1 DO
      nothing;
     turn := 2
    END;
   <critical region 2>;
   flag2 := false
  END;
```

Table 6.3: Hynan's algorithm.

In the general case, a while-loop is written

```
WHILE turn <> 1 DO Body
```

Such a loop can be represented by a CSP process of the form

$$\begin{aligned} WhileLoop \mathrel{\widehat{=}} & \; readTurn.1.1 \rightarrow Skip \\ & \Box \\ & readTurn.1.2 \rightarrow Body \fatsemi WhileLoop \end{aligned}$$

We write our CSP interpretation of this algorithm for the case of $n = 2$ in terms of the process *Hyman*.

Initially, program P_i writes its flag to *true*.

$$Hyman(i,j) \mathrel{\widehat{=}} ready.i \rightarrow writeFlag.i.i!true \rightarrow Wait(i,j)$$

The process $Wait(i,j)$ reads the value of the turn variable: if it is program P_j's turn, then the process $WaitFlag(i,j)$ is invoked; if it is program P_i's turn, then that program enters its critical region.

$$\begin{aligned} Wait(i,j) \mathrel{\widehat{=}} & \; readTurn.i.j \rightarrow WaitFlag(i,j) \\ & \Box \\ & readTurn.i.i \rightarrow Critical(i,j) \end{aligned}$$

The process $WaitFlag(i,j)$ waits for program P_j's flag to become *false*, at which point it sets

Program 1	Program 2
	ready.2
	writeFlag.2.2.*true*
	readTurn.2.1
	readFlag.2.1.*false*
ready.1	
writeFlag.1.1.*true*	
readTurn.1.1	
	writeTurn.2.2
	readTurn.2.2
enter.1	
	enter.2

Table 6.4: An unsafe behaviour of Hyman's algorithm.

the value of the turn variable to i, and returns to the state described by process $Wait(i,j)$.

$$WaitFlag(i,j) \mathrel{\hat{=}} readFlag.i.j.false \rightarrow writeTurn.i!i \rightarrow Wait(i,j)$$

After executing its critical region, program P_i writes the value of its flag to *false*, and returns to its initial state.

$$Critical(i,j) \mathrel{\hat{=}} enter.i \rightarrow leave.i \rightarrow writeFlag.i.i!false \rightarrow Hyman(i,j)$$

FDR confirms that the above algorithm is not safe. As an illustration, a possible behaviour of this process which doesn't satisfy our mutual exclusion requirement is given in Table 6.4.

6.4.4 Knuth's Algorithm

Knuth's algorithm [Knu66] was suggested as an alternative to Hyman's, and successfully built upon that of Dijkstra, in the sense that it guarantees both mutual exclusion and fairness. The version given in Table 6.5 is taken from [Wal89b]. The algorithm involves two variables, $c1$ and $c2$, which can take the value 0, 1 or 2, and a further variable *turn*, which may take any value $i \in Id$. The initial values of $c1$ and $c2$ are both 0, while the initial value of turn is arbitrary. We again consider the special case of $n = 2$.

We model variables in the usual way, while our CSP representation of the algorithm is as follows. Note that processes $Knuth_0$, $Knuth_1$ and $Knuth_2$ correspond to lines L0, L1 and L2 of the algorithm.

$$Knuth(i,j) \mathrel{\hat{=}} ready.i \rightarrow Knuth_0(i,j)$$

Initially, program P_i sets the value of $c[i]$ to 1.

$$Knuth_0(i,j) \mathrel{\hat{=}} writeC.i.i!1 \rightarrow Knuth_1(i,j)$$

```
VAR c1, c2 : {0,1,2}
c1 := 0; c2 := 0;

VAR turn : Id
turn := 1;

{P1}
 WHILE true DO
   BEGIN
     L0: c1 := 1;
     L1: IF turn = 1 THEN
       GOTO L2;
     IF c2 <> 0 THEN
       GOTO L1;
     L2: c1 := 2;
     IF c2 = 2 THEN
       GOTO L0;
     turn := 1;
     <critical region 1>;
     turn := 2;
     c1 := 0
   END;
```

```
{P2}
  WHILE true DO
    BEGIN
      L0: c2 := 1;
      L1: IF turn = 2 THEN
        GOTO L2;
      IF c1 <> 0 THEN
        GOTO L1;
      L2: c2 := 2;
      IF c1 = 2 THEN
        GOTO L0;
      turn := 2;
      <critical region 2>;
      turn := 1;
      c2 := 0
    END;
```

Table 6.5: Knuth's algorithm.

If it is program P_i's turn, then $Knuth_2$ is invoked. Alternatively, the value of $c[j]$ is checked, which results in either $Knuth_1$ or $Knuth_2$.

$$
\begin{aligned}
Knuth_1(i,j) \mathrel{\hat{=}} \; & readTurn.i.i \rightarrow Knuth_2(i,j) \\
& \Box \\
& readTurn.i.j \rightarrow readC.i.j.0 \rightarrow Knuth_2(i,j) \\
& \qquad \Box \\
& \qquad readC.i.j.1 \rightarrow Knuth_1(i,j) \\
& \qquad \Box \\
& \qquad readC.i.j.2 \rightarrow Knuth_1(i,j)
\end{aligned}
$$

After the value of $c[i]$ has been set to 2, the value of $c[j]$ is checked. If it is 0 or 1, then program P_i is free to enter its critical region; otherwise the program returns to its initial state.

$$
\begin{aligned}
Knuth_2(i,j) \mathrel{\hat{=}} \; & writeC.i.i!2 \rightarrow readC.i.j.0 \rightarrow writeTurn.i!i \rightarrow Critical(i,j) \\
& \qquad \Box \\
& \qquad readC.i.j.1 \rightarrow writeTurn.i!i \rightarrow Critical(i,j) \\
& \qquad \Box \\
& \qquad readC.i.j.2 \rightarrow Knuth_0(i,j)
\end{aligned}
$$

After leaving its critical region, program P_i sets the value of the turn variable and $c[i]$ to j

```
VAR flag1, flag2 : Boolean
flag1 := false; flag2 := false;

VAR turn : Id
turn := 1;
```

```
{P1}
 WHILE true DO
  BEGIN
   flag1 := true;
   turn := 2;
   WHILE flag2 AND turn = 2 DO
    nothing;
   <critical region 1>;
   flag1 := false
  END;
```

```
{P2}
 WHILE true DO
  BEGIN
   flag2 := true;
   turn := 1;
   WHILE flag1 AND turn = 1 DO
    nothing;
   <critical region 2>;
   flag2 := false
  END;
```

Table 6.6: Peterson's algorithm.

and 0 respectively.

$$Critical(i,j) \mathrel{\hat{=}} enter.i \rightarrow leave.i \rightarrow writeTurn.i!j \rightarrow writeC.i.i!0 \rightarrow Knuth(i,j)$$

This algorithm is live, safe, fair and starvation-free, with the process to be checked consisting of 132 states.

6.4.5 Peterson's Algorithm

This version of Peterson's algorithm [Pet81] ("an elegant and very simple solution" [Ray86]) is taken from [Wal88]. Again taking the case of $n = 2$, the algorithm involves two flag variables which are both initialised to false, and a turn variable which has an arbitrary initial value.

If program P_i wishes to enter its critical region, then it must first set its flag to *true*, and then write the value of the other program to the turn variable.

$$Peterson(i,j) \mathrel{\hat{=}} ready.i \rightarrow writeFlag.i.i!true \rightarrow writeTurn.i!j \rightarrow Wait(i,j)$$

If it is the case that either it is P_i's turn, or P_j's flag is false, then program P_i can make progress.

$$\begin{aligned} Wait(i,j) \mathrel{\hat{=}}\ & readFlag.i.j.false \rightarrow Critical(i,j) \\ & \Box \\ & readTurn.i.i \rightarrow Critical(i,j) \end{aligned}$$

$$Critical(i,j) \mathrel{\hat{=}} enter.i \rightarrow leave.i \rightarrow writeFlag.i.i!false \rightarrow Peterson(i,j)$$

An advantage of this algorithm is that it is fair, without enforcing alternation, which is a drawback associated with Dekker's algorithm. What we mean by this, is that the entry of a

```
VAR flag : array [1..n] of Boolean
FOR count := 1 TO n DO flag[count] := false;

{Pi}
  VAR j : integer;
  WHILE true DO
    BEGIN
      L0: flag[i] := true;
      FOR j := 1 TO i-1 DO
        IF flag[j] THEN
          BEGIN
            flag[i] := false;
            WHILE flag[j] DO
              nothing;
            GOTO L0
          END;
      FOR j := i+1 TO n DO
        WHILE flag[j] DO
          nothing;
      <critical region i>;
      flag[i] := false
    END;
```

Table 6.7: Lamport's algorithm.

process cannot be delayed by the need for the other to enter first — even if the other does not wish to enter. A further advantage is that the overall system description consists of only 32 possible states.

6.4.6 Lamport's Algorithm

The final algorithm which we consider is that of [Lam86], with the formulation given in Table 6.7 being taken from [Wal89b]. Note that if $n = 2$, then Lamport's algorithm equates to the pair of algorithms given in Table 6.8.

Initially, program P_i writes its flag to true. Following this, a for-loop is entered with initial value 1.

$$Lamport(i) \mathrel{\hat{=}} ready.i \rightarrow Lamport_0(i)$$

$$Lamport_0(i) \mathrel{\hat{=}} writeFlag.i.i!true \rightarrow ForLoop_1(i, 1)$$

If $count > i - 1$, then a second for-loop is invoked, with initial value $i + 1$. On the other hand, if $count \leq i - 1$, then, for every program $1 \leq count < i$, a check is performed. If program P_{count}'s flag is *false*, then the loop counter is incremented by 1. Alternatively, if it is *true*, then program P_i writes its flag to false, and waits for program P_{count}'s flag to show false, at which

```
VAR flag1, flag2 : Boolean
flag1 := false; flag2 := false;

{P1}                              {P2}
  WHILE true DO                     WHILE true DO
    BEGIN                             BEGIN
      L1: flag1 := true;                L1: flag2 := true;
      WHILE flag2 DO                    IF flag1 THEN
        nothing;                          BEGIN
      <critical region 1>;                  flag2 := false;
      flag1 := false                        WHILE flag1 DO
    END;                                      nothing;
                                            GOTO L1
                                          END;
                                        <critical region 2>;
                                        flag2 := false
                                      END;
```

Table 6.8: Lamport's algorithm for $n = 2$.

point program P_i returns to its initial state.

$$
\begin{aligned}
ForLoop_1(i, count) \mathrel{\widehat{=}} \ & \textbf{if} \quad count > i - 1 \\
& \textbf{then } ForLoop_2(i, i + 1) \\
& \textbf{else } readFlag.i.count.false \rightarrow ForLoop_1(i, count + 1) \\
& \Box \\
& readFlag.i.count.true \rightarrow \\
& writeFlag.i.i!false \rightarrow \\
& readFlag.i.count.false \rightarrow Lamport_0(i)
\end{aligned}
$$

The process $ForLoop_2$ is defined thus:

$$
\begin{aligned}
ForLoop_2(i, count) \mathrel{\widehat{=}} \ & \textbf{if} \quad count > n \\
& \textbf{then } Critical(i) \\
& \textbf{else } readFlag.i.count.false \rightarrow ForLoop_2(i, count + 1)
\end{aligned}
$$

If the flags of all programs P_{count} for all $i+1 \leq count \leq n$ are *false*, then the process $Critical(i)$ is invoked.

$$Critical(i) \mathrel{\widehat{=}} enter.i \rightarrow leave.i \rightarrow writeFlag.i.i!false \rightarrow Lamport(i)$$

After executing its critical section, program P_i writes its flag to false, and returns to its initial state.

Our CSP interpretation of this algorithm for the special case of $n = 2$ is defined as the interleaving of the processes $Lamport_1$ and $Lamport_2$.

$$Lamport \mathrel{\widehat{=}} Lamport_1 \ ||| \ Lamport_2$$

The former process is defined thus:

$$Lamport_1 \mathrel{\hat{=}} ready.1 \rightarrow writeFlag.1.1!true \rightarrow readFlag.1.2.false \rightarrow Critical_1$$

$$Critical_1 \mathrel{\hat{=}} enter.1 \rightarrow leave.1 \rightarrow writeFlag.1.1!false \rightarrow Lamport_1$$

The process $Lamport_2$, on the other hand, is defined as follows.

$$Lamport_2 \mathrel{\hat{=}} ready.2 \rightarrow WriteFlag_2$$

$$WriteFlag_2 \mathrel{\hat{=}} writeFlag.2.2!true \rightarrow CheckFlag_2$$

$$\begin{aligned} CheckFlag_2 \mathrel{\hat{=}}\ & readFlag.2.1.false \rightarrow Critical_2 \\ & \Box \\ & readFlag.2.1.true \rightarrow WaitFlag_2 \end{aligned}$$

$$WaitFlag_2 \mathrel{\hat{=}} writeFlag.2.2!false \rightarrow readFlag.2.1.false \rightarrow WriteFlag_2$$

$$Critical_2 \mathrel{\hat{=}} enter.2 \rightarrow leave.2 \rightarrow writeFlag.2.2!false \rightarrow Lamport_2$$

Again, this algorithm is live, safe and starvation-free, but not fair.

6.4.7 Discussion

We have seen how a number of proposed solutions to the mutual exclusion problem can be expressed in terms of CSP processes. We have also seen how a number of different programming concepts, such as assignment and loops can be modelled in CSP. This overview is by no means comprehensive, as many other solutions to the mutual exclusion problem exist.[4]

By representing mutual exclusion algorithms as CSP processes, we can carry out, in an efficient manner, automatic analysis of such algorithms via FDR. As such, not only can we test for the essential properties of deadlock-freedom and the guarantee of mutual exclusion, but we can also investigate whether such algorithms satisfy certain fairness and liveness criteria.

The results of our automated analysis of mutual exclusion algorithms are summarised in Table 6.9. As we can see, all of the algorithms, with the exception of Hyman's, guarantee mutual exclusion. Furthermore, three of the algorithms are fair.

The purpose of the work described in this section was to illustrate how FDR can be applied to the analysis of a number of different properties in an elegant manner. Indeed, when compared with the results of [Wal88], [Wal89b], and [Wal89a], one may conclude that not only do CSP and FDR afford us an elegant means of verifying such properties, they also afford us an *efficient* means. This statement can be qualified by the fact that difficulty was encountered when testing for liveness properties with the Concurrency Workbench [Wal89b].

Non-interference in its form of mutual exclusion has applications in many areas. In particular, we shall find that mutual exclusion plays a major part in defining several safety properties, as well as a number of simplifying assumptions, in the work of Chapter 7. Indeed,

[4] See, for example, the algorithms of de Bruijn [dB67] and Doran and Thomas [DT80].

Algorithm	Live	Safe	Fairness	Starvation-freedom
Dijkstra	✓	✓	×	✓
Dekker	✓	✓	✓	✓
Hyman	✓	×	×	✓
Knuth	✓	✓	✓	✓
Peterson	✓	✓	✓	✓
Lamport	✓	✓	×	✓

Table 6.9: Summary of mutual exclusion algorithms.

we shall see that the proof of one safety invariant, which is a mutual exclusion property, supports the automated analysis of other safety invariants.

Having investigated a number of properties of mutual exclusion algorithms, we now relate the work of Chapter 4 to that described in this chapter, by investigating the fault-tolerance of mutual exclusion algorithms.

6.5 Fault-tolerance

We have seen how we can analyse mutual exclusion algorithms for safety, liveness, fairness, and starvation-freedom with FDR. In this section, we investigate the concept of fault-tolerance for mutual exclusion algorithms. The reader is referred to [Lam86] for a description of different types of fault that one may wish to reason about with regard to mutual exclusion.

For the general case of n programs, we state our interpretation of fault-tolerance thus: If a program P_i is no longer capable of making any further progress, then our required properties of should still be satisfied. That is, if a mutual exclusion algorithm Alg satisfies a given property p, then it should also satisfy p in the presence of faults.

6.5.1 Modelling Faults

In the following, we let $fault.i$, for some $i \in Id$, denote the occurrence of a fault in program P_i. We make three assumptions with regard to faults.

1. Following an observation of $fault.i$, no further events are performed by program P_i.
2. A fault cannot occur when a program is in its critical region.
3. There is a 'safety controller' which, following the observation of a fault, takes appropriate recovery action.

Our second assumption is justified by the fact that if a program fails in its critical region (thus tying up the shared resource), then, despite the fact that mutual exclusion is satisfied trivially, no other program will ever be able to enter its critical region, which invalidates the starvation-freedom condition. While this is an interesting problem, we consider the case of faults occurring outside critical regions, which could have an adverse affect upon mutual exclusion. Given assumptions 1 and 2, we model such faults in the following way.

The process *Faults* is concerned with three channels: *fault*, *enter*, and *leave*.

$$Faults \mathrel{\widehat{=}} |||_{i \in Id} Fault(i)$$

We assume that a fault can occur in any program, and that such faults occur independently.

The process *Fault* is defined thus:

$$\begin{aligned} Fault(i) \mathrel{\widehat{=}}\ & enter.i \rightarrow leave.i \rightarrow Fault(i) \\ & \sqcap \\ & fault.i \rightarrow Stop \end{aligned}$$

The above process allows a fault to occur at any stage of the system's execution, provided that the program in which the fault occurs is not in its critical region. In other words, it models the fault described by assumptions 1 and 2. We say that program P_i is *dead* if *fault.i* has occurred, and that it is *live* otherwise.

6.5.2 Approaches to Fault-tolerance

There are a number of possible ways of dealing with a fault such as that described above. We propose two techniques, both of which involve monitor processes.

Approach 1

Our first approach involves a process which monitors faults. If *fault.i* for any $i \in Id$ is observed, then the monitor process writes program P_i's flag to *false*. We think of such a monitor process as being program P_0, and define the set $Id0$ accordingly.

$$Id0 = \{0, \ldots, n\}$$

Following a fault, the monitor process takes appropriate action.

$$Monitor_1 \mathrel{\widehat{=}} fault?i \rightarrow writeFlag.0.i!false \rightarrow Monitor_1$$

If a given mutual exclusion algorithm involves a turn variable as well as flag variables, then $Monitor_1$ should also be capable of updating the turn variable. We redefine our process accordingly.

$$Monitor_1 \mathrel{\widehat{=}} Monitor(\varnothing)$$

In this case, the set s records which programs have died.

$$\begin{aligned} Monitor_1(s) \mathrel{\widehat{=}}\ & fault?i \rightarrow writeFlag.0.i.false \rightarrow \\ & readTurn.0?j \rightarrow \textbf{if} \quad i = j \\ & \qquad\qquad\qquad\quad \textbf{then}\ MonitorForLoop(s \cup \{i\}, i) \\ & \qquad\qquad\qquad\quad \textbf{else}\ Monitor_1(s \cup \{i\}) \end{aligned}$$

This time, after a program dies, the value of the *turn* variable is read. If it is program P_i's turn and P_i has just died, then the turn is passed to the next live program. In both cases, s is

updated accordingly.

$$\begin{aligned} \mathit{MonitorForLoop}(s,k) \mathrel{\widehat{=}} & \textbf{if} \quad k > n \\ & \textbf{then}\ \mathit{MonitorForLoop}(s,1) \\ & \textbf{else}\ \textbf{if} \quad k \in s \\ & \qquad \textbf{then}\ \mathit{MonitorForLoop}(s,k+1) \\ & \qquad \textbf{else}\ \mathit{writeTurn}.0!k \rightarrow \mathit{Monitor}_1(s) \end{aligned}$$

We think of this approach as taking an active role in the provision of fault-tolerance — in some sense, the monitor process 'cleans up' after a fault.

Approach 2

A more passive means of fault-tolerance is provided by our second approach. In this case, the monitor process observes faults, and, upon request, provides information related to which programs are dead. Thus, rather than resetting a flag following a fault, this process counters occurrences of faults by informing programs that another program is dead and, therefore even if its flag is up, it will never be capable of entering its critical region.

Such a monitor process is defined thus:

$$\mathit{Monitor}_2 \mathrel{\widehat{=}} \mathit{Monitor}_2(\varnothing)$$

$$\begin{aligned} \mathit{Monitor}_2(s) \mathrel{\widehat{=}} & \ \mathit{fault}?i \rightarrow \mathit{Monitor}_2(s \cup \{i\}) \\ & \ \Box \\ & \ \mathit{readFailed}?i!s \rightarrow \mathit{Monitor}_2(s) \end{aligned}$$

Note that a similar passive treatment for a turn variable is possible: rather than checking whether it is program P_i's turn, we check whether it is program P_i's turn *and* if it is live.

6.5.3 An Example

As a means of illustrating how this approach might be applied, we take Lamport's one-bit algorithm for n programs. If we were to take an active approach to fault-tolerance, then of course we do not need to rewrite our process, as faults will be dealt with by a monitor process, and flags will be set to *false* following the occurrence of faults. On the other hand, if we take a passive view, then the checks have to be made as to which processes are live or dead. As such, we rewrite *Lamport* in the following way.

The first action of program P_i is the same as for the 'normal' version of the algorithm: it writes the value of its flag to *true*.

$$\mathit{Lamport}(i) \mathrel{\widehat{=}} \mathit{ready}.i \rightarrow \mathit{Lamport}_0(i)$$

$$\mathit{Lamport}_0(i) \mathrel{\widehat{=}} \mathit{writeFlag}.i.i!\mathit{true} \rightarrow \mathit{ForLoop}_1(i,1)$$

As before, if the value of *count* exceeds $i - 1$, then a second for-loop is invoked.

$$
\begin{aligned}
&ForLoop_1(i, count) \mathrel{\widehat{=}} \\
&readFailed.i?s \rightarrow \textbf{if} \quad count > i - 1 \\
&\qquad\qquad \textbf{then}\ ForLoop_2(i, i + 1) \\
&\qquad\qquad \textbf{else}\ \textbf{if} \quad count \in s \\
&\qquad\qquad\qquad \textbf{then}\ ForLoop_1(i, count + 1) \\
&\qquad\qquad\qquad \textbf{else}\ readFlag.i.count.false \rightarrow ForLoop_1(i, count + 1) \\
&\qquad\qquad\qquad\qquad \Box \\
&\qquad\qquad\qquad\qquad readFlag.i.count.true \rightarrow \\
&\qquad\qquad\qquad\qquad writeFlag.i.i!false \rightarrow \\
&\qquad\qquad\qquad\qquad readFlag.i.count.false \rightarrow Lamport_0(i)
\end{aligned}
$$

If *count* exceeds n in the process $ForLoop_2$, then program P_i enters its critical section. Otherwise, the process waits for program P_{count}'s flag to become *false* before continuing.

$$
\begin{aligned}
&ForLoop_2(i, count) \mathrel{\widehat{=}} \textbf{if} \quad count > n \\
&\qquad\qquad\qquad\qquad \textbf{then}\ Critical(i) \\
&\qquad\qquad\qquad\qquad \textbf{else}\ CheckFlag(i, count)
\end{aligned}
$$

$$
\begin{aligned}
&CheckFlag(i, count) \mathrel{\widehat{=}} readFlag.i.count.false \rightarrow ForLoop_2(i, count + 1) \\
&\qquad\qquad\qquad\qquad \Box \\
&\qquad\qquad\qquad\qquad readFlag.i.count.true \rightarrow \\
&\qquad\qquad\qquad\qquad readFailed.i?s \rightarrow \textbf{if} \quad count \in s \\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{then}\ ForLoop_2(i, count + 1) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ CheckFlag(i, count)
\end{aligned}
$$

After program P_i has left its critical section, it sets its flag to *false*.

$$Critical(i) \mathrel{\widehat{=}} enter.i \rightarrow leave.i \rightarrow writeFlag.i.i!false \rightarrow Lamport(i)$$

Given our updated CSP interpretation of Lamport's algorithm, we define *LamportAlg* in the following way.

$$LamportAlg \mathrel{\widehat{=}} |||_{i \in Id}(Lamport(i) \triangle (fault.i \rightarrow Stop))$$

Here, following a fault in program P_i, that program is no longer capable of interacting with the system.

We define our safety controller, which incorporates our notions of passive and active fault-tolerance, in the following way.

Initially, no programs have failed.

$$SafetyController \mathrel{\widehat{=}} SafetyController(\varnothing)$$

$$
\begin{aligned}
&SafetyController(s) \mathrel{\widehat{=}} readFailed?i!s \rightarrow SafetyController(s) \\
&\qquad\qquad\qquad\qquad \Box \\
&\qquad\qquad\qquad\qquad fault?i \rightarrow writeFlag.0.i!false \rightarrow SafetyController(s \cup \{i\})
\end{aligned}
$$

Here, if a fault occurs in program P_i, then the safety controller writes that program's flag to *false*. It follows that we define the process *FTLamport* by

$$FTLamport \mathrel{\widehat{=}} (LamportAlg \parallel SafetyController \parallel LamportVars \parallel Faults) \setminus \alpha SafetyController$$

It follows that if this process satisfies our requirements, then Lamport's algorithm is fault-tolerant with respect to this fault.

Verification with FDR confirms that this process is safe, live, fair, and starvation-free, provided that it is guaranteed that at least one program will never fail — obviously, if all programs fail, then safety is satisfied trivially, but liveness cannot be satisfied.

6.5.4 Formalising Fault-tolerance

In the general case of a mutual exclusion algorithm *Alg*, if we define

$$FTAlg \mathrel{\widehat{=}} (Alg \parallel SafetyController \parallel Vars \parallel Fault) \setminus \alpha SafetyController$$

then we may recast our definitions of safety, liveness, fairness, and starvation-freedom in terms of fault-tolerance.

Definition 6.5 A fault-tolerant mutual exclusion algorithm *FTAlg* is safe if and only if

$$FTAlg \textbf{ imp } MutualExclusionCheck$$

□

Definition 6.6 A fault-tolerant mutual exclusion algorithm *FTAlg* is live if and only if

$$CHAOS_\Sigma \sqsubseteq FTAlg$$

□

Definition 6.7 A fault-tolerant mutual exclusion algorithm *FTAlg* is fair if and only if

$$FTAlg \textbf{ imp } FairnessCheck$$

□

Definition 6.8 A fault-tolerant mutual exclusion algorithm *FTAlg* is starvation-free if and only if

$$\textbf{nondiv}(FTAlg \setminus (\alpha FTAlg - Critical))$$

□

Finally, we provide two definitions of what we mean for a mutual exclusion algorithm to be fault-tolerant, one in terms of the above properties, and the other in terms of protection.

Definition 6.9 A mutual exclusion algorithm *Alg* is fault-tolerant if and only if

$$
\begin{aligned}
&(Alg \ \mathbf{imp}\ MutualExclusionCheck) \Rightarrow (FTAlg \ \mathbf{imp}\ MutualExclusionCheck) \\
&\wedge \\
&(CHAOS_\Sigma \sqsubseteq Alg) \Rightarrow (CHAOS_\Sigma \sqsubseteq FTAlg) \\
&\wedge \\
&(Alg \ \mathbf{imp}\ FairnessCheck) \Rightarrow (FTAlg \ \mathbf{imp}\ FairnessCheck) \\
&\wedge \\
&(\mathbf{nondiv}(Alg \setminus (\alpha Alg - Critical))) \Rightarrow (\mathbf{nondiv}(FTAlg \setminus (\alpha FTAlg - Critical)))
\end{aligned}
$$

□

Definition 6.10 A fault-tolerant mutual exclusion algorithm *FTAlg* is protected if and only if

$$\forall\, i, j : Id \mid i \neq j \bullet \{enter.i\} \nleftarrow \{fault.j\}\ [FTAlg]$$

□

In a sense, this definition brings us full circle in that it further emphasises the relationship between non-interference and mutual exclusion — a fault-tolerant mutual exclusion algorithm is protected if and only if it is firewalled.

6.6 A Comparison of Approaches

In this chapter, we have translated CCS representations of a number of mutual exclusion algorithms into CSP. Our translation from CCS to CSP has been rather informal — the process has been one of transliteration, rather than of a formal translation. The aim was not to show that such links could be made between the process algebras, nor to provide a comparison between the techniques, but rather to compare the virtues of the automatic analysis techniques available for them.

As we have already seen, the mechanical verification of mutual exclusion with FDR outperforms similar verification with the Concurrency Workbench.

Each of the problems that have been described thus far in this thesis have been relatively simple, and one may pose the question: How well can FDR be applied to real-life, large-scale problems? We attempt to answer this question in the next chapter, by applying FDR to such a problem — the mechanical verification of geographic data associated with Solid State Interlocking railway signalling systems.

Chapter 7

The Mechanical Verification of SSI Data

In this chapter, we describe how FDR has been applied to the analysis of a number of safety invariants for geographic data associated with Solid State Interlocking (SSI) railway signalling systems. Solid State Interlocking is an interesting case study as many of its safety requirements can be stated in terms of mutual exclusion. As well as demonstrating our approach on a simple interlocking, we illustrate that the methods scale up to industrial size by applying them to a simplified version of data associated with a real interlocking system. The geographic data used in this chapter was kindly supplied by British Rail Research (Derby).

7.1 Formal Methods and the Railways Signalling Industry

As is the case with most other industries, it is hard to judge the uptake of formal methods in the railways signalling industry because of the over-riding 'commercial-in-confidence' factor. Nevertheless, a number of references relating formal methods and railways signalling do exist in the literature. In general, such texts can be placed into one of three distinct categories.

The first category consists of references where the main aim is to present theoretical results, and the railway aspect is used merely as a case study to illustrate these results — this often leads to the systems in question being dismissed as 'toy examples'. An example is [dLSA92], where a signalling layout is used to illustrate a method for formal requirements analysis.

The second category is made up of references which apply established formal techniques to well-documented problems in the railway industry. An example of such a problem is described in this chapter — the mechanical verification of geographic data associated with Solid State Interlocking signalling systems. Other approaches applied to this problem include CCS [Mor91] and HOL [Mor93].

The final and most important category involves references which describe the application of formal methods by the industry's practitioners themselves. An example is [Kin94], which describes work associated with the formalising of British Rail's signalling rules in Z. Others include [IM95], [DM94], and [DM95]. The latter two papers summarise the use of B in three projects by GEC Alsthom in the development of railway signalling applications. The best known of these applications is that of the use of B in the Système d'Aide à la Conduite et à la Maintenance (SACEM) project, which is also described in [CGR93].

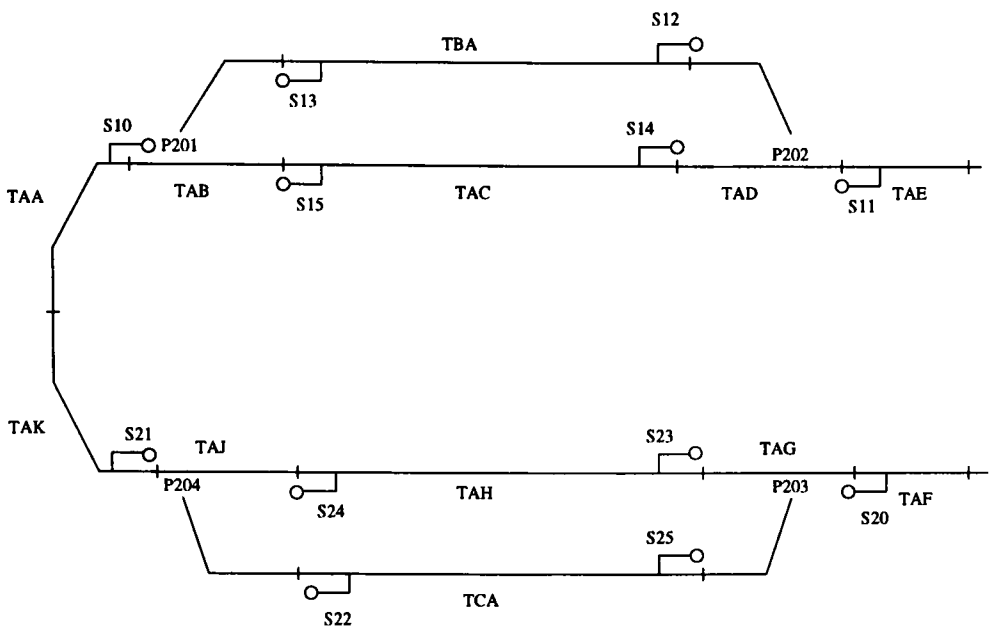

Figure 7.1: The Open ALVEY interlocking.

7.2 Solid State Interlocking

Solid State Interlocking (SSI) signalling systems [Cri87, CM92] control a large percentage of Great Britain's railway network. As well as being applied in Great Britain, SSI has been adapted successfully to Belgian, Indonesian, Korean, Danish, Australian, Portuguese, Chinese, and South African signalling practices; it is also in competition for use in other markets because of the ease with which its data language can be customised with minimal maintenance to generic programs. In addition, adaptions to new methods of train detection, level crossing control, radio electronic token blocks, and integration into multi-function control centres are all under active development and commercially available.

SSI is a computerised control system used on railways, which replaces the traditional relay-based electromechanical systems. One of its functions is to grant, subject to successful requests by a signalman, routes to trains. This involves the locking of points and sub-routes, and the setting of signals. As is the case for all signalling systems, the fundamental requirement of the system is that it should reduce the possibility of a number of hazards. Examples of hazards include collisions, which are possible if two trains occupy the same section of track, and derailments, which are possible if points move under a train [DTI94, IM95].

At their simplest, SSIs can be viewed as consisting of two components: generic programs, which are concerned with the transfer of data between interlockings and the track-side; and geographic databases, which are concerned with local data, such as signals and points.

The need for complete, consistent and correct geographic data for SSI systems is clear. Such databases are, to a large extent, currently verified manually — as such, any steps taken towards a fully automated formal approach in this area would be of great benefit. This mostly manual approach to verification is the 'state-of-the-art' — we propose a method for automating the process in this chapter.

Because of the state explosion involved (the internal state typically has 2^n possible

configurations [Mor91][1]) and the difficulty in "identifying the actual behaviour of this system — the intrinsic state space of the SSI — among this chaos" [Mor91], the mechanical verification of safety invariants for geographic data has been considered to be of too great a complexity for the current generation of automated proof tools. Indeed, it is precisely this difficulty in decomposing the problem at hand that has limited the success of other approaches, such as CCS.

In this chapter, we suggest an approach for separating the safety invariant for geographic data of [Mor93] into a series of smaller, more manageable proofs. We identify the specific interlocking components with which particular invariants are concerned and, via FDR, illustrate that it is feasible to verify safety properties of complex interlocking data in reasonable time on a standard workstation.

Work described in this chapter has previously appeared as [Sim95b] and [SWD96]. Other references of interest in this area include [AC91], [Mor91], [IM92], [Mor93], [Han94], [CP95], [Ing95], and [IM95].

7.3 Geographic Data

British Rail's Geographic Data Language (GDL) [BRB90] provides a means of describing signalling rules for railway junctions in terms of its track circuits, points, signals, routes, and sub-routes. Each of these concepts are introduced in this section, and are illustrated by the Open ALVEY interlocking of Figure 7.1 [Mee94], the geographic data for which is given in Appendix B.

Track Circuits

Track circuits (e.g., TAA, TAB, etc.) divide the track into sections, with train detection devices informing the interlocking if a section is *occupied* or *clear*, represented by 'o' and 'c' respectively.

Points

Points (e.g., P201, P202, etc.) perform the function of steering trains across junctions, with a set of points being in one of two positions at any time — *controlled normal* ('cn') or *controlled reverse* ('cr'). Taking Figure 7.1 as an example, if a train is travelling over track circuit TAA towards track circuit TAB and points P201 are in controlled normal position, then the train will continue along TAB towards track circuit TAC. On the other hand, if points P201 are in controlled reverse position, then the train will continue towards track circuit TBA. In addition, Boolean checks may be performed on points. These checks determine whether or not points are able to move into controlled normal or controlled reverse position, and are termed *free to go normal* and *free to go reverse* respectively. A set of points is controlled free to go normal ('cfn') (respectively, controlled free to go reverse ('cfr')) either if they are free to go normal (respectively, free to go reverse), or if they are already in the normal position (respectively, already in the reverse position).

[1] Where n is the number of components, e.g., points, signals, etc. with which the system is concerned.

Signals

Signals (e.g., S10, S11, etc.), when viewed at the simplest level, can allow or refuse, depending upon their *aspect*, the entry of a train onto particular sections of track. Signals are situated in advance of the section which they control. For example, signal S20 controls entry onto the section made up of track circuits TAG, TAH and TCA, while signals S23 and S25 both control entry onto the section consisting of track circuits TAF and TAG.

Routes

Routes are sections of track between two signals, which proceed from an *entry signal* to an *exit signal.* For example, in Figure 7.1, route R13 is the section of track between signal S13 (the entry signal) and signal S21 (the exit signal). This route spans three track circuits: TAB, TAA and TAK, and contains one set of points: P201. A route may be in one of two states at any given time: *set* ('s'), or *unset* ('xs').[2] Note that there are two routes which start at signal S10 — one travels over circuits TAB and TBA, and the other over circuits TAB and TAC. These routes are identified by R10A and R10B respectively.

Sub-routes

Sub-routes are subsections of routes which are associated with specific track circuits. The naming convention for sub-routes is as follows. Given any track circuit, there may be any number of sub-routes over that circuit, with that number being dependent upon the possible entries and exits for that track circuit. As an example, track circuit TAB has three entry and exit points: from (or to) track circuits TAA, TAC and TBA. These entry and exit points are labelled clockwise from a 12.00 position. Thus, entry (or exit) from (or to) circuit TBA is labelled A, entry/exit from/to TAC is labelled B, and C is associated with entry/exit from/to TAA. It follows that sub-route UAB-AC travels over track circuit TAB, with entry from track circuit TBA and exit at track circuit TAA. In addition, sub-route UAB-CB has entry from TAA and exit at TAC, and sub-route UAB-BC runs from TAC to TAA. Finally, the entry and exit points for UAB-CA are TAA and TBA respectively. As with other system components, sub-routes can be in one of two states at any time: *locked* ('l') or *free* ('f').

7.4 Conditional Checks

The setting of routes, freeing of sub-routes, and movement of points are all dependent upon conditional checks performed upon geographic data. We introduce such checks in this section. Note that we limit ourselves to these types of check as a means of simplification. Note also that SSI performs functions other than those dealt with in this thesis.

Route Setting Data

The first conditional check with which we are concerned is that associated with the setting of routes. We take, as an example, the condition associated with the setting of route R14.

[2] Although in the following, we identify a further state — *requested* ('req') — which represents the intermediate state between a route being requested and the moment it becomes set.

This route runs from signal S14 over track circuits TAD and TAE and points P202. Referring to Appendix B, the condition concerned with setting route R14 is written

Q14 **if** P202 cfn UAE-AB f UAD-AB f
then R14 s P202 cn UAD-BA l UAE-BA l

This condition is interpreted in the following way. Before route R14 can be set, points P202 must be controlled free to go normal, and sub-routes UAE-AB and UAD-AB must both be free. If any of these conditions are not met, then the route cannot be granted. Alternatively, if each Boolean check evaluates to true, then route R14 can be set. This involves moving points P202 to controlled normal position (if they are not already in that position), and locking sub-routes UAD-BA and UAE-BA.

We assume that the locking of routes is atomic. This is in the sense that two routes cannot be in the process of being locked simultaneously. Also, for the moment, we apply no conditions upon routes becoming unset, and assume that any route can enter that state unconditionally and at any time.

Sub-route Release Data

Recall that a sub-route can be in one of two states at any time — locked or free. A sub-route is locked when some route of which that sub-route is part is set. A sub-route becomes free when a conditional check performed upon conditions associated with it evaluates to true. In the above example, when route R14 is set, sub-routes UAD-BA and UAE-BA are both locked. Again referring to Appendix B, the latter becomes free when the following condition is met.

UAE-BA f **if** TAE c UAD-BA f UAD-CA f

That is, UAE-BA becomes free when track circuit TAE is clear, and sub-routes UAD-BA and UAD-CA are both free. Furthermore, sub-route UAD-BA becomes free when track circuit TAD is clear and route R14 is unset.

UAD-BA f **if** TAD c R14 xs

Points Free to Move Data

Certain conditions must hold of an interlocking before points can move. Such conditions are referred to as points free to move data. Referring to Figure 7.1, the berth track circuit of points P201 (TAB) must be clear before that set of points can move in either direction. If it were not, a derailment might occur. In addition, before this set of points can move into controlled normal position, sub-routes UAB-AC and UAB-CA must both be free. Furthermore, UAB-BC and UAB-CB must both be free before these points become free to go reverse.

Such a condition is written thus:

P201N TAB c UAB-AC f UAB-CA f
P201R TAB c UAB-BC f UAB-CB f

Here, points P201 are free to go normal if and only if track circuit TAB is clear and sub-routes UAB-AC and UAB-CA are both free. Furthermore, they are controlled free to go normal if and only if P201 are controlled normal or they are free to go normal.

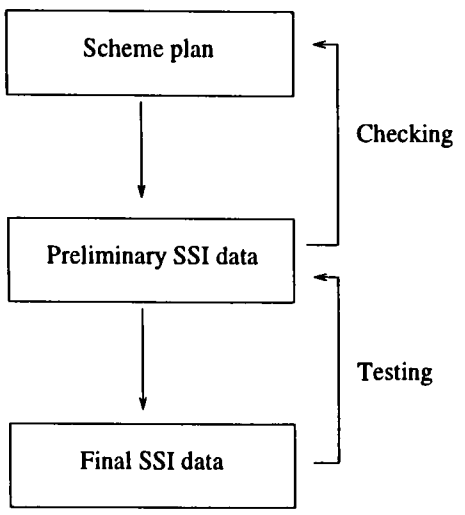

Figure 7.2: The preparation of SSI data.

7.5 Safety Invariants

There are a number of safety invariants which must hold of all SSI geographic data. Examples of such properties are given below.

1. For every track circuit, no more than one of the sub-routes passing over it should be locked for a route at any time.

2. If a sub-route over a track section containing points is locked for a route, then the points are correctly aligned with that sub-route.

3. If a route is set, then all of its sub-routes are locked.

4. If a track circuit containing points is occupied, then the points are locked.

5. If a sub-route is locked for a route, then all sub-routes ahead of it on that route are also locked.

We refer to the Open ALVEY as a means of illustrating these invariants. Invariant 1 states that only one of the sub-routes which run over track circuit TAB, i.e., only one of UAB-AC, UAB-BC, UAB-CA and UAB-CB can be locked for a route at any time. Invariant 2 states that if sub-route UAB-BC is locked, then points P201 are controlled normal. Alternatively, if sub-route UAB-AC is locked, then points P201 are controlled reverse. Invariant 3 states that if route R14 is set, then sub-routes UAD-AB and UAE-AB are both locked. Invariant 4 states that if track circuit TAB is occupied, then points P201 are locked. Finally, invariant 5 states that if sub-route UAD-BA is locked for route R14, then sub-route UAE-BA is also locked.

If any of the above invariants are not satisfied for a particular interlocking, then there is an error in the geographic data. We describe a means for analysing such data via FDR in this chapter, and start, in the next section, by providing a formal treatment of SSI components.

Referring to Figure 7.2, we can think of the approach described in this chapter as playing the role of testing.

7.6 A CSP Approach

7.6.1 Basic Types and Functions

The first step towards the verification of geographic data for a particular interlocking is that of identifying the basic elements with which we are concerned. In all cases, these are an interlocking's track circuits, points, signals, routes, and sub-routes.

Referring to Figure 7.1, we define the set of track circuits, $\mathcal{T}$, the set of points, $\mathcal{P}$, the set of signals, S, the set of routes, $\mathcal{R}$, and the set of sub-routes, $\mathcal{U}$, thus:

$$\mathcal{T} = \{\text{TAA, TAB, TAC, TAD, TAE, TAF, TAG, TAH, TAJ, TAK, TBA, TCA}\}$$

$$\mathcal{P} = \{\text{P201, P202, P203, P204}\}$$

$$S = \{\text{S10, S11, S12, S13, S14, S15, S20, S21, S22, S23, S24, S25}\}$$

$$\mathcal{R} = \{\text{R10A, R10B, R11A, R11B, R12, R13, R14, R15, R20A, R20B, R21A, R21B, R22, R23, R24, R25}\}$$

$$\begin{aligned}\mathcal{U} = \{&\text{UAA-AB, UAA-BA, UAB-AC, UAB-BC, UAB-CA, UAB-CB, UAC-AB, UAC-BA, UAD-AB, UAD-AC,}\\ &\text{UAD-BA, UAD-CA, UAE-BA, UAF-BA, UAG-AB, UAG-AC, UAG-BA, UAG-CA, UAH-AB, UAH-BA,}\\ &\text{UAJ-AC, UAJ-BC, UAJ-CA, UAJ-CB, UAK-AB, UAK-BA, UBA-AB, UBA-BA, UCA-AB, UCA-BA}\}\end{aligned}$$

Note that we construct these sets from the interlocking scheme plan, rather than the geographic data. This gives us a means of verification which is independent of the data.

We represent an interlocking, $\mathcal{I}$, in terms of a quintuple

$$\mathcal{I} = (\mathcal{T}, \mathcal{P}, S, \mathcal{R}, \mathcal{U})$$

We also define several functions, which define the relationships between the components.

The first such function,

$$locks : \mathcal{R} \rightarrow \mathbb{P}\,\mathcal{P}$$

relates routes and points. Given some route $r \in \mathcal{R}$, $locks(r)$ returns the set of all points which are locked when r is set. Taking for example our Open ALVEY interlocking,

$$locks(\text{R10A}) = \{\text{P201}\}$$

The function

$$sr : \mathcal{T} \rightarrowtail \mathbb{P}\,\mathcal{U}$$

takes a track circuit $t \in \mathcal{T}$ and returns the set of sub-routes which are associated with that circuit. In the case of the Open ALVEY,

$$sr(\text{TAB}) = \{\text{UAB-AC, UAB-BC, UAB-CA, UAB-CB}\}$$

The function

$$berth : \mathcal{P} \rightarrow \mathbb{P}\,\mathcal{T}$$

returns, for all $p \in \mathcal{P}$, the berth circuits of points p. In the case of the Open ALVEY,

$$berth = \{\text{P201} \mapsto \{\text{TAB}\}, \text{P202} \mapsto \{\text{TAD}\}, \text{P203} \mapsto \{\text{TAG}\}, \text{P204} \mapsto \{\text{TAJ}\}\}$$

The function *circuit* returns the berth circuit of a sub-route.

$$circuit : \mathcal{U} \rightarrow \mathcal{T}$$

For example,

$$circuit(\text{UAB-AC}) = \text{TAB}$$

Also,

$$subroutes : \mathcal{R} \rightarrowtail \text{seq}\,\mathcal{U}$$

relates a route $r \in \mathcal{R}$ to its component sub-routes in the order in which they are occupied by any train which follows that route. In the case of route R13,

$$subroutes(\text{R13}) = \langle \text{UAB-AC, UAA-AB, UAK-AB} \rangle$$

Finally, the functions

$$norm : \mathcal{P} \rightarrow \mathbb{P}\,\mathcal{U}$$
$$rev : \mathcal{P} \rightarrow \mathbb{P}\,\mathcal{U}$$

relate points to the sub-routes which run over them when they are in controlled normal or controlled reverse position respectively. In the case of the Open ALVEY, these functions are defined thus:

$$\begin{aligned} norm = \{&\text{P201} \mapsto \{\text{UAB-BC, UAB-CB}\}, \text{P202} \mapsto \{\text{UAD-AB, UAD-BA}\}, \\ &\text{P203} \mapsto \{\text{UAG-AC, UAG-CA}\}, \text{P204} \mapsto \{\text{UAJ-AC, UAJ-CA}\}\} \end{aligned}$$

$$\begin{aligned} rev = \{&\text{P201} \mapsto \{\text{UAB-AC, UAB-CA}\}, \text{P202} \mapsto \{\text{UAD-AC, UAD-CA}\}, \\ &\text{P203} \mapsto \{\text{UAG-AB, UAG-BA}\}, \text{P204} \mapsto \{\text{UAJ-BC, UAJ-CB}\}\} \end{aligned}$$

7.6.2 Channels

We start our translation of SSI geographic data into CSP by introducing a number of channels which are associated with changes in state of interlocking components.

The changes in states of routes are represented by communications on the channel *routeState*.

$$routeState = \{r : \mathcal{R};\ x : \{req, s, xs\} \bullet routeState.r.x\}$$

Here, *routeState.r.req* indicates that some route $r \in \mathcal{R}$ has been requested, *routeState.r.s* indicates that r has been set, and *routeState.r.xs* indicates that r has become unset. We choose not to record unsuccessful requests — if the state of the system is such that a particular route cannot be set, then the communication representing the successful request for that route is refused.

Communications on *subrouteState* represent the change in state of a sub-route.

$$subrouteState = \{u : \mathcal{U};\ x : \{f, l\} \bullet subrouteState.u.x\}$$

The events contained in *pointState* indicate whether a set of points has become (or ceased to be) controlled free to go normal or controlled free to go reverse respectively.

$$pointState = \{p : \mathcal{P};\ x : \{cfn, cfr\};\ b : Bool \bullet pointState.p.x.b\}$$

A change in the position of a set of points is represented by observations on the channel *pointPosition*.

$$pointPosition = \{p : \mathcal{P};\ pos : \{cn, cr\} \bullet pointPosition.p.pos\}$$

We also have a notion of points locking, which is represented by communications on the channel *pointLocked*.

$$pointLocked = \{p : \mathcal{P};\ b : Bool \bullet pointLocked.p.b\}$$

Finally, the set *circuitState* contains observations relating to track circuits becoming clear or occupied.

$$circuitState = \{t : \mathcal{T};\ x : \{c, o\} \bullet circuitState.t.x\}$$

7.6.3 Initial Assumptions

We make a number of assumptions with regard to an interlocking's initial state. First, we assume that all track circuits are initially free, and that all points are initially unlocked and controlled normal (and, therefore, controlled free to go normal). We also assume that, initially, no points are free to move in either direction, all sub-routes are initially locked, and all routes are initially unset.[3]

7.7 Translating From GDL to CSP

The need to establish the correctness of our translation from GDL to CSP is essential if our approach to modelling SSI geographic data is to be accepted. In this section, we provide

[3]Note that this is referred to as a cold or mode 1 startup. SSI has other warm startup procedures, which may equally well be modelled.

an outline of our translation, which, although being no more than semi-formal, should be sufficiently straightforward to convince the reader of its correctness.

We start by introducing a number of functions, which at any time, provide the states of particular interlocking components. We assume given types

$$[VAR_T, VAR_P, VAR_R, VAR_U]$$

which contain variables of types $\mathcal{T}$, $\mathcal{P}$, $\mathcal{R}$, and $\mathcal{U}$ respectively. The following functions assign values to variables.

$$value_T : VAR_T \twoheadrightarrow \{c, o\}$$
$$value_P : VAR_P \times \{cfn, cfr\} \twoheadrightarrow Bool$$
$$value_R : VAR_R \twoheadrightarrow \{req, s, xs\}$$
$$value_U : VAR_U \twoheadrightarrow \{f, l\}$$

In addition, we define

$$state_T : \mathcal{T} \rightarrow \{c, o\}$$
$$state_P : \mathcal{P} \times \{cfn, cfr\} \rightarrow Bool$$
$$state_R : \mathcal{R} \rightarrow \{req, s, xs\}$$
$$state_U : \mathcal{U} \rightarrow \{f, l\}$$

which relate interlocking components to their states.

We assign variable names to interlocking components via the total injections

$$var_T : \mathcal{T} \rightarrowtail VAR_T$$
$$var_P : \mathcal{P} \rightarrowtail VAR_P$$
$$var_R : \mathcal{R} \rightarrowtail VAR_R$$
$$var_U : \mathcal{U} \rightarrowtail VAR_U$$

Note that the following are invariants of this translation process.

$$\forall\, t : \mathcal{T} \bullet value_T(var_T(t)) = state_T(t)$$
$$\wedge$$
$$\forall\, p : \mathcal{P} \bullet value_P(var_P(p), cfn) = state_P(p, cfn)$$
$$\wedge$$
$$\forall\, p : \mathcal{P} \bullet value_P(var_P(p), cfr) = state_P(p, cfr)$$
$$\wedge$$
$$\forall\, r : \mathcal{R} \bullet value_R(var_R(r)) = state_R(r)$$
$$\wedge$$
$$\forall\, u : \mathcal{U} \bullet value_U(var_U(u)) = state_U(u)$$

That is, the value of a variable representing an interlocking component is always consistent with that component's actual state.

7.7.1 Route Setting Data

We think of route setting data as a quintuple of the form

$$\mathcal{R} \times \mathbb{P}(\mathcal{P} \times \{cfn, cfr\}) \times \mathbb{P}\,\mathcal{U} \times \text{seq}(\mathcal{P} \times \{cn, cr\}) \times \text{seq}\,\mathcal{U}$$

consisting of the route to be set, a set of conditions on sets of points, a set of sub-routes which are required to be free, a sequence of sets of points (together with their required positions) to be locked, and a sequence of sub-routes to be locked.

For example, the route setting data for R14, which is given by

Q14 **if** P202 cfn UAE-AB f UAD-AB f
then R14 s P202 cn UAD-BA l UAE-BA l

can be rewritten as (R14, {(P202, *cfn*)}, {UAE-AB, UAD-AB}, ⟨(P202, *cn*)⟩, ⟨UAD-BA, UAE-BA⟩).

In general, given a quintuple (r, P_1, U_1, P_2, U_2), we construct the corresponding CSP process, *r_false*, in the following way.

We start by defining the alphabet of this process, which is given by αr_false.

$$\begin{aligned}\alpha r\_false = {} & \{routeState.r.req, routeState.r.s\} \\ & \cup \\ & \{p : \mathcal{P};\ c : \{cfn, cfr\};\ b : Bool \mid (p, c) \in P_1 \bullet pointState.p.c.b\} \\ & \cup \\ & \{u : \mathcal{U};\ x : \{f, l\} \mid u \in U_1 \bullet subrouteState.u.x\} \\ & \cup \\ & \{p : \mathcal{P};\ pos : \{cn, cr\} \mid (p, pos) \in \mathbf{elems}(P_2) \bullet pointPosition.p.pos\} \\ & \cup \\ & \{u : \mathcal{U} \mid u \in \mathbf{elems}(U_2) \bullet subrouteState.u.l\}\end{aligned}$$

The number of parameters which this process deals with depends upon the sizes of P_1 and U_1, as it is these conditions on points and sub-routes which determine when r can be set.

Note that initially,

$$\begin{aligned}& \forall u : \mathcal{U} \mid u \in U_1 \bullet state_U(u) = l \\ & \wedge \\ & \forall p : \mathcal{P};\ c : \{cfn, cfr\} \mid (p, c) \in P_1 \bullet c = cfn \Rightarrow state_P(p, c) = true \wedge \\ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad c = cfr \Rightarrow state_P(p, c) = false\end{aligned}$$

The process *r_false* is defined informally as follows. Initially, it performs a conditional check upon its variables. If

$$\begin{aligned}& \forall u : \mathcal{U} \mid u \in U_1 \bullet value_U(var_U(u)) = f \\ & \wedge \\ & \forall p : \mathcal{P};\ c : \{cfn, cfr\} \mid (p, c) \in P_1 \bullet value_P(var_P(p), c) = true\end{aligned}$$

then the process *r_true* is invoked. Alternatively, the external choice between observations on the channels corresponding to the elements of U_1 and P_1 is offered.

The process *r_true* offers the external choice between observations of *routeState.r.req*, the locking of sub-routes $u \in U_1$, and relevant changes in state of points contained in the set $p \in points(P_1)$, where *points* is defined by

$$\begin{array}{|l} points : \mathbb{P}(\mathcal{P} \times \{cfn, cfr\}) \rightarrow \mathbb{P}\,\mathcal{P} \\ \hline \forall P : \mathbb{P}(\mathcal{P} \times \{cfn, cfr\}) \bullet points(P) = \{p : \mathcal{P};\ c : \{cfn, cfr\} \mid (p, c) \in P \bullet p\} \end{array}$$

If *routeState.r.req* is observed, then the points are locked in accordance with P_2, sub-routes contained in U_2 are locked, and the route is set. Alternatively, if relevant changes in state are observed, then *r_false* is invoked with relevant values.

7.7.2 Sub-route Release Data

We think of sub-route release data as a quadruple of the form

$$\mathcal{U} \times \mathbb{P}\,\mathcal{T} \times \mathbb{P}\,\mathcal{U} \times \mathbb{P}\,\mathcal{R}$$

consisting of the sub-route to be released, a set of track circuits which are required to be clear, a set of conditions on sub-routes which are required to be free, and a set of conditions on routes which are required to be unset.

As an example, we interpret the sub-route release data given by

UAD-BA f **if** TAD c R14 xs

as (UAD-BA, {TAD}, ∅, {R14}). As a further example, the sub-route release data

UAE-BA f **if** TAE c UAD-BA f UAD-CA f

is interpreted as (UAE-BA, {TAE}, {UAD-BA, UAD-CA}, ∅).

In general, given the quadruple (u, T, U, R), the alphabet of the corresponding CSP process, *u_locked*, is defined by

$$\begin{aligned}\alpha u\_locked = {} & \{u' : \mathcal{U};\ v : \{f, l\} \mid u' = u \lor u' \in U \bullet subrouteState.u.v\} \\ & \cup \\ & \{t : \mathcal{T};\ v : \{c, o\} \mid t \in T \bullet circuitState.t.v\} \\ & \cup \\ & \{r : \mathcal{R} \mid r \in R \bullet routeState.r.req\} \cup \{r : \mathcal{R} \mid r \in R \bullet routeState.r.xs\}\end{aligned}$$

Initially,

$$\begin{aligned}& \forall\, t : \mathcal{T} \mid t \in T \bullet state_T(t) = c \\ \land\ & \forall\, u' : \mathcal{U} \mid u' \in U \bullet state_U(u') = l \\ \land\ & \forall\, r : \mathcal{R} \mid r \in R \bullet state_R(r) = xs\end{aligned}$$

Initially, *u_locked* performs a conditional check upon its variables. If

$$\begin{aligned}& \forall\, t : \mathcal{T} \mid t \in T \bullet value_T(var_T(t)) = c \\ \land\ & \forall\, u' : \mathcal{U} \mid u' \in U \bullet value_U(var_U(u')) = f \\ \land\ & \forall\, r : \mathcal{R} \mid r \in R \bullet value_R(var_R(r)) = xs\end{aligned}$$

then *subrouteState.u.f* occurs, and the process *u_free* is invoked. Alternatively, if the above does not hold, then the external choice between updates in state for all track circuits $t \in T$, sub-routes $u' \in U$ and routes $r \in R$ is offered.

The process *u_free* offers the external choice between updates in the state of all track circuits $t \in T$, sub-routes $u' \in U$, and routes $r \in R$. It also offers choice of an observation of *subrouteState.u.l*. If the latter is observed, then the process *u_locked* is invoked.

7.7.3 Points Free to Move Data

We think of points free to move data as a quadruple of the form

$$\mathcal{P} \times \mathbb{P}\,\mathcal{T} \times \mathbb{P}\,\mathcal{U} \times \mathbb{P}\,\mathcal{U}$$

consisting of the set of points with which we concerned, the berth circuits of the points, a set of sub-routes associated with the points' controlled free to go normal condition, and a set of sub-routes associated with the points' controlled free to go reverse condition.

For example, we represent the points free to move data given by

P201N TAB c UAB-AC f UAB-CA f
P201R TAB c UAB-BC f UAB-CB f

as the quadruple (P201, {TAB}, {UAB-AC, UAB-CA}, {UAB-BC, UAB-CB}).

In general, given the quadruple (p, T, U_1, U_2), the corresponding CSP process, p, is defined as the parallel composition

$$p \mathrel{\widehat{=}} p\_cfnTrue \parallel p\_cfrFalse$$

The alphabets of these processes are defined thus:

$$\begin{aligned}\alpha p\_cfnTrue = {} & \{b : Bool \bullet pointState.p.cfn.b\} \\ & \cup \\ & \{pos : \{cn, cr\} \bullet pointPosition.p.pos\} \\ & \cup \\ & \{t : \mathcal{T};\ v : \{c, o\} \mid t \in T \bullet circuitState.t.v\} \\ & \cup \\ & \{u : \mathcal{U};\ v : \{f, l\} \mid u \in U_1 \bullet subrouteState.u.v\}\end{aligned}$$

$$\begin{aligned}\alpha p\_cfrFalse = {} & \{b : Bool \bullet pointState.p.cfr.b\} \\ & \cup \\ & \{pos : \{cn, cr\} \bullet pointPosition.p.pos\} \\ & \cup \\ & \{t : \mathcal{T};\ v : \{c, o\} \mid t \in T \bullet circuitState.t.v\} \\ & \cup \\ & \{u : \mathcal{U};\ v : \{f, l\} \mid u \in U_2 \bullet subrouteState.u.v\}\end{aligned}$$

We map interlocking components to variables in the same manner as described previously.

Initially, the values of all variables representing track circuits are c, and the values of all variables representing sub-routes are l. If, at any stage, all relevant track circuits are clear and all relevant sub-routes are free, or the points become controlled normal (respectively, controlled reverse), then *pointState.p.cfn.true* (respectively, *pointState.p.cfr.true*) is observed, and the process *p_cfnTrue* (respectively, *p_cfrTrue*) is invoked.

The processes *p_cfnTrue* and *p_cfrTrue* offer the external choice of observations relevant to track circuits contained in the set T, as well as communications pertaining to

sub-routes contained in either U_1 or U_2. Changes in the position of a set of points may also be observed. If p enters a state where it is no longer controlled normal (respectively, controlled reverse) and it is no longer free to go normal (respectively, free to go reverse), then *pointState.p.cfn.false* (respectively, *pointState.p.cfr.false*) is observed, and the process *p_cfnFalse* (respectively, *p_cfrFalse*) is invoked.

Given the above outline, one might conclude that the translation of GDL constructs into CSP processes can be fully automated — an important step in increasing the confidence in our approach to the mechanical verification of SSI geographic data. This, together with a number of other suggestions for research into further automating our approach, are outlined in Chapter 8. We first give examples of how the translations described in this section might be implemented.

7.8 Modelling Conditional Checks

7.8.1 Route Setting Data

Recall the components with which the route setting data for R14 was concerned: points P202, and sub-routes UAD-AB, UAD-BA, UAE-AB and UAE-BA. As such, the alphabet of the CSP process representing this conditional check is given by

$$\begin{aligned}\alpha\text{R14} = \{ & \mathit{routeState}.\text{R14}.\mathit{req}, \mathit{routeState}.\text{R14}.\mathit{s},\\ & \mathit{pointPosition}.\text{P202}.\mathit{cn}, \mathit{subrouteState}.\text{UAD-BA}.\mathit{l}, \mathit{subrouteState}.\text{UAE-BA}.\mathit{l}\}\\ \cup\ & \{b : \mathit{Bool} \bullet \mathit{pointState}.\text{P202}.\mathit{cfn}.b\}\\ \cup\ & \{u : \mathcal{U};\ v : \{f, l\} \mid u \in \{\text{UAD-AB}, \text{UAE-AB}\} \bullet \mathit{subrouteState}.u.v\}\end{aligned}$$

Bearing in mind our assumptions regarding the initial state of an interlocking, we conclude that points P202 are initially controlled free to go normal, and that sub-routes UAE-AB and UAD-AB are both initially locked.

The process R14*true* represents the route setting data for route R14 when all of these conditions are met, i.e., points P202 are controlled free to go normal, and sub-routes UAD-AB and UAE-AB are both free.

$$\begin{aligned}\text{R14}\mathit{true} \mathrel{\hat{=}} (\ & \mathit{routeState}.\text{R14}.\mathit{req} \rightarrow\\ & \mathit{pointPosition}.\text{P202}.\mathit{cn} \rightarrow \mathit{subrouteState}.\text{UAD-BA}.\mathit{l} \rightarrow\\ & \mathit{subrouteState}.\text{UAE-BA}.\mathit{l} \rightarrow \mathit{routeState}.\text{R14}.\mathit{s} \rightarrow \text{R14}\mathit{true})\\ \Box\ & \mathit{pointState}.\text{P202}.\mathit{cfn}.\mathit{false} \rightarrow \text{R14}\mathit{false}(\mathit{false}, f, f)\\ \Box\ & \mathit{subrouteState}.\text{UAE-AB}.\mathit{l} \rightarrow \text{R14}\mathit{false}(\mathit{true}, l, f)\\ \Box\ & \mathit{subrouteState}.\text{UAD-AB}.\mathit{l} \rightarrow \text{R14}\mathit{false}(\mathit{true}, f, l)\end{aligned}$$

If there is a request for R14 in this state, then points P202 are locked in controlled normal position, and sub-routes UAD-BA and UAE-BA are both locked. Alternatively, if any of the

three conditions necessary for the setting of route R14 become false, then this request is blocked and the process behaves as R14*false*. Note how the above relates to the route setting data of Section 7.4 — *if* each of the relevant conditions are met, *then* the route may be set.

The process R14*false* is defined similarly, although a request for route R14 to be set is not allowed when the system is in this state. In this process, the value of variable x determines whether points P202 are controlled free to go normal. Similarly, the values of y and z represent whether or not sub-routes UAE-AB and UAD-AB are free respectively. For example, if the value of x is *true*, then points P202 are controlled free to go normal; if the value of y (respectively z) is f, then UAE-AB (respectively UAD-AB) is free. If all of these conditions are met, then the behaviour of this process becomes that of R14*true*.

$$
\begin{array}{l}
\text{R14}\mathit{false}(x, y, z) \mathrel{\hat{=}} \\
\textbf{if} \quad x = \mathit{true} \wedge y = f \wedge z = f \\
\textbf{then}\ \text{R14}\mathit{true} \\
\textbf{else}\ \mathit{pointState}.\text{P202}.\mathit{cfn}?\mathit{newState} \rightarrow \text{R14}\mathit{false}(\mathit{newState}, y, z) \\
\qquad \Box \\
\qquad \mathit{subrouteState}.\text{UAE-AB}?\mathit{newState} \rightarrow \text{R14}\mathit{false}(x, \mathit{newState}, z) \\
\qquad \Box \\
\qquad \mathit{subrouteState}.\text{UAD-AB}?\mathit{newState} \rightarrow \text{R14}\mathit{false}(x, y, \mathit{newState})
\end{array}
$$

Given the above, we define

$$\text{R14} \mathrel{\hat{=}} \text{R14}\mathit{false}(\mathit{true}, l, l)$$

7.8.2 Sub-route Release Data

Processes concerning sub-route release data are defined in a similar fashion to those associated with route setting data. The alphabet of the process representing sub-route release data for UAE-BA is given by

$$
\begin{array}{rl}
\alpha\text{UAE-BA} = & \{v : \{c, o\} \bullet \mathit{circuitState}.\text{TAE}.v\} \\
& \cup \\
& \{u : \mathcal{U};\ v : \{f, l\} \mid u \in \{\text{UAE-BA}, \text{UAD-BA}, \text{UAD-CA}\} \bullet \mathit{subrouteState}.u.v\}
\end{array}
$$

We assume that all sub-routes are initially locked and that all track circuits are initially clear.

$$\text{UAE-BA} \mathrel{\hat{=}} \text{UAE-BA}\mathit{locked}(c, l, l)$$

The process UAE-BA*locked* is defined thus:

$$
\begin{array}{l}
\text{UAE-BA}\mathit{locked}(x, y, z) \mathrel{\hat{=}} \\
\textbf{if} \quad x = c \wedge y = f \wedge z = f \\
\textbf{then}\ \mathit{subrouteState}.\text{UAE-BA}.f \rightarrow \text{UAE-BA}\mathit{free}(c, f, f) \\
\textbf{else}\ \mathit{circuitState}.\text{TAE}?\mathit{newState} \rightarrow \text{UAE-BA}\mathit{locked}(\mathit{newState}, y, z) \\
\qquad \Box \\
\qquad \mathit{subrouteState}.\text{UAD-BA}?\mathit{newState} \rightarrow \text{UAE-BA}\mathit{locked}(x, \mathit{newState}, z) \\
\qquad \Box \\
\qquad \mathit{subrouteState}.\text{UAD-CA}?\mathit{newState} \rightarrow \text{UAE-BA}\mathit{locked}(x, y, \mathit{newState})
\end{array}
$$

Here, the values of the variables x, y and z represent the states of track circuit TAE, and sub-routes UAD-BA and UAD-CA respectively. For example, the value of x is c if and only if TAE is clear. Sub-route UAE-BA becomes free after being locked if the value of x is c and the values of both y and z are f. If this is the case, then the process UAE-BA*free* is invoked. If this is not the case and there is any change in the state of track circuit TAE or sub-routes UAD-BA or UAD-CA, then the value of the appropriate variable is updated accordingly.

The conditional check on sub-route UAE-BA when it is free is represented by UAE-BA*free*. Again, the value of x represents the state of track circuit TAE, and the values of y and z represent the states of sub-routes UAD-BA and UAD-CA respectively. Note that UAE-BA can become locked at any time, at which point the process UAE-BA*locked* is invoked.

$$
\begin{aligned}
\text{UAE-BA}\mathit{free}(x, y, z) \mathrel{\widehat{=}} \; & \mathit{subrouteState}.\text{UAE-BA}.l \rightarrow \text{UAE-BA}\mathit{locked}(x, y, z) \\
& \Box \\
& \mathit{circuitState}.\text{TAE}?\mathit{newState} \rightarrow \text{UAE-BA}\mathit{free}(\mathit{newState}, y, z) \\
& \Box \\
& \mathit{subrouteState}.\text{UAD-BA}?\mathit{newState} \rightarrow \text{UAE-BA}\mathit{free}(x, \mathit{newState}, z) \\
& \Box \\
& \mathit{subrouteState}.\text{UAD-CA}?\mathit{newState} \rightarrow \text{UAE-BA}\mathit{free}(x, y, \mathit{newState})
\end{aligned}
$$

As we have seen, it is not always the case that conditionals for sub-route release are based upon the state of other sub-routes. In the case of the first sub-route of a route, sub-route release is conditional upon the state of that route. To illustrate this point, we take sub-route UAD-BA.

$$
\begin{aligned}
\alpha\text{UAD-BA}\mathit{locked}(x, y) = \; & \{v : \{c, o\} \bullet \mathit{circuitState}.\text{TAD}.v\} \\
& \cup \\
& \{v : \{f, l\} \bullet \mathit{subrouteState}.\text{UAD-BA}.v\} \\
& \cup \\
& \{v : \{\mathit{req}, \mathit{xs}\} \bullet \mathit{routeState}.\text{R14}.v\}
\end{aligned}
$$

Here, the relevant conditions are given by the states of track circuit TAD and route R14. Initially, the former is clear and the latter is unset.

$$
\begin{aligned}
& \text{UAD-BA}\mathit{locked}(x, y) \mathrel{\widehat{=}} \\
& \textbf{if} \quad x = c \land y = \mathit{xs} \\
& \textbf{then} \; \mathit{subrouteState}.\text{UAD-BA}.f \rightarrow \text{UAD-BA}\mathit{free}(c, \mathit{xs}) \\
& \textbf{else} \; \mathit{circuitState}.\text{TAD}?\mathit{newState} \rightarrow \text{UAD-BA}\mathit{locked}(\mathit{newState}, y) \\
& \qquad \Box \\
& \qquad \Box_{\mathit{newState} \in \{\mathit{req}, \mathit{xs}\}} \mathit{routeState}.\text{R14}.\mathit{newState} \rightarrow \text{UAD-BA}\mathit{locked}(x, \mathit{newState})
\end{aligned}
$$

Initially, UAD-BA can be freed. Following an occurrence of $\mathit{subrouteState}.\text{UAD-BA}.f$, the process UAD-BA*free* is invoked. This process is defined in the following way.

$$
\begin{aligned}
&\text{UAD-BA}\mathit{free}(x, y) \mathrel{\widehat{=}} \\
&\mathit{subrouteState}.\text{UAD-BA}.\mathit{l} \rightarrow \text{UAD-BA}\mathit{locked}(x, y) \\
&\Box \\
&\mathit{circuitState}.\text{TAD}?\mathit{newState} \rightarrow \text{UAD-BA}\mathit{free}(\mathit{newState}, y) \\
&\Box \\
&\Box_{\mathit{newState} \in \{\mathit{req}, \mathit{xs}\}}\ \mathit{routeState}.\text{R14}.\mathit{newState} \rightarrow \text{UAD-BA}\mathit{free}(x, \mathit{newState})
\end{aligned}
$$

Given the above, we define

$$\text{UAD-BA} \mathrel{\widehat{=}} \text{UAD-BA}\mathit{locked}(c, \mathit{xs})$$

7.8.3 Points Free to Move Data

There are two Boolean checks concerned with every set of points: one associated with whether the points are free to go normal and the other with whether they are free to go reverse. We represent points free to move data as the synchronisation of processes representing such checks. Rather than having separate processes concerned with the position of a set of points, as well as processes representing points free to move data, we combine both components, and define two processes — one related to whether a set of points is controlled free to go normal, and one related to whether it is controlled free to go reverse. We take points P201 as an example. Here, if P201 is controlled normal or free to go normal, then they are controlled free to go normal. Recall that this set of points is free to go normal if track circuit TAB is clear, and sub-routes UAB-AC and UAB-CA are both free. In the following, *pos* represents P201's position, while the variables x, y and z represent the states of track circuit TAB and sub-routes UAB-AC and UAB-CA respectively.

$$
\begin{aligned}
\alpha\text{P201}\mathit{cfn} = \ & \{v : \{c, o\} \bullet \mathit{circuitState}.\text{TAB}.v\} \\
& \cup \\
& \{\mathit{pos} : \{\mathit{cn}, \mathit{cr}\} \bullet \mathit{pointPosition}.\text{P201}.\mathit{pos}\} \\
& \cup \\
& \{b : \mathit{Bool} \bullet \mathit{pointState}.\text{P201}.\mathit{cfn}.b\} \\
& \cup \\
& \{u : \mathcal{U};\ v : \{f, l\} \mid u \in \{\text{UAB-AC}, \text{UAB-CA}\} \bullet \mathit{subrouteState}.u.v\}
\end{aligned}
$$

$$
\begin{aligned}
&\text{P201}\mathit{cfnTrue}(\mathit{pos}, x, y, z) \mathrel{\widehat{=}} \\
&\textbf{if}\quad \mathit{pos} \neq \mathit{cn} \wedge (x = o \vee y = l \vee z = l) \\
&\textbf{then}\ \mathit{pointState}.\text{P201}.\mathit{cfn}.\mathit{false} \rightarrow \text{P201}\mathit{cfnFalse}(\mathit{pos}, x, y, z) \\
&\textbf{else}\ \ \mathit{pointPosition}.\text{P201}?\mathit{newPos} \rightarrow \text{P201}\mathit{cfnTrue}(\mathit{newPos}, x, y, z) \\
&\qquad \Box \\
&\qquad \mathit{circuitState}.\text{TAB}?\mathit{newState} \rightarrow \text{P201}\mathit{cfnTrue}(\mathit{pos}, \mathit{newState}, y, z) \\
&\qquad \Box \\
&\qquad \mathit{subrouteState}.\text{UAB-AC}?\mathit{newState} \rightarrow \text{P201}\mathit{cfnTrue}(\mathit{pos}, x, \mathit{newState}, z) \\
&\qquad \Box \\
&\qquad \mathit{subrouteState}.\text{UAB-CA}?\mathit{newState} \rightarrow \text{P201}\mathit{cfnTrue}(\mathit{pos}, x, y, \mathit{newState})
\end{aligned}
$$

Alternatively, if this condition is contradicted, then P201*cfnFalse* is invoked.

$$
\begin{array}{l}
\text{P201}\mathit{cfnFalse}(\mathit{pos}, x, y, z) \mathrel{\hat{=}} \\
\textbf{if} \quad \mathit{pos} = \mathit{cn} \lor (x = c \land y = f \land z = f) \\
\textbf{then}\ \mathit{pointState}.\text{P201}.\mathit{cfn}.\mathit{true} \rightarrow \text{P201}\mathit{cfnTrue}(\mathit{pos}, x, y, z) \\
\textbf{else}\ \ \mathit{pointPosition}.\text{P201}?\mathit{newPos} \rightarrow \text{P201}\mathit{cfnFalse}(\mathit{newPos}, x, y, z) \\
\qquad \Box \\
\qquad \mathit{circuitState}.\text{TAB}?\mathit{newState} \rightarrow \text{P201}\mathit{cfnFalse}(\mathit{pos}, \mathit{newState}, y, z) \\
\qquad \Box \\
\qquad \mathit{subrouteState}.\text{UAB-AC}?\mathit{newState} \rightarrow \text{P201}\mathit{cfnFalse}(\mathit{pos}, x, \mathit{newState}, z) \\
\qquad \Box \\
\qquad \mathit{subrouteState}.\text{UAB-CA}?\mathit{newState} \rightarrow \text{P201}\mathit{cfnFalse}(\mathit{pos}, x, y, \mathit{newState})
\end{array}
$$

Only when P201 is controlled normal, or track circuit TAB is clear and sub-routes UAB-AC and UAB-CA are both free, will P201 again be controlled free to go normal.

The processes concerned with the controlled free to go reverse condition for points P201 are defined similarly.

$$
\begin{array}{rl}
\alpha\text{P201}\mathit{cfr} = & \{v : \{c, o\} \bullet \mathit{circuitState}.\text{TAB}.v\} \\
& \cup \\
& \{\mathit{pos} : \{\mathit{cn}, \mathit{cr}\} \bullet \mathit{pointPosition}.\text{P201}.\mathit{pos}\} \\
& \cup \\
& \{b : \mathit{Bool} \bullet \mathit{pointState}.\text{P201}.\mathit{cfr}.b\} \\
& \cup \\
& \{u : \mathcal{U};\ v : \{f, l\} \mid u \in \{\text{UAB-BC}, \text{UAB-CB}\} \bullet \mathit{subrouteState}.u.v\}
\end{array}
$$

In this case, the variable x represents the state of track circuit TAB, while y and z represent the states of sub-routes UAB-BC and UAB-CB respectively.

$$
\begin{array}{l}
\text{P201}\mathit{cfrTrue}(\mathit{pos}, x, y, z) \mathrel{\hat{=}} \\
\textbf{if} \quad \mathit{pos} \neq \mathit{cr} \land (x = o \lor y = l \lor z = l) \\
\textbf{then}\ \mathit{pointState}.\text{P201}.\mathit{cfr}.\mathit{false} \rightarrow \text{P201}\mathit{cfrFalse}(\mathit{pos}, x, y, z) \\
\textbf{else}\ \ \mathit{pointPosition}.\text{P201}?\mathit{newPos} \rightarrow \text{P201}\mathit{cfrTrue}(\mathit{newPos}, x, y, z) \\
\qquad \Box \\
\qquad \mathit{circuitState}.\text{TAB}?\mathit{newState} \rightarrow \text{P201}\mathit{cfrTrue}(\mathit{pos}, \mathit{newState}, y, z) \\
\qquad \Box \\
\qquad \mathit{subrouteState}.\text{UAB-BC}?\mathit{newState} \rightarrow \text{P201}\mathit{cfrTrue}(\mathit{pos}, x, \mathit{newState}, z) \\
\qquad \Box \\
\qquad \mathit{subrouteState}.\text{UAB-CB}?\mathit{newState} \rightarrow \text{P201}\mathit{cfrTrue}(\mathit{pos}, x, y, \mathit{newState})
\end{array}
$$

Here, we are concerned with the state of track circuit TAB, and whether sub-routes UAB-BC and UAB-CB are free or locked.

$$\begin{aligned}
&\text{P201}\mathit{cfrFalse}(\mathit{pos}, x, y, z) \mathrel{\hat{=}} \\
&\textbf{if} \quad \mathit{pos} = \mathit{cr} \lor (x = c \land y = f \land z = f) \\
&\textbf{then } \mathit{pointState}.\text{P201}.\mathit{cfr}.\mathit{true} \rightarrow \text{P201}\mathit{cfrTrue}(\mathit{pos}, x, y, z) \\
&\textbf{else } \mathit{pointPosition}.\text{P201}?\mathit{newPos} \rightarrow \text{P201}\mathit{cfrFalse}(\mathit{newPos}, x, y, z) \\
&\qquad \Box \\
&\qquad \mathit{circuitState}.\text{TAB}?\mathit{newState} \rightarrow \text{P201}\mathit{cfrFalse}(\mathit{pos}, \mathit{newState}, y, z) \\
&\qquad \Box \\
&\qquad \mathit{subrouteState}.\text{UAB-BC}?\mathit{newState} \rightarrow \text{P201}\mathit{cfrFalse}(\mathit{pos}, x, \mathit{newState}, y) \\
&\qquad \Box \\
&\qquad \mathit{subrouteState}.\text{UAB-CB}?\mathit{newState} \rightarrow \text{P201}\mathit{cfrFalse}(\mathit{pos}, x, y, \mathit{newState})
\end{aligned}$$

Thus, we represent the points free to move data for points P201 by

$$\text{P201} \mathrel{\hat{=}} \text{P201}\mathit{cfnTrue}(\mathit{cn}, c, l, l) \parallel \text{P201}\mathit{cfrFalse}(\mathit{cn}, c, l, l)$$

7.8.4 Further Processes

Our CSP representation of an interlocking's geographic data involves the composition of processes of the form described above. In addition, our assumption regarding the atomicity of route setting is enforced by the process *RC*.

$$RC(R) \mathrel{\hat{=}} \Box_{r \in R}\, \mathit{routeState}.r.\mathit{req} \rightarrow \mathit{routeState}.r.s \rightarrow RC(R)$$

This process ensures that, following a successful request for some route $r \in R$, no successful request for another route can occur until r is set.

We model the movement of points in such a way that points cannot move if they are locked. We define such a process thus:

$$\begin{aligned}
\mathit{Point}(p) \mathrel{\hat{=}}\ & \mathit{pointPosition}.p?\mathit{pos} \rightarrow \mathit{Point}(p) \\
& \Box \\
& \mathit{pointLocked}.p.\mathit{true} \rightarrow \mathit{pointLocked}.p.\mathit{false} \rightarrow \mathit{Point}(p)
\end{aligned}$$

It follows that we define

$$\mathit{Points} \mathrel{\hat{=}} |||_{p \in \mathcal{P}}\, \mathit{Point}(p)$$

We model track occupancy by the process *Train*. In each case, we are concerned with a particular route $r \in \mathcal{R}$, its component track circuits, given by $\mathit{circuit}^*(\mathit{subroutes}(r))$, and the set of points which are locked by r, given by $\mathit{locks}(r)$.

In the following, we denote $\mathit{circuit}^*(\mathit{subroutes}(r))$ by T and $\mathit{locks}(r)$ by P.

After the route has been requested, the relevant points are locked.

$$\mathit{Train}(r, T, P) \mathrel{\hat{=}} \mathit{routeState}.r.\mathit{req} \rightarrow \mathit{LockPoints}(r, T, P, P)$$

$$\begin{aligned}
&\mathit{LockPoints}(r, T, P, \mathit{toLock}) \mathrel{\hat{=}} \\
&\textbf{if} \quad \mathit{toLock} = \varnothing \\
&\textbf{then } \mathit{routeState}.r.s \rightarrow \mathit{circuitState}.\mathit{hd}(T).o \rightarrow \mathit{Train}'(r, T, \mathit{hd}(T), \mathit{tl}(T), P) \\
&\textbf{else } \Box_{p \in \mathit{toLock}}\, \mathit{pointLocked}.p.\mathit{true} \rightarrow \mathit{LockPoints}(r, T, P, \mathit{toLock} - \{p\})
\end{aligned}$$

Here, *hd* and *tl* are the standard head and tail operations on sequences.

As the train moves along the route, each track circuit becomes occupied in turn, until the train has passed through the route. At this point, the route becomes unset, and the points that have been locked for the route are released.

$$
\begin{aligned}
&Train'(r, T, occ, tail, P) \mathrel{\hat{=}} \\
&\quad \textbf{if} \quad tail = \langle\rangle \\
&\quad \textbf{then} \; circuitState.occ.c \rightarrow routeState.r.xs \rightarrow UnlockPoints(r, T, P, P) \\
&\quad \textbf{else} \; circuitState.hd(tail).o \rightarrow circuitState.occ.c \rightarrow Train'(r, T, hd(tail), tl(tail), P)
\end{aligned}
$$

$$
\begin{aligned}
&UnlockPoints(r, T, P, locked) \mathrel{\hat{=}} \\
&\quad \textbf{if} \quad locked = \varnothing \\
&\quad \textbf{then} \; Train(r, T, P) \\
&\quad \textbf{else} \; \Box_{p \in locked} \, pointLocked.p.false \rightarrow UnlockPoints(r, T, P, locked - \{p\})
\end{aligned}
$$

Given the above, we define

$$Trains \mathrel{\hat{=}} |||_{r \in \mathcal{R}} \, Train(r, circuit^*(subroutes(r)), locks(r))$$

Given an interlocking $\mathcal{I} = (\mathcal{T}, \mathcal{P}, S, \mathcal{R}, \mathcal{U})$, we denote the composition of the conditional checks on the geographic data pertaining to $\mathcal{I}$, together with $RC(\mathcal{R})$, *Points* and *Trains* by $\mathcal{I}_{\text{CSP}}$.

7.9 Verifying SSI Safety Invariants

In this section, we illustrate how our CSP representation of SSI geographic data of the previous section can be verified with respect to the safety invariants of Section 7.5.

7.9.1 Invariant 1

Recall from Section 7.5 that safety invariant 1 is stated thus:

> For every track circuit, no more than one of the sub-routes passing over it should be locked for a route at any time.

Recall that all sub-routes are initially locked; however, they are not *locked for routes*, i.e., their locking was not brought about as part of the process of setting a route. Thus, this invariant is not invalidated at initialisation as might appear at first observation. However, all communications of the form *subrouteState.u.l* after initialisation do form part of the route setting process. Therefore, we must ensure that all relevant observations of sub-routes being locked are separated by corresponding observations of these sub-routes being freed.

Assume a set of sub-routes $U \subseteq \mathcal{U}$. We define a condition on U, such that if any $u \in U$ becomes locked for a route, then u must become free before further elements of U can be locked in this fashion.

Initially, all the elements of U are locked. As such, we define

$$S_1(U) \mathrel{\hat{=}} S_1 Init(U, \varnothing)$$

where

$$
\begin{array}{l}
S_1 Init(Locked, Free) \mathrel{\widehat{=}} \\
\textbf{if} \quad Locked = \varnothing \\
\textbf{then}\ S_1 AllFree(Free) \\
\textbf{else}\ \Box_{u \in Locked}\, subrouteState.u.f \rightarrow S_1 Init(Locked - \{u\}, Free \cup \{u\}) \\
\qquad \Box \\
\qquad \Box_{u \in Free}\, subrouteState.u.l \rightarrow S_1'(Locked, Free - \{u\}, u)
\end{array}
$$

If at any stage the value of *Locked* is the empty set, then the process $S_1 AllFree$ is invoked. On the other hand, if there are sub-routes which are locked, then there is a choice between a locked sub-route becoming free, and a free sub-route becoming locked. If a sub-route is locked for a route, then it must be freed before another can be locked in this manner.

$$
\begin{array}{l}
S_1'(Locked, Free, u) \mathrel{\widehat{=}} \\
\qquad subrouteState.u.f \rightarrow S_1 Init(Locked, Free \cup \{u\}) \\
\qquad \Box \\
\qquad \Box_{u' \in Locked}\, subrouteState.u'.f \rightarrow S_1'(Locked - \{u'\}, Free \cup \{u'\}, u)
\end{array}
$$

When all sub-routes have been released, the invariant states that the locking and freeing of sub-routes from U must alternate.

$$S_1 AllFree(U) \mathrel{\widehat{=}} \Box_{u \in U}\, subrouteState.u.l \rightarrow subrouteState.u.f \rightarrow S_1 AllFree(U)$$

Take, as an example, track circuit TAK. Here,

$$sr(\text{TAK}) = \{\text{UAK-AB}, \text{UAK-BA}\}$$

Recalling the convention for naming sub-routes, UAK-AB runs from track circuit TAA in the direction of track circuit TAJ, while UAK-BA runs in the opposite direction. Thus, safety invariant 1 for track circuit TAK states that UAK-AB and UAK-BA should not be locked for routes simultaneously.

We can now define what we mean for an interlocking to satisfy safety invariant 1.

Definition 7.1 Safety invariant 1 holds of an interlocking $\mathcal{I} = (\mathcal{T}, \mathcal{P}, S, \mathcal{R}, \mathcal{U})$ if and only if

$$\forall\, t : \mathcal{T} \bullet \mathcal{I}_{\text{CSP}}\ \textbf{imp}\ S_1(sr(t))$$

□

Our representation of the Open ALVEY interlocking consists of several million possible configurations — there are 4 sets of points, each of which may be controlled free to go normal, may be controlled free to go reverse, and may be in one of two positions; 30 sub-routes, each of which may be free or locked; 12 track circuits, each of which may be clear or occupied; and 16 routes, each of which may be requested, set or unset. Despite the fact that many of these configurations are impossible, it can be seen that the problem of verifying SSI geographic data, even for such a simple interlocking, is one of enormous complexity. As such, we look to reduce the size of the problem at hand.

Recall from Section 7.4 that a sub-route becomes locked when a route passing over it is set. Given a track circuit $t \in \mathcal{T}$, we consider the route setting data for those routes which may lock sub-routes contained in $sr(t)$.

Referring to Figure 7.1, four routes travel over track circuit TAK: routes R13, R15, R22 and R24. Referring to Appendix B, the route setting data for these routes is as follows.

Q13 **if** P201 cfr UAA-BA f UAB-CA f
then R13 s P201 cr UAB-AC l UAA-AB l UAK-AB l

Q15 **if** P201 cfn UAA-BA f UAB-CB f
then R15 s P201 cn UAB-BC l UAA-AB l UAK-AB l

Q22 **if** P204 cfr UAJ-CB f UAK-AB f
then R22 s P204 cr UAJ-BC l UAK-BA l UAA-BA l

Q24 **if** P204 cfn UAJ-CA f UAK-AB f
then R24 s P204 cn UAJ-AC l UAK-BA l UAA-BA l

Referring to the above, only routes R22 and R24 can lock sub-route UAK-BA. Observing the conditions for these routes to be set, we see that before this sub-route can be locked by either route, sub-route UAK-AB must be free. Therefore, we need take no other processes into account to ensure that UAK-AB and *then* UAK-BA cannot be locked — if sub-route UAK-AB is locked, then neither route R22 nor route R24 can be set, and, therefore, sub-route UAK-BA cannot be locked.

Examining the conditions for routes R13 and R15 to be set, we see that before either route can be set, sub-route UAA-BA must be free. Referring to Appendix B, the sub-route release data for UAA-BA is given by

UAA-BA f **if** TAA c UAK-BA f

Therefore, we need only take into account processes representing route setting data for routes R13 and R15, and sub-route release data for sub-route UAA-BA to ensure that if UAK-AB is locked then UAK-BA cannot be locked — if sub-route UAK-BA is locked, then UAA-BA cannot be freed, and, therefore, neither route R13 nor route R15 can be set.

In this case, only five processes need to be considered to ensure that safety invariant 1 holds for track circuit TAK. The justification for this is based upon the fact that the events with which we are concerned (the locking of sub-routes over track circuit TAK) can only ever occur in $\mathcal{I}_{CSP}$ with the co-operation of processes representing route setting data for routes R13, R15, R22 and R24. Composition with further processes will only serve to reduce the set of possible behaviours for these components, while introducing further independent components, thus expanding the state space of the check to be performed.

Given our approach to constructing CSP processes corresponding to SSI conditional checks, we define the process

$$\begin{aligned}\text{TAK}imp \mathrel{\hat{=}} \quad & (RC(\{\text{R13}, \text{R15}, \text{R22}, \text{R24}\}) \\ & \| \\ & ((\text{R13} \,[\![\, \{subrouteState.\text{UAA-BA}.*\} \,]\!]\, \text{R15}) \\ & \| \\ & (\text{R22} \,[\![\, \{subrouteState.\text{UAK-AB}.*\} \,]\!]\, \text{R24}))) \\ & \| \\ & \text{UAA-BA}\end{aligned}$$

If TAK*imp* **imp** $S_1(sr(\text{TAK}))$, then we can claim that safety invariant 1 holds of $\mathcal{I}_{\text{CSP}}$ for track circuit TAK because all relevant behaviours of the sub-routes contained in the set $sr(\text{TAK})$ which are of consequence to this safety invariant are observed by TAK*imp*.

As a further example, routes R11A, R11B, R12 and R14 can set sub-routes over track circuit TAD. Referring to Appendix B, the conditional checks representing route setting data for these routes are as follows.

Q11A **if** P202 cfn UAD-BA f UAC-BA f
then R11A s P202 cn UAD-AB l UAC-AB l

Q11B **if** P202 cfr UAD-CA f UBA-BA f
then R11B s P202 cr UAD-AC l UBA-AB l

Q12 **if** P202 cfr UAE-AB f UAD-AC f
then R12 s P202 cr UAD-CA l UAE-BA l

Q14 **if** P202 cfn UAE-AB f UAD-AB f
then R14 s P202 cn UAD-BA l UAE-BA l

Given the above, it is apparent that the setting of routes which lock sub-routes over track circuit TAD are dependent upon points P202 being free to move in the relevant direction. Recalling from Section 7.7 the construction of alphabets for our processes, we define the process TAD*imp* thus:

$$\begin{aligned}\text{TAD}imp \mathrel{\hat{=}} \quad & (RC(\{\text{R11A}, \text{R11B}, \text{R12}, \text{R14}\}) \\ & \| \\ & ((\text{R11A} \,\|\, \text{R11B}) \\ & \| \\ & (\text{R12} \,[\![\, \{subrouteState.\text{UAE-AB}.*\} \,]\!]\, \text{R14}))) \\ & \| \\ & \text{P202}\end{aligned}$$

In the above examples, we term the processes UAA-BA and P202 the *offset* for TAK*imp* and TAD*imp* respectively. In addition, we term TAK*imp* and TAD*imp* *extractions* for TAK and TAD respectively.

In the following, we let $\gamma_{\mathcal{I}}(sr(t))$ denote the extraction from $\mathcal{I}_{CSP}$ of those processes associated with the setting of sub-routes over some track circuit $t \in \mathcal{T}$.

The following proposition formalises our intuition that if safety invariant holds of the relevant extraction, then it holds of the interlocking as a whole.

Proposition 7.1 Given an interlocking $\mathcal{I} = (\mathcal{T}, \mathcal{P}, S, \mathcal{R}, \mathcal{U})$,

$$\forall\, t : \mathcal{T} \bullet \gamma_{\mathcal{I}}(sr(t)) \textbf{ imp } S_1(sr(t)) \Rightarrow \mathcal{I}_{\mathrm{CSP}} \textbf{ imp } S_1(sr(t))$$

Proof. By virtue of the fact that all behaviours relevant to the analysis of safety invariant 1 for t are observable in $\gamma_{\mathcal{I}}(sr(t))$. □

Synchronisation

Some care is needed in determining which events that the processes involved in extractions should synchronise upon. For example, in $\gamma_{\mathcal{I}}(sr(\mathrm{TAK}))$, the alphabets of the processes representing route setting data for routes R13 and R15 have four events in common: the event associated with the release of sub-route UAA-BA, and those events associated with the locking of UAA-BA, UAA-AB and UAK-AB.

These processes must synchronise upon communications occurring on the channel *subrouteState*.UAA-BA — the state of this sub-route plays a part in determining whether routes R13 and R15 can be set, so it is essential that the two process's views of this sub-route's state are identical. On the other hand, they do not synchronise on the locking of UAA-AB and UAK-AB. This is because the locking of sub-routes UAA-AB and UAK-AB for route R13 is of no consequence to the route setting data for route R15; similarly the locking of these sub-routes for route R15 is irrelevant to the route setting data for R13. As such, there is no need for the processes representing route setting data for these routes to synchronise on the communications *subrouteState*.UAA-AB.*l* and *subrouteState*.UAK-AB.*l*.

As a further example, the alphabets of processes R22 and R24 have the events representing the release of sub-route UAK-AB, and the locking of UAK-AB, UAK-BA and UAA-BA in common. For similar reasons to those outlined above, these processes synchronise on communications on the channel *subrouteState*.UAK-AB only.

Synchronisation between the two pairs of routes is slightly more complicated. Routes R13 and R15 are both concerned with the state of sub-route UAA-BA. As routes R22 and R24 both lock UAA-BA, these pairs of processes must synchronise on *subrouteState*.UAA-BA.*l*. Similarly, routes R22 and R24 are both concerned with the state of sub-route UAK-AB. As routes R13 and R15 both lock sub-route UAK-AB, the pair of processes representing route setting data for these routes should synchronise on *subrouteState*.UAK-AB.*l*. These are the *only* events upon which the two pairs of routes synchronise.

Further composition with $RC(\{\mathrm{R13}, \mathrm{R15}, \mathrm{R22}, \mathrm{R24}\})$ and UAA-BA provides us with the process against which we verify safety invariant 1 for track circuit TAK.[4]

The procedure for verifying safety invariant 1 for an interlocking is given by the following algorithm.

[4]Note that the latest version of FDR2, when given processes (together with their observable alphabets) to be composed, is capable of calculating the most efficient synchronisation strategy in terms of state size.

Algorithm 7.1

1. Define $\mathcal{T}$, the set of track circuits.
2. Select an unmarked $t \in \mathcal{T}$ and generate the set $sr(t)$.
3. Select an unmarked pair $u, u' \in sr(t)$ such that $u \neq u'$ and generate the sets

$$R_u = \{r : \mathcal{R} \mid u \in \mathbf{elems}(subroutes(r))\}$$
$$R_{u'} = \{r : \mathcal{R} \mid u' \in \mathbf{elems}(subroutes(r))\}$$

4. Select an unmarked $r \in R_u$.
5. If r being set is conditional upon u' being free, then mark r and proceed to step 7; otherwise, proceed to step 6.
6. Unravel the conditions associated with r being set until **either** one is dependent upon u' being free, in which case note the conditional checks which led to this condition, mark r and proceed to step 7; **or** no such condition is found, in which case, the data is incomplete.
7. If every $r \in R_u$ has been checked, then repeat steps 4-6 with $R_{u'}$ replacing R_u and u replacing u'; otherwise, return to step 4.
8. Mark (u, u'). If all such pairs have been marked for $sr(t)$, then proceed to step 9; otherwise, return to step 3.
9. Given the set

$$R = \{r : \mathcal{R} \mid \exists\, u \in sr(t) \bullet u \in \mathbf{elems}(subroutes(r))\}$$

let $\gamma_1(sr(t))$ denote the composition of $RC(R)$ and the processes encountered by steps 3-8.

10. If $\gamma_1(sr(t))$ **imp** $S_1(sr(t))$, then mark t; if not, the data is inconsistent.
11. If all $t \in \mathcal{T}$ have been marked, then $\mathcal{I}$ is correct with respect to safety invariant 1; otherwise, return to step 2.

□

As an example of the application of Algorithm 7.1, the unravelling of conditions for the route setting data for route R13 is illustrated by Figure 7.3. This route can lock sub-routes UAB-AC, UAA-AB and UAK-AB, so we need to ensure that none of UAB-CA, UAB-BC, UAB-CB, UAA-BA and UAK-BA have been locked by another route prior to this route being set. Referring to Appendix B, the setting of route R13 is dependent upon points P201 being controlled free to go reverse, and sub-routes UAA-BA and UAB-CA both being free. Referring to Appendix B, P201 being controlled free to go reverse is dependent upon UAB-BC and UAB-CB both being free, and UAK-BA being free is dependent upon UAA-BA being free.

It follows that route R13 can be set only if the offending sub-routes are all free, i.e., the conditional check associated with the setting of this route ensures that route R13 cannot be set if any of UAB-CA, UAB-BC, UAB-CB, UAA-BA or UAK-BA are already locked.

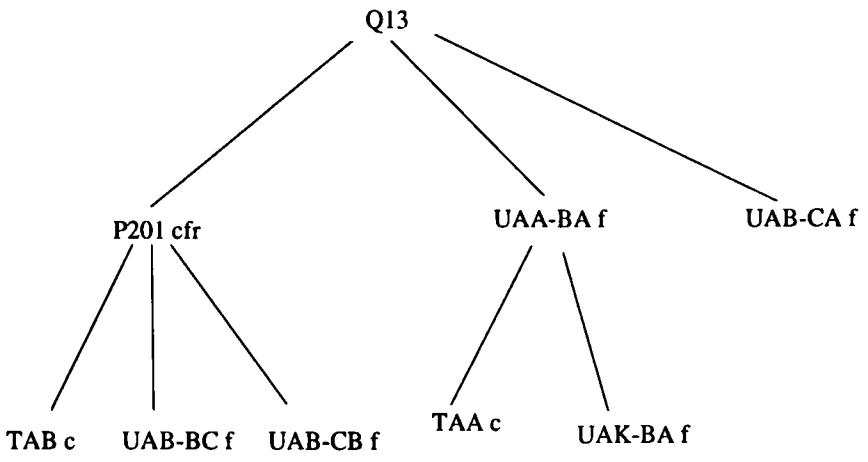

Figure 7.3: Conditions for setting route R13.

Alternation

Note that in the above extractions, we did not consider track occupancy — it may be possible in such extractions to observe consecutive occurrences of *circuitState.t.o* for some track circuit $t \in \mathcal{T}$, with no separating observation of *circuitState.t.c*.

We may, therefore, introduce processes such as *ALTcircuit*, which is defined by

$$ALTcircuit(t) \mathrel{\widehat{=}} circuitState.t.o \rightarrow circuitState.t.c \rightarrow ALTcircuit(t)$$

as a means of reducing complexity. Of course, other processes such as *ALTsubroute* and *ALTpointState* might be similarly defined.

Complexity

Checking invariant 1 for individual track circuits for a simple interlocking such as the Open ALVEY can be carried out in a matter of seconds with FDR, with the complexity of checks ranging in size from only 232 states (for track circuits TAC, TAH, TBA and TCA), to 14592 states (for track circuits TAA and TAK). The complexity of checking safety invariant 1 for the Open ALVEY is summarised in Table D.1, which, along with all of the other tables referred to in this chapter, can be found in Appendix D. Note that verifying this safety invariant for track circuit TAK involves a check of the same size as that required to verify its 'mirror image' circuit, TAA; verifying this invariant for our other example track circuit, TAD, involves checking 1232 states.

Recall that the extraction $\gamma_1(sr(\text{TAK}))$ includes processes representing route setting data for R13, R15, R22 and R24, each of which consists of 13 states. In addition, the offset for this extraction, UAA-BA*locked*(c, l), has 8 possible states. As for our other example, the extraction $\gamma_1(sr(\text{TAD}))$ includes the processes R11A, R11B, R12 and R14, each of which have 12 possible states. In this case, the offset is P202, which has 368 possible states.

The difference in complexity between processes representing route setting data is explained in part by the fact that setting one of R13, R15, R22 or R24 involves locking three sub-routes, while setting one of R11A, R11B, R12 or R14 involves locking two sub-routes. Furthermore, looking at $\gamma_1(sr(\text{TAD}))$, there is greater synchronisation between its constituent

processes than in the case of $\gamma_1(sr(\text{TAK}))$ — the latter involves several independent events. Thus, the main reason for the smaller size of our check on track circuit TAD is that *all* of the events of the offset of $\gamma_1(sr(\text{TAD}))$ synchronise with the route setting part, thus reducing the set of possible behaviours for those events involved in the latter. On the other hand, there are independent events in the offset of $\gamma_1(sr(\text{TAK}))$, which results in a check of greater complexity.

The verification of this invariant for other track circuits follows similar patterns. Points P201 (respectively P203 and P204) provide the offset for circuit TAB (respectively TAG and TAJ), and the size of checks for these are of a similar order as those for TAD.

Referring to Appendix B, we see that checks on track circuits TAC, TAH, TBA and TCA require no offset at all, and, as such, are of a relatively small size. As an example, only two routes lock sub-routes over track circuit TAC: R10B and R11A. Referring to Appendix B, the route setting data for these routes is as follows.

R10B **if** P201 cfn UAB-BC f UAC-AB f
then R10B s P201 cn UAB-CB l UAC-BA l

R11A **if** P202 cfn UAD-BA f UAC-BA f
then R11A s P202 cn UAD-AB l UAC-AB l

Given this route setting data, by Algorithm 7.1,

$$\gamma_1(sr(\text{TAC})) \mathrel{\hat{=}} RC(\{\text{R10B}, \text{R11A}\}) \parallel \text{R10B} \parallel \text{R11A}$$

Reducing Complexity

As well as incorporating processes such as *ALTcircuit*, there are a number of ways of further reducing the sizes of our checks. One such method is to improve our extraction algorithm, so that it provides the optimum offset. This issue is discussed further in Chapter 8.

A further method of reducing complexity involves concealing those events of an extraction which are not relevant to the safety invariant in question. For example, we might redefine $\gamma_1(sr(\text{TAK}))$ as

$$\begin{aligned}\gamma_1(sr(\text{TAK})) \mathrel{\hat{=}} \quad & (RC(\{\text{R13}, \text{R15}, \text{R22}, \text{R24}\}) \\ & \parallel \\ & ((\text{R13}' \,[\![\, \{subrouteState.\text{UAA-BA}.*\} \,]\!]\, \text{R15}') \\ & \parallel \\ & (\text{R22}' \,[\![\, \{subrouteState.\text{UAK-AB}.*\} \,]\!]\, \text{R24}'))) \\ & \parallel \\ & \text{UAA-BA}'\end{aligned}$$

We define R13′, R15′, R22′, R24′ and UAA-BA′ thus:

$$\begin{aligned}\text{R13}' \mathrel{\hat{=}} \text{R13} \setminus \{ & pointPosition.\text{P201}.cr, \\ & subrouteState.\text{UAB-AC}.l, subrouteState.\text{UAA-AB}.l\}\end{aligned}$$

$$\text{R15}' \mathrel{\widehat{=}} \text{R15} \setminus \{ \textit{pointPosition}.\text{P201}.\textit{cn}, \textit{subroutestate}.\text{UAB-BC}.\textit{l}, \textit{subrouteState}.\text{UAA-AB}.\textit{l}\}$$

$$\text{R22}' \mathrel{\widehat{=}} \text{R22} \setminus \{\textit{pointPosition}.\text{P204}.\textit{cr}, \textit{subrouteState}.\text{UAJ-BC}.\textit{l}\}$$

$$\text{R24}' \mathrel{\widehat{=}} \text{R24} \setminus \{\textit{pointPosition}.\text{P204}.\textit{cn}, \textit{subrouteState}.\text{UAJ-AC}.\textit{l}\}$$

$$\text{UAA-BA}' \mathrel{\widehat{=}} \text{UAA-BA} \setminus \{\textit{circuitState}.\text{UAA-BA}.*\}$$

The justification for concealing these communications comes from the fact that they are non-interfering with the conditions relating to the route setting or sub-route release of the routes and sub-routes with which this process is concerned.

A further method of reducing complexity is to place those communications which update states but are not directly relevant to an invariant in parallel with *Stop*. For example, we might redefine R13′ as

$$\text{R13}' \mathrel{\widehat{=}}$$
$$\text{R13}\textit{false}(\textit{true}, \textit{l}, \textit{f}) \,[\![\, \{\textit{pointState}.\text{P201}.\textit{cfr}.*, \textit{subrouteState}.\text{UAB-CA}.*\} \,]\!]\, \textit{Stop}$$
$$\setminus \{\textit{pointPosition}.\text{P201}.\textit{cr}, \textit{subrouteState}.\text{UAB-AC}.\textit{l}, \textit{subrouteState}.\text{UAA-AB}.\textit{l}\}$$

Again, these communications are non-interfering with the conditions essential to the analysis of the invariant. Note that if these events were concealed rather than 'blocked', then the possibility of divergence would be introduced into R13′. Note also how we change the initial state of such conditions so that we do not restrict the liveness of our extraction — if the initial state of UAB-CA was locked and we are no longer capable of observing communications associated with this sub-route, then R13 could never be set. As such, if we were to block communications on these channels, then the initial states of points P201 being controlled free to go reverse and sub-route UAB-CA must be true and free respectively.

Another Interlocking

As well as verifying properties of our Open ALVEY interlocking of Figure 7.1, we have also verified our approach with the Closed ALVEY interlocking of Figure 7.4, the geographic data for which is given in Appendix C.

The complexity of checking safety invariant 1 for this interlocking's track circuits is summarised in Table D.2.

Mutual Exclusion

We can further reduce the complexity of verifying safety invariant 1 for track circuits in the following way. Consider some track circuit $t \in \mathcal{T}$ for an interlocking $\mathcal{I} = (\mathcal{T}, \mathcal{P}, S, \mathcal{R}, \mathcal{U})$. If we can prove that, for every pair of sub-routes $u, u' \in sr(t)$, u and u' cannot be locked simultaneously, then it follows that no more than one of $sr(t)$ can be locked simultaneously. We state this formally in the following way.

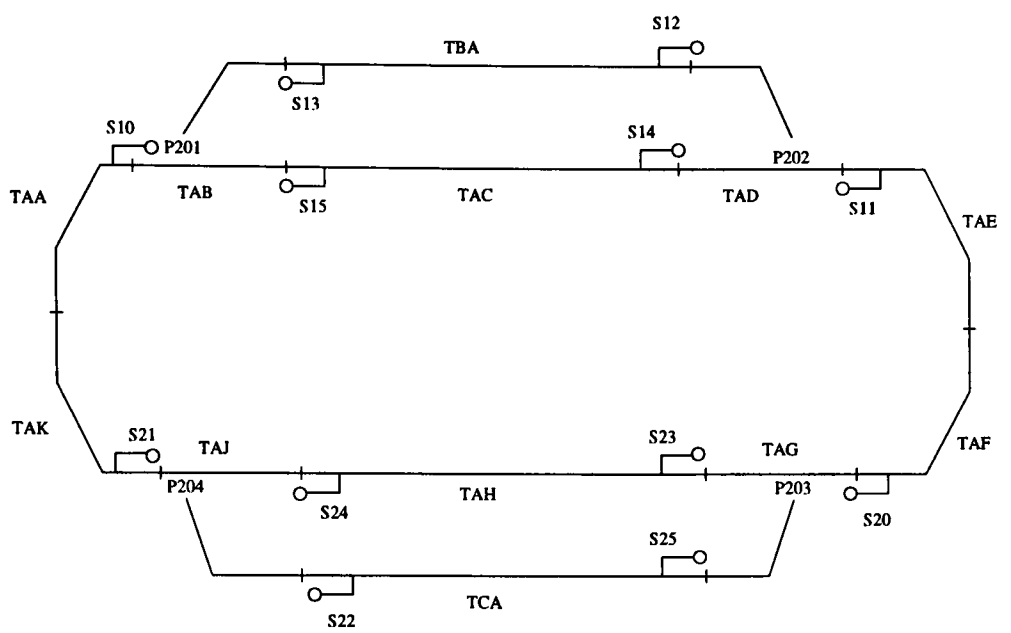

Figure 7.4: The Closed ALVEY interlocking.

Proposition 7.2 Given an interlocking $\mathcal{I} = (\mathcal{T}, \mathcal{P}, S, \mathcal{R}, \mathcal{U})$,

$$\forall\, t : \mathcal{T} \bullet$$
$$(\forall\, u, u' : \mathcal{U} \mid u \in sr(t) \wedge u' \in sr(t) \wedge u \neq u' \bullet \gamma_{\mathcal{I}}(\{u, u'\})\ \mathbf{imp}\ S_1(\{u, u'\})) \Rightarrow$$
$$\gamma_{\mathcal{I}}(sr(t))\ \mathbf{imp}\ S_1(sr(t))$$

Proof. Assume

$$\forall\, u, u' : \mathcal{U} \mid u \in sr(t) \wedge u' \in sr(t) \wedge u \neq u' \bullet \gamma_{\mathcal{I}}(\{u, u'\})\ \mathbf{imp}\ S_1(\{u, u'\})$$

Given any sub-route $u \in sr(t)$, our assumption states that no other sub-route $u' \in sr(t)$ such that $u' \neq u$ can be locked when u is locked. As such,

$$\gamma_{\mathcal{I}}(sr(t))\ \mathbf{imp}\ S_1(sr(t))$$

□

Given the above proposition, the verification of safety invariant 1 for a track circuit reduces to analysing pairs of sub-routes over that track circuit. For example, in the case of track circuit TAD,

$$(\forall\, u, u' \in \{\text{UAD-AB}, \text{UAD-AC}, \text{UAD-BA}, \text{UAD-CA}\} \mid u \neq u' \bullet$$
$$\gamma_{\mathcal{I}}(\{u, u'\})\ \mathbf{imp}\ S_1(\{u, u'\})) \Rightarrow \gamma_{\mathcal{I}}(sr(\text{TAD}))\ \mathbf{imp}\ S_1(sr(\text{TAD}))$$

The complexity of verifying track circuit TAD in this fashion is summarised in Table D.3.

We can further reduce the complexity of verifying safety invariant 1 by reducing our checks from an exclusion property on pairs of sub-routes to an exclusion property on pairs of routes which run over these sub-routes.

We first define a new extraction algorithm, which allows us to identify the components we need to take into account to verify that two routes running over the same track circuit cannot be set simultaneously.

Algorithm 7.2

1. Given any unmarked track circuit $t \in \mathcal{T}$, define the set

$$R_t = \{r : \mathcal{R} \mid \exists\, u : \mathcal{U} \bullet u \in sr(t) \land u \in \mathbf{elems}(subroutes(r))\}$$

2. Take any unmarked pair $r, r' \in R_t$ such that $r \neq r'$.
3. By the very nature of interlockings, there are unique elements $u, u' \in \mathcal{U}$ such that

$$\begin{aligned} & u \neq u' \\ \land\ & u \in sr(t) \land u \in \mathbf{elems}(subroutes(r)) \\ \land\ & u' \in sr(t) \land u' \in \mathbf{elems}(subroutes(r')) \end{aligned}$$

4. If r being set is dependent upon u' being free, then include the processes representing route setting data for r and sub-route release data for u', and proceed to the next step; if not, unravel the conditions for this route to be set in the manner described in Algorithm 7.1. If, after following this path, no condition dependent on u' being free is found, then the data is incomplete; otherwise include the conditional checks encountered along the shortest path.
5. Repeat step 4, replacing r' for r and u for u'.
6. Mark r and r', and denote the composition of $RC(r, r')$ and the processes encountered by steps 4 and 5 by $\delta_1(t, r, r')$.
7. If all pairs have been checked, then mark t and proceed to step 8, otherwise return to step 2.
8. If all track circuits have been marked, then the data is correct; otherwise return to step 1.

□

Given the above, we restate safety invariant 1 in terms of mutual exclusion.

Proposition 7.3 Given an interlocking $\mathcal{I} = (\mathcal{T}, \mathcal{P}, S, \mathcal{R}, \mathcal{U})$, a track circuit $t \in \mathcal{T}$, and sub-routes $u, u' \in sr(t)$ such that $u \neq u'$, define the sets

$$\begin{aligned} R &= \{r : \mathcal{R} \mid u \in \mathbf{elems}(subroutes(r))\} \\ R' &= \{r' : \mathcal{R} \mid u' \in \mathbf{elems}(subroutes(r'))\} \end{aligned}$$

which are the sets of routes which can lock u and u' respectively. It follows that

$$(\forall\, r \in R;\ r' \in R' \bullet \delta_1(t, r, r')\ \mathbf{imp}\ S_1(\{u, u'\})) \Rightarrow \gamma_1(\{u, u'\})\ \mathbf{imp}\ S_1(\{u, u'\})$$

Proof. Assume

$$\forall\, r \in R;\ r' \in R' \bullet \delta_1(t, r, r')\ \mathbf{imp}\ S_1(\{u, u'\})$$

Take any route $r \in R$. Our assumption states that no route $r' \in R'$ can be set when r is. Therefore, u' cannot be locked when u is. Similarly, whenever any $r' \in R'$ is locked, our assumption states that u is free. Therefore,

$$\gamma_{\mathcal{I}}(\{u, u'\}) \textbf{ imp } S_1(\{u, u'\})$$

□

Finally, safety invariant 1 holds of an interlocking $\mathcal{I} = (\mathcal{T}, \mathcal{P}, S, \mathcal{R}, \mathcal{U})$ if and only if, for every track circuit $t \in \mathcal{T}$, no two routes can simultaneously be set over t. In other words, that the setting of all such routes is mutually exclusive.

Proposition 7.4 Given an interlocking $\mathcal{I} = (\mathcal{T}, \mathcal{P}, S, \mathcal{R}, \mathcal{U})$, if

$$\begin{array}{l} \forall\, t : \mathcal{T};\ u_1, u_2 : \mathcal{U};\ r_1, r_2 : \mathcal{R} \mid \\ u_1 \in sr(t) \land u_2 \in sr(t) \land u_1 \neq u_2 \land \\ r_1 \in \{r : \mathcal{R} \mid u_1 \in \textbf{elems}(subroutes(r))\} \land \\ r_2 \in \{r : \mathcal{R} \mid u_2 \in \textbf{elems}(subroutes(r))\} \bullet \delta_{\mathcal{I}}(t, r_1, r_2) \textbf{ imp } S_1(\{u_1, u_2\}) \end{array}$$

then safety invariant 1 holds of $\mathcal{I}$.

Proof. Assume

$$\begin{array}{l} \forall\, t : \mathcal{T};\ u_1, u_2 : \mathcal{U};\ r_1, r_2 : \mathcal{R} \mid \\ u_1 \in sr(t) \land u_2 \in sr(t) \land u_1 \neq u_2 \land \\ r_1 \in \{r : \mathcal{R} \mid u_1 \in \textbf{elems}(subroutes(r))\} \land \\ r_2 \in \{r : \mathcal{R} \mid u_2 \in \textbf{elems}(subroutes(r))\} \bullet \delta_{\mathcal{I}}(t, r_1, r_2) \textbf{ imp } S_1(\{u_1, u_2\}) \end{array}$$

By Proposition 7.3,

$$\forall\, t : \mathcal{T};\ u_1, u_2 : \mathcal{U} \mid u_1 \in sr(t) \land u_2 \in sr(t) \land u_1 \neq u_2 \bullet \gamma_{\mathcal{I}}(\{u_1, u_2\}) \textbf{ imp } S_1(\{u_1, u_2\})$$

By Proposition 7.2,

$$\forall\, t : \mathcal{T} \bullet \gamma_{\mathcal{I}}(sr(t)) \textbf{ imp } S_1(sr(t))$$

By Proposition 7.1,

$$\forall\, t : \mathcal{T} \bullet \mathcal{I}_{\mathrm{CSP}} \textbf{ imp } S_1(sr(t))$$

Finally, by Definition 7.1, safety invariant 1 holds of $\mathcal{I}$. □

The complexity of checks carried out in the above fashion for track circuit TAK is summarised in Table D.4.

7.9.2 Invariant 2

Recall from Section 7.5 that safety invariant 2 is stated thus:

> If a sub-route over a track section containing points is locked for a route, then the points are correctly aligned with that sub-route.

In the following, we apply mutual exclusion, together with an assumption of the satisfaction of safety invariant 1, as a means of analysing this invariant.

We start by defining a CSP representation of this safety invariant. Our requirement is concerned with four parameters — a given sub-route, a given set of points, the points' current position, and the points' required position. If the points are in their required position, then the sub-route can be locked. We place no other restrictions upon the behaviour of the sub-route or the points. As such, we define the following process.

$$
\begin{array}{l}
S_2(u, p, pos, reqPos) \mathrel{\widehat{=}} \\
\textbf{if} \quad pos = reqPos \\
\textbf{then} \; subrouteState.u.l \rightarrow subrouteState.u.f \rightarrow S_2(u, p, pos, reqPos) \\
\qquad \Box \\
\qquad subrouteState.u.f \rightarrow S_2(u, p, pos, reqPos) \\
\qquad \Box \\
\qquad pointPosition.p?newPos \rightarrow S_2(u, p, newPos, reqPos) \\
\textbf{else} \; subrouteState.u.f \rightarrow S_2(u, p, pos, reqPos) \\
\qquad \Box \\
\qquad pointPosition.p?newPos \rightarrow S_2(u, p, newPos, reqPos)
\end{array}
$$

We define S_2 in terms of the functions *norm* and *rev* as follows.

$$
\begin{array}{l}
\forall\, p : \mathcal{P};\; u : \mathcal{U} \mid u \in norm(p) \cup rev(p) \bullet S_2(u, p) = S_2(u, p, cn, cn) \Leftrightarrow u \in norm(p) \\
\qquad \wedge\; S_2(u, p) = S_2(u, p, cn, cr) \Leftrightarrow u \in rev(p)
\end{array}
$$

Here, if $u \in norm(p)$, then we require that p is controlled normal whenever u is locked. Alternatively, if $u \in rev(p)$, then we require that p is controlled reverse whenever u is locked.

We are now in a position to state what it means for safety invariant 2 to hold of an interlocking.

Definition 7.2 Safety invariant 2 holds of an interlocking $\mathcal{I} = (\mathcal{T}, \mathcal{P}, S, \mathcal{R}, \mathcal{U})$ if safety invariant 1 holds of $\mathcal{I}$, and

$$\forall\, p : \mathcal{P};\; u : \mathcal{U} \mid u \in norm(p) \cup rev(p) \bullet \mathcal{I}_{CSP} \textbf{ imp } S_2(u, p)$$

□

Our justification for this definition is based upon the satisfaction of invariant 1. If we can establish that, given any track circuit $t \in \mathcal{T}$, no more than one sub-route $u \in sr(t)$ can be locked for a route at any time, then it follows that only one such sub-route can be locked for a route over a given set of points at any time.

Our next step is to establish how to reduce $\mathcal{I}_{CSP}$ sufficiently so that it can be analysed with respect to this invariant. All communications relevant to this safety invariant for some set of points $p \in \mathcal{P}$ occur in the points free to move data for p and the route setting data for those routes which can lock sub-routes over those points. Assuming satisfaction of safety

invariant 1, we can conclude that only one route which runs over p can be set at any time. As such, we may consider each case in isolation. The subprocess of $\mathcal{I}_{\text{CSP}}$ with which we are concerned for a route r is given by Algorithm 7.3.

Algorithm 7.3

1. Include the process representing points free to move data for p.
2. Include the process representing sub-route release data for u.
3. Include the process representing route setting data for r.
4. Include $\mathit{Point}(p)$.
5. Include $\mathit{Train}(r, \mathit{circuit}^*(\mathit{subroutes}(r)), \mathit{locks}(r))$.
6. Compose the processes mentioned in steps 1-5, and denote this composition $\beta_1(u, p, r)$.

□

As is the case for invariant 1, composition with further processes will only serve to reduce the set of possible behaviours, while increasing complexity. We introduce the following proposition accordingly.

Proposition 7.5 Given an interlocking $\mathcal{I} = (\mathcal{T}, \mathcal{P}, S, \mathcal{R}, \mathcal{U})$,

$$\forall\, p : \mathcal{P};\ u : \mathcal{U} \mid u \in \mathit{norm}(p) \cup \mathit{rev}(p) \bullet$$
$$(\forall\, r : \mathcal{R} \mid u \in \mathbf{elems}(\mathit{subroutes}(r)) \bullet \beta_1(u, p, r)\ \mathbf{imp}\ S_2(u, p)) \Rightarrow \mathcal{I}_{\text{CSP}}\ \mathbf{imp}\ S_2(u, p)$$

Proof. By virtue of the fact that all communications relevant to the analysis of safety invariant 2 with respect to u, p and r are observable in $\beta_1(u, p, r)$. □

As an example, we take points P201 of the Open ALVEY circuit, such that we have the following.

$$\mathit{norm}(\text{P201}) = \{\text{UAB-BC}, \text{UAB-CB}\}$$
$$\mathit{rev}(\text{P201}) = \{\text{UAB-AC}, \text{UAB-CA}\}$$

The set of all routes which can lock sub-routes over this set of points is given by

$$R = \{\text{R10A}, \text{R10B}, \text{R13}, \text{R15}\}$$

Referring to Appendix B, UAB-BC $\in$ **elems**(*subroutes*(R15)). To verify safety invariant 2 with respect to points P201, sub-route UAB-BC and route R15, we check that S_2(UAB-BC, P201) is implemented by β_1(UAB-BC, P201, R15).

By Algorithm 7.3, β_1(UAB-BC, P201, R15) is defined by

$$\beta_1(\text{UAB-BC}, \text{P201}, \text{R15}) \mathrel{\hat{=}}$$
$$\text{UAB-BC} \parallel \text{P201} \parallel \text{R15} \parallel \mathit{Train}(\text{R15}, \langle \text{TAB}, \text{TAA}, \text{TAK} \rangle, \{\text{P201}\}) \parallel \mathit{Point}(\text{P201})$$

This does indeed hold.

The extractions representing the three other cases for points P201 are defined similarly.

The complexity of verifying safety invariant 2 for points P201 and P202 is summarised in Tables D.5 and D.6 respectively. Note that the complexity of verifying P204 is the same as that of verifying P201 as the two points are symmetrical. This is also true of P202 and P203.

To conclude, we restate safety invariant 2 in the following way.

Proposition 7.6 Safety invariant 2 holds of an interlocking $\mathcal{I} = (\mathcal{T}, \mathcal{P}, S, \mathcal{R}, \mathcal{U})$ if and only if safety invariant 1 holds of $\mathcal{I}$ and

$$\forall\, p : \mathcal{P};\ u : \mathcal{U};\ r : \mathcal{R} \mid u \in norm(p) \cup rev(p) \wedge u \in \mathbf{elems}(subroutes(r)) \bullet \beta_1(u, p, r)\ \mathbf{imp}\ S_2(u, p)$$

Proof. By Definition 7.2 and Proposition 7.5. □

7.9.3 Invariant 3

Recall from Section 7.5 that safety invariant 3 is stated thus:

> If a route is set, then all of its sub-routes are locked.

Again, our first step is to define a CSP representation of this requirement. Given some route $r \in \mathcal{R}$, and the set of sub-routes

$$U = \{u : \mathcal{U} \mid u \in \mathbf{elems}(subroutes(r))\}$$

we define the process S_3 in the following way.

$$\begin{array}{ll} & S_3(r, U, locked) \mathrel{\widehat{=}} \\ \mathbf{if} & locked = U \\ \mathbf{then} & \Box_{u \in locked}\ subrouteState.u.f \rightarrow S_3(r, U, locked - \{u\}) \\ & \Box \\ & routeState.r.s \rightarrow routeState.r.xs \rightarrow S_3(r, U, locked) \\ \mathbf{else} & \Box_{u \in U - locked}\ subrouteState.u.l \rightarrow S_3(r, U, locked \cup \{u\}) \\ & \Box \\ & \Box_{u \in locked}\ subrouteState.u.f \rightarrow S_3(r, U, locked - \{u\}) \end{array}$$

Here, only when all sub-routes $u \in \mathbf{elems}(subroutes(r))$ are locked, can r be set. In addition, r must become unset before any of these sub-routes can be released.

Recalling that all sub-routes are initially locked, we define, for any route $r \in \mathcal{R}$,

$$S_3(r) \mathrel{\widehat{=}} S_3(r, \mathbf{elems}(subroutes(r)), \mathbf{elems}(subroutes(r)))$$

and state safety invariant 3 in the following way.

Definition 7.3 Safety invariant 3 holds of an interlocking $\mathcal{I} = (\mathcal{T}, \mathcal{P}, S, \mathcal{R}, \mathcal{U})$ if and only if

$$\forall\, r : \mathcal{R} \bullet \mathcal{I}_{\mathrm{CSP}}\ \mathbf{imp}\ S_3(r)$$

□

Given any $r \in \mathcal{R}$ for some interlocking $\mathcal{I}$, the method for determining the extraction required to verify that safety invariant 3 holds of r, which we denote $imp_{\mathcal{I}}(r)$, is defined as follows.

Algorithm 7.4

1. Include the process representing route setting data for r.
2. Include the processes representing sub-route release data for each element contained in the set $\mathbf{elems}(subroutes(r))$.
3. Include $Train(r, circuit^*(subroutes(r)), locks(r))$.
4. Denote the composition of the above processes by $imp_{\mathcal{I}}(r)$.

□

Proposition 7.7 Given an interlocking $\mathcal{I} = (\mathcal{T}, \mathcal{P}, S, \mathcal{R}, \mathcal{U})$,

$$\forall r : \mathcal{R} \bullet imp_{\mathcal{I}}(r) \ \mathbf{imp}\ S_3(r) \Rightarrow \mathcal{I}_{\mathrm{CSP}}\ \mathbf{imp}\ S_3(r)$$

Proof. By virtue of the fact that all behaviours relevant to the verification of safety invariant 3 for r are observable in $imp_{\mathcal{I}}(r)$. □

As an example, we take route R10A, where

$$\mathbf{elems}(subroutes(\mathrm{R10A})) = \{\mathrm{UAB\text{-}CA}, \mathrm{UBA\text{-}BA}\}$$

To prove safety invariant 3 for route R10A, we need to verify that

$$\mathcal{I}_{\mathrm{CSP}}\ \mathbf{imp}\ S_3(\mathrm{R10A}, \{\mathrm{UAB\text{-}CA}, \mathrm{UBA\text{-}BA}\}, \{\mathrm{UAB\text{-}CA}, \mathrm{UBA\text{-}BA}\})$$

Algorithm 7.4 gives us the following process.

$$imp_{\mathcal{I}}(\mathrm{R10A}) \mathrel{\widehat{=}} \begin{array}{c} \mathrm{R10A} \parallel \mathrm{UAB\text{-}CA} \parallel \mathrm{UBA\text{-}BA} \\ \parallel \\ Train(\mathrm{R10A}, \langle \mathrm{TAB}, \mathrm{TBA} \rangle, \{\mathrm{P201}\}) \end{array}$$

Again, this invariant can be verified quickly and efficiently with FDR. The complexity of verifying safety invariant 3 for the Open and Closed ALVEY interlockings is summarised in Table D.7 and Table D.8 respectively.

7.9.4 Invariant 4

Safety invariant 4 was stated thus:

> If a track circuit containing points is occupied, then the points are locked.

As with the previous cases, our first step towards the mechanical verification of this invariant is to define a CSP process which represents this requirement.

Given a set of points $p \in \mathcal{P}$, together with a track circuit $t \in \mathcal{T}$ such that $t \in berth(p)$, we define process S_4 in the following way.

$$
\begin{aligned}
S_4(p,t) \mathrel{\hat{=}}\ & pointLocked.p.false \rightarrow S_4(p,t) \\
& \Box \\
& pointLocked.p.true \rightarrow S_4'(p,t)
\end{aligned}
$$

$$
\begin{aligned}
S_4'(p,t) \mathrel{\hat{=}}\ & pointLocked.p.false \rightarrow S_4(p,t) \\
& \Box \\
& pointLocked.p.true \rightarrow S_4'(p,t) \\
& \Box \\
& circuitState.t.o \rightarrow circuitState.t.c \rightarrow S_4'(p,t)
\end{aligned}
$$

Here, track circuit t can only become occupied if points p are locked.

Given the above, we state safety invariant 4 in the following way.

Definition 7.4 Safety invariant 4 holds of an interlocking $\mathcal{I} = (\mathcal{T}, \mathcal{P}, S, \mathcal{R}, \mathcal{U})$ if and only if

$$\forall\, p : \mathcal{P};\ t : \mathcal{T} \mid t \in berth(p) \bullet \mathcal{I}_{CSP}\ \mathbf{imp}\ S_4(p,t)$$

□

Assuming that invariant 1 holds of $\mathcal{I}$, we can conclude that only at most one from the set of routes which pass over a given track circuit can be set at any one time. As such, we can verify points p against each such route individually. It follows that if this condition holds with respect to every route over p, then it holds of p.

The following algorithm defines our extraction process for this invariant.

Algorithm 7.5

1. Assume a set of points $p \in \mathcal{P}$ and a route $r \in \mathcal{R}$, such that $p \in locks(r)$.
2. Denote the composition of $Train(r, circuit^*(subroutes(r)), locks(r))$, the process representing points free to move data for p, and the process representing route setting data for r, by $\psi_{\mathcal{I}}(p,r)$.

□

Proposition 7.8 Given an interlocking $\mathcal{I} = (\mathcal{T}, \mathcal{P}, S, \mathcal{R}, \mathcal{U})$, if safety invariant 1 holds of $\mathcal{I}$, then

$$
\begin{aligned}
&\forall\, p : \mathcal{P};\ t : \mathcal{T};\ r : \mathcal{R} \mid \\
&p \in locks(r) \land t \in berth(p) \bullet \psi_{\mathcal{I}}(p,r)\ \mathbf{imp}\ S_4(p,t) \Rightarrow \mathcal{I}_{CSP}\ \mathbf{imp}\ S_4(p,t)
\end{aligned}
$$

Proof. Assuming that safety invariant 1 holds of $\mathcal{I}$, only one route can lock sub-routes over t at any time. Also, all behaviours relevant to the analysis of safety invariant 4 with respect to p and r are observed by $\psi_{\mathcal{I}}(p,r)$. As such, we can conclude that $\mathcal{I}_{CSP}\ \mathbf{imp}\ S_4(p,t)$. □

Take as an example points P202 for the Closed ALVEY. The set of routes which lock sub-routes over this set of points is given by

$$R = \{\text{R11A}, \text{R11B}, \text{R12}, \text{R14}\}$$

In the case of R11A, we have

$$\psi_1(\text{P202}, \text{R11A}) \mathrel{\hat{=}} \text{R11A} \parallel \text{P202} \parallel \mathit{Train}(\text{R11A}, \langle \text{TAD}, \text{TAC} \rangle, \{\text{P202}\})$$

The complexity of analysing this invariant for routes over points P202 for the Closed ALVEY is summarised in Table D.9.

7.9.5 Invariant 5

Recall our statement of safety invariant 5:

> If a sub-route is locked for a route, then all sub-routes ahead of it on that route are also locked.

We consider pairs of sub-routes for a given route, and develop a CSP process, S_5, accordingly. This process involves five parameters, representing the route, the two sub-routes, and the states of the two sub-routes.

Assuming that u_2 is in advance of u_1, if the latter is locked, then the former can also become locked only after a request for some route r to be set. In addition, u_1 may be released at any time. On the other hand, if u_1 is free, then we are willing to observe any change in state of our sub-routes, as well as a request for r to be set.

$$\begin{array}{l}
S_5(r, u_1, state_1, u_2, state_2) \mathrel{\hat{=}} \\
\quad \textbf{if} \quad state_1 = l \\
\quad \textbf{then} \; routeState.r.req \rightarrow \mathit{LockSubroutes}(r, u_1, state_1, u_2, state_2) \\
\qquad \Box \\
\qquad subrouteState.u_1.f \rightarrow S_5(r, u_1, f, u_2, state_2) \\
\quad \textbf{else} \; routeState.r.req \rightarrow \mathit{LockSubroutes}(r, u_1, state_1, u_2, state_2) \\
\qquad \Box \\
\qquad subrouteState.u_1?newState \rightarrow S_5(r, u_1, newState, u_2, state_2) \\
\qquad \Box \\
\qquad subrouteState.u_2?newState \rightarrow S_5(r, u_1, state_1, u_2, newState)
\end{array}$$

If a request for r is forthcoming, then this process behaves as *LockSubroutes*, until both sub-routes are locked, at which point it again behaves as S_5.

$$\begin{array}{l}
\mathit{LockSubroutes}(r, u_1, state_1, u_2, state_2) \mathrel{\hat{=}} \\
\quad \textbf{if} \quad state_1 = l \wedge state_2 = l \\
\quad \textbf{then} \; S_5(r, u_1, l, u_2, l) \\
\quad \textbf{else} \; subrouteState.u_1.l \rightarrow \mathit{LockSubroutes}(r, u_1, l, u_2, state_2) \\
\qquad \Box \\
\qquad subrouteState.u_2.l \rightarrow \mathit{LockSubroutes}(r, u_1, state_1, u_2, l)
\end{array}$$

We can now define satisfaction of safety invariant 5.

Definition 7.5 Safety invariant 5 holds of an interlocking $\mathcal{I} = (\mathcal{T}, \mathcal{P}, S, \mathcal{R}, \mathcal{U})$ if and only if

$$\begin{array}{l}\forall\, r : \mathcal{R};\ u_1, u_2 : \mathcal{U} \mid \\ u_1 \in \mathbf{elems}(\mathit{subroutes}(r)) \wedge u_2 \in \mathbf{elems}(\mathit{subroutes}(r)) \wedge \\ \mathit{subroutes}(r)^{\sim}(u_1) < \mathit{subroutes}(r)^{\sim}(u_2) \bullet \mathcal{I}_{\mathrm{CSP}}\ \mathbf{imp}\ S_5(r, u_1, l, u_2, l)\end{array}$$

□

That is, given any route $r \in \mathcal{R}$ and any sub-routes $u_1, u_2 \in \mathcal{U}$, such that u_1 and u_2 are both sub-routes of r and u_1 precedes u_2, then $\mathcal{I}_{\mathrm{CSP}}$ should implement the above requirement for these routes and sub-routes.

As with our previous examples, we define an extraction, $\kappa_{\mathcal{I}}(r, u_1, u_2)$, which considers only those communications relevant to the safety invariant.

Algorithm 7.6

1. Assume some route $r \in \mathcal{R}$ and two sub-routes

$$u_1, u_2 : \mathcal{U} \mid u_1 \in \mathbf{elems}(\mathit{subroutes}(r)) \wedge u_2 \in \mathbf{elems}(\mathit{subroutes}(r)) \wedge u_1 \neq u_2$$

2. Extract the process representing route setting data for r.
3. Extract the processes representing sub-route release data for u_1 and u_2.
4. Include $\mathit{ALTsubroute}(u_1)$ and $\mathit{ALTsubroute}(u_2)$.
5. Include $\mathit{Train}(r, \mathit{circuit}^*(\mathit{subroutes}(r)), \mathit{locks}(r))$.
6. Compose each of the above processes, and denote this composition $\kappa_{\mathcal{I}}(r, u, u')$.

□

As with previous invariants, we reduce our analysis of this invariant for an interlocking to the analysis of extractions.

Proposition 7.9 Given an interlocking $\mathcal{I} = (\mathcal{T}, \mathcal{P}, S, \mathcal{R}, \mathcal{U})$,

$$\begin{array}{l}\forall\, r : \mathcal{R};\ u_1, u_2 : \mathcal{U} \mid u_1 \in \mathbf{elems}(\mathit{subroutes}(r)) \wedge \\ u_2 \in \mathbf{elems}(\mathit{subroutes}(r)) \wedge \mathit{subroutes}(r)^{\sim}(u_1) < \mathit{subroutes}(r)^{\sim}(u_2) \bullet \\ \kappa_{\mathcal{I}}(r, u_1, u_2)\ \mathbf{imp}\ S_5(r, u_1, l, u_2, l) \Rightarrow \mathcal{I}_{\mathrm{CSP}}\ \mathbf{imp}\ S_5(r, u_1, l, u_2, l)\end{array}$$

Proof. By virtue of the fact that all communications relevant to the analysis of safety invariant 5 for r, u_1 and u_2 are observable in $\kappa_{\mathcal{I}}(r, u_1, u_2)$. □

Proposition 7.10 Given an interlocking $\mathcal{I} = (\mathcal{T}, \mathcal{P}, S, \mathcal{U})$, if

$$\begin{array}{l}\forall\, r : \mathcal{R};\ u_1, u_2 : \mathcal{U} \mid \\ u_1 \in \mathbf{elems}(\mathit{subroutes}(r)) \wedge u_2 \in \mathbf{elems}(\mathit{subroutes}(r)) \wedge \\ \mathit{subroutes}(r)^{\sim}(u_1) < \mathit{subroutes}(r)^{\sim}(u_2) \bullet \\ \kappa_{\mathcal{I}}(r, u_1, u_2)\ \mathbf{imp}\ S_5(r, u_1, l, u_2, l)\end{array}$$

then safety invariant 5 holds of $\mathcal{I}$.

Proof. By Definition 7.5 and Proposition 7.9. □

As an example, we take route R15, where

$$\mathit{subroutes}(\text{R15}) = \langle \text{UAB-BC}, \text{UAA-AB}, \text{UAK-AB} \rangle$$

By Proposition 7.9 and given that this relation is transitive, we therefore need to prove that

$$\begin{aligned} & \kappa_{\mathcal{I}}(\text{R15}, \text{UAB-BC}, \text{UAA-AB}) \ \mathbf{imp}\ S_5(\text{R15}, \text{UAB-BC}, l, \text{UAA-AB}, l) \\ \wedge\ & \kappa_{\mathcal{I}}(\text{R15}, \text{UAB-BC}, \text{UAK-AB}) \ \mathbf{imp}\ S_5(\text{R15}, \text{UAB-BC}, l, \text{UAK-AB}, l) \\ \wedge\ & \kappa_{\mathcal{I}}(\text{R15}, \text{UAA-AB}, \text{UAK-AB}) \ \mathbf{imp}\ S_5(\text{R15}, \text{UAA-AB}, l, \text{UAK-AB}, l) \end{aligned}$$

Taking the first case as an example, Algorithm 7.6 gives us

$$\begin{aligned} \kappa_{\mathcal{I}}(\text{R15}, \text{UAB-BC}, \text{UAA-AB}) \mathrel{\widehat{=}} & (\mathit{ALTsubroute}(\text{UAB-BC}) \ |||\ \mathit{ALTsubroute}(\text{UAA-AB})) \\ & \| \\ & \text{R15} \parallel \text{UAB-BC} \parallel \text{UAA-AB} \\ & \| \\ & \mathit{Train}(\text{R15}, \langle \text{TAB}, \text{TAA}, \text{TAK} \rangle, \text{P201}) \end{aligned}$$

The complexity of analysing this invariant for the Open ALVEY is detailed in Table D.10.

7.10 Error Injection

We have illustrated how to verify, via refinement checking, safety invariants of SSI geographic data. The geographic data which we have analysed has been tested many times, and our analysis simply confirms, albeit in an extremely efficient manner, that it is correct with respect to the invariants which we have chosen to analyse. Having illustrated that the verification of such data is feasible with FDR, we now need to establish that, when faced with geographic data which does not satisfy its required safety invariants, that our approach is capable of illuminating such errors.

Errors may be introduced in any number of ways: they may be present in the original SSI database, they may be introduced in the translation from GDL to CSP, or they may occur as a result of an incorrect implementation of one of our extraction algorithms.

In Chapter 8, we discuss plans for further automating the process of verifying SSI data. Such plans include the automation of the translation and extraction generation steps. As such, we shall not consider the possibility of introducing errors at these stages here. Rather, we concentrate upon the ability of detecting errors already present in geographic databases. Therefore, in this section, we present some examples of error injection into the geographic data for the Open ALVEY, and explain how our approach might help to illuminate such errors.

Invariant 1

Recall our verification of safety invariant 1 for track circuit TAK. Recall further, that four routes can lock sub-routes over this track circuit: R13, R15, R22 and R24. Suppose that the

route setting data for route R22 has been written incorrectly as

Q22 **if** P204 cfr UAJ-CB f UAK-AB f
then R22 s P204 cr UAJ-BC l UAK-AB l UAA-BA l

(This route locks UAK-BA *not* UAK-AB.) As routes R13 and R15 both lock UAK-AB and given the incorrect condition given above, by Proposition 7.4, if

$$\begin{aligned} &\delta_1(\text{TAK}, \text{R13}, \text{R24}) \ \textbf{imp}\ S_1(\{\text{R13}, \text{R24}\}) \\ \wedge\ &\delta_1(\text{TAK}, \text{R15}, \text{R24}) \ \textbf{imp}\ S_1(\{\text{R15}, \text{R24}\}) \\ \wedge\ &\delta_1(\text{TAK}, \text{R22}, \text{R24}) \ \textbf{imp}\ S_1(\{\text{R22}, \text{R24}\}) \end{aligned}$$

then safety invariant 1 holds of track circuit TAK.

Unsurprisingly, cases 1 and 2 are successful — they involve correct data. However, when FDR is asked to verify case 3, the relevant check fails. Recalling our method of extraction for safety invariant 1, we compose the incorrect version of R22 with our CSP interpretation of the route setting data for route R24, which is given by

Q24 **if** P204 cfn UAJ-CA f UAK-AB f
then R24 s P204 cn UAJ-AC l UAK-BA l UAA-BA l

Assume that route R24 has been requested — this results in sub-route UAK-BA being locked. Because both processes can observe the communication *subrouteState*.UAK-AB.*l*, and such an observation is independent of any other processes in $\delta_1(\text{TAK}, \text{R22}, \text{R24})$ (after all, this extraction involves no route setting data to determine when this sub-route becomes locked), the invariant is invalidated for this pair of sub-routes.

Invariant 2

Recalling our verification of safety invariant 2 for points P201, suppose that the points free to move data for this set of points had been written incorrectly as

P201N TAB c UAD-AC f UAB-CA f
P201R TAB c UAB-BC f UAB-CB f

That is, UAB-AC has been replaced by UAD-AC.

Recalling our approach to analysing this invariant, we take the pair (UAB-AC, UAB-BC) as an example. Unsurprisingly, FDR detects that this pair fails to satisfy the invariant.

Invariant 3

Recall our verification of safety invariant 3 for route R10A. Recall also that

$$\textit{subroutes}(\text{R10A}) = \langle \text{UAB-CA}, \text{UBA-BA} \rangle$$

and that the sub-route release data for UAB-CA is given by

UAB-CA f **if** TAB c R10A xs

If this conditional check was rewritten as

UAB-CA f **if** TAB c R10B xs

then *imp*(R10A) would be defined by

$$
\begin{array}{c}
(\text{R10A} \\
[\![\ \{\mathit{subrouteState}.\text{UAB-CA}.\mathit{l}, \mathit{subrouteState}.\text{UBA-BA}.\mathit{l}\}\ ]\!] \\
(\text{UAB-CA}\ [\![\ \{\mathit{subrouteState}.\text{UAB-CA}\}\ ]\!]\ \text{UBA-BA})) \\
[\![\ \{\mathit{routeState}.\text{R10A}.\mathit{req}, \mathit{routeState}.\text{R10A}.\mathit{s}, \mathit{circuitState}.\text{TAB}, \mathit{circuitState}.\text{TBA}\}\ ]\!] \\
\mathit{Train}(\text{R10A}, \langle\text{TAB,TBA}\rangle, \{\text{P201}\})
\end{array}
$$

When verifying the above against S_3(R10A), FDR detects that, after route R10A has been set, therefore locking sub-route UAB-CA, this sub-route is released immediately if track circuit TAB is clear and route R10B is unset.

Invariant 4

Recall our verification of safety invariant 4 for points P202 and route R11A. Assume that the route setting data for this route was incorrectly written in such a way that route R11A locked points P201 instead of points P202, i.e., in the form

Q11A **if** P202 cfn UAD-BA f UAC-BA f
then P201 cn UAD-AB l UAC-AB l

FDR confirms that, after this route is set, track circuit TAD will become occupied without points P202 being locked.

Given the above examples, one can conclude that, when faced with conditional checks which foul our safety invariants, FDR does detect such errors. FDR also offers the opportunity to analyse each conditional check *in isolation*, thereby allowing one to detect exactly which component is behaving incorrectly.

7.11 Scaling Up

As discussed in Section 7.1, formal techniques are often seen to be applied to what some dismiss as 'toy' examples, such as the Open and Closed versions of the ALVEY junction, but rarely is evidence given of their 'scalability'.

We have outlined an approach for verifying SSI geographic data with respect to a number of safety invariants, but so far have shown only how it is applicable to small interlockings. To test whether this approach is applicable to real interlockings, we have attempted to verify safety invariants of (simplified) geographic data associated with such an interlocking. In this section, we provide a brief overview of that work. Again, the data described in this section was kindly supplied by British Rail Research (Derby).

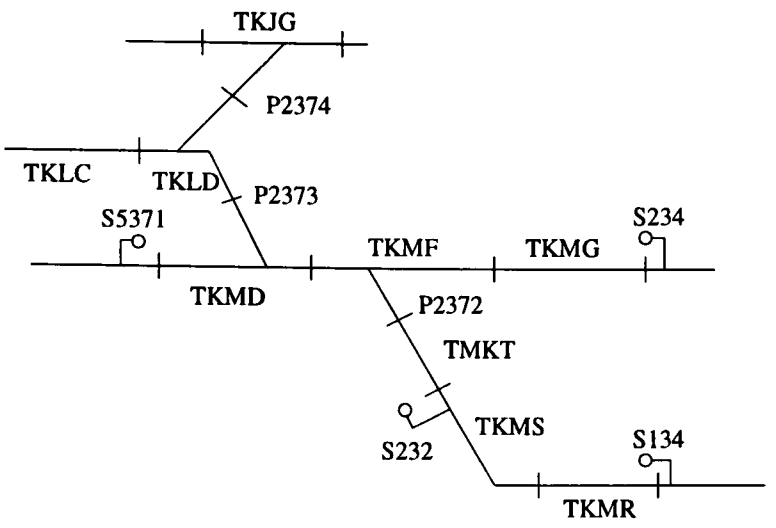

Figure 7.5: The context of track circuit TKMF.

7.11.1 Invariant 1

We concern ourselves with track circuits: TKMF, the context of which is given by Figure 7.5. This route has four associated sub-routes, given by sr(TKMF).

$$sr(\text{TKMF}) = \{\text{UKMF-AC}, \text{UKMF-BC}, \text{UKMF-CA}, \text{UKMF-CB}\}$$

Referring to Table 7.1, we find that 12 routes can lock sub-routes over this track circuit, and that we need to test this invariant against 51 pairs of routes — a far greater number than for the Open and Closed ALVEY interlockings.

The complexity of the route setting data for these routes is, on the whole, of a greater complexity than that which we have seen previously. Arguably, this second problem is less of a drawback than the first. Via abstraction, the sizes of checks to be performed can be reduced to a manageable complexity. However, we cannot escape the time consuming nature of the need to extract, construct, and verify relevant processes for 57 pairs of sub-routes. Thus, there is a definite trade-off between reducing the complexity of checks to a sufficient level, and the time taken to construct so many checks. This would make an interesting area for further study.

As an example, we take the pair of sub-routes UKMF-CA and UKMF-CB. The sets of routes which can lock these sub-routes are given by the sets R_1 and R_2.

$$R_1 = \{\text{R231C}, \text{R233B}, \text{R235B}, \text{R5371A}\}$$
$$R_2 = \{\text{R231D}, \text{R233C}, \text{R235C}, \text{R5371B}\}$$

We take the pair (R231C, R231D) as an example. The route setting data for these routes is as follows.

Q231C **if** UKCZ-AB f UKMH-AB f
P2380 cfr P2375 cfr P2374 cfn P2373 cfr P2372 cfr
then R231C s P2380 cr P2375 cr P2380 cr P2374 cr P2373 cr P2372cr
UKCZ-BA l UKHA-CB l UKJA-CA l UKJB-BA l UKJC-CB l
UKLC-CA l UKLD-CB l UKMD-CA l UKMF-CA l UKMG-BA l UKMH-BA l

Subroute	Set by
UKMF-AC	R234A R234B R234C
UKMF-BC	R232A
UKMF-CA	R231C R233B R235B R5371A
UKMF-CB	R231D R233C R235C R5371B

Table 7.1: Route setting over TKMF.

Q231D **if** UKCZ-AB f UKMS-AB f
P2380 cfn P2375 cfr P2374 cfn P2373 cfr P2372 cfn
then R231D s P2373 cr P2372 cn P2380 cr P2375 cr P2374 cn
UKCZ-BA l UKHA-CB l UKJA-CA l UKJB-BA l UKJC-CB l
UKLC-CA l UKLD-CB l UKMD-CA l UKMF-CB l UKMT-BA l UKMS-BA l

The route setting data for both of these routes requires that points P2372 is controlled free to go reverse — the points free to move data for this set of points is given by

P2372N TKMF c UKMF-AC f UKMF-CA f
P2372R TKMF c UKMF-BC f UKMF-CB f

It follows that

$$\delta_1'(\text{TKMF}, \text{R231C}, \text{R231D}) \mathrel{\widehat{=}} \text{P2372} \parallel (RC(\{\text{R231C}, \text{R231D}\}) \parallel (\text{R231C} \;[\![\{ \begin{array}{c} \mathit{subrouteState}.\text{UKCZ-AB}.*, \mathit{pointState}.\text{P2375}.\mathit{cfr}.*, \\ \mathit{pointState}.\text{P2374}.\mathit{cfn}.*, \mathit{pointState}.\text{P2373}.\mathit{cfr}.* \end{array} \}]\!] \text{R231D}))$$

By careful use of abstraction, FDR is able to verify safety invariant 1 for track circuit TKMF, with the average length of checks being approximately 1 hour on a standard workstation.

7.11.2 Invariant 2

As an example of the verification of invariant 2, we take points P2380, the points free to move data for which is given below.

P2380N TKJA c TKHA c UKHA-BC f UKJA-BC f
P2380R TKHA c TKJA c UKHA-AC f UKHA-CA f UKJA-AB f UKJA-BA f

Recall that safety invariant holds of points P2380 if and only if

$$\forall\, u : \{\text{UKHA-AC}, \text{UKHA-BC}, \text{UKHA-CA}, \text{UKJA-AB}, \text{UKJA-BA}, \text{UKJA-BC}\};\ r : \mathcal{R} \mid$$
$$u \in \mathit{subroutes}(r) \bullet \beta_1(u, \text{P2380}, r)\ \mathbf{imp}\ S_2(u, \text{P2380})$$

If we take u = UKHA-AC as an example, then we only need to consider route R238. In this case,

$$\beta_1(\text{UKHA-AC}, \text{P2380}, \text{R238}) \mathrel{\widehat{=}}$$
$$\text{UKHA-AC} \parallel \text{P2380} \parallel \text{R238} \parallel \mathit{Point}(\text{P2380})$$
$$\parallel$$
$$\mathit{Train}(\text{R238}, \langle \text{TKHG}, \text{TKHB}, \text{TKHA}, \text{TKCZ}, \text{TKCY}, \text{TKCX} \rangle, \{\text{P2380}\})$$

Such checks can be carried out extremely efficiently, as in each case we deal with compositions of only three processes.

7.11.3 Invariant 3

We choose route R231D as a means of illustrating this approach on a larger interlocking. Recalling the route setting data for this route, we find that 11 sub-routes are locked when this route is set, and define the process *imp*(R231D) accordingly.

$$\mathit{imp}(\text{R231D}) \mathrel{\widehat{=}} \text{R231D}$$
$$\parallel$$
$$\parallel_{u \in \mathbf{elems}(\mathit{subroutes}(\text{R231D}))} u$$
$$\parallel$$
$$\mathit{Train}(\text{R231D}, \mathit{circuit}^*(\mathit{subroutes}(\text{R231D})), \mathit{locks}(\text{R231D}))$$

Again, by applying appropriate abstractions, verification of invariant 3 for this route reduces to a problem of manageable complexity.

7.11.4 Invariant 4

We take the verification of invariant 4 for one the more complex points associated with our larger interlocking, the points free to move data for which, is given by

P2380N TKJA c TKHA c UKHA-BC f UKJA-CA f
P2380R TKHA c TKJA c UKHA-AC f UKHA-CA f UKJA-AB f UKJA-BA f

By Proposition 7.8, safety invariant 4 holds of this point if and only if

$$\forall\, r : \mathcal{R} \mid \text{P2380} \in \mathit{locks}(r) \bullet \psi_1(\text{P2380}, r)\ \mathbf{imp}\ S_4(\text{P2380}, \text{TKHA})$$
$$\wedge\ \psi_1(\text{P2380}, r)\ \mathbf{imp}\ S_4(\text{P2380}, \text{TKHA})$$

The set of all routes which lock sub-routes over this set of points is given by

$$R = \{\text{R231A}, \text{R231B}, \text{R231C}, \text{R231D}, \text{R233A}, \text{R233B},$$
$$\text{R233C}, \text{R234B}, \text{R234C}, \text{R236A}, \text{R236B}, \text{R238}\}$$

Therefore, 12 checks have to be performed to verify this invariant for point P2380. An example of a process to be checked is given by

$$\psi(\text{P2380}, \text{R231D}) \mathrel{\widehat{=}} \begin{array}{c} \text{R231D} \parallel \text{P2380} \\ \parallel \\ \mathit{Train}(\text{R231D}, \mathit{circuit}^*(\mathit{subroutes}(\text{R231D})), \mathit{locks}(\text{R231D})) \end{array}$$

As was the case with our ALVEY interlockings, assuming the satisfaction of invariant 1, verifying this invariant is relatively straightforward.

7.11.5 Invariant 5

As a final example, we illustrate the verification of invariant 5 for route R231D. To ensure that this invariant holds of this set of points, we need to prove that

$$\begin{array}{l} \forall\, u_1, u_2 : \mathcal{U} \mid \\ u_1 \in \mathbf{elems}(\mathit{subroutes}(\text{R231D})) \wedge u_2 \in \mathbf{elems}(\mathit{subroutes}(\text{R231D})) \wedge \\ \mathit{subroutes}(\text{R231D})^{\sim}(u_1) < \mathit{subroutes}(\text{R231D})^{\sim}(u_2) \bullet \\ \kappa_7(\text{R231D}, u_1, u_2) \ \mathbf{imp}\ S_5(\text{R231D}, u_1, l, u_2, l) \end{array}$$

Referring to the above,

$$\begin{array}{l} \mathit{subroutes}(\text{R231D}) = \{\text{UKCZ-BA}, \text{UKHA-CB}, \text{UKJA-CA}, \text{UKJB-BA}, \text{UKJC-CB}, \\ \qquad\qquad \text{UKLC-CA}, \text{UKLD-CB}, \text{UKMD-CA}, \text{UKMF-CB}, \text{UKMT-BA}, \text{UKMS-BA}\} \end{array}$$

As such, we need to verify ten adjacent pairs of sub-routes. The process representing the check for the pair (UKJA-CA, UKJB-BA) is given by

$$\begin{array}{l} \kappa_7(\text{R231D}, \text{UKJA-CA}, \text{UKJB-BA}) \mathrel{\widehat{=}} \\ \qquad (\mathit{ALTsubroute}(\text{UKJA-CA}) \mid\mid\mid \mathit{ALTsubroute}(\text{UKJB-BA})) \\ \qquad\qquad \parallel \\ \qquad \text{R231D} \parallel \text{UKJA-CA} \parallel \text{UKJB-BA} \\ \qquad\qquad \parallel \\ \mathit{Train}(\text{R231D}, \langle \text{TKCZ}, \text{TKHA}, \text{TKJA}, \text{TKJB}, \text{TKJC}, \\ \qquad \text{TKLC}, \text{TKLD}, \text{TKMD}, \text{TKMF}, \text{TKMT}, \text{TKMS}\rangle, \{\text{P2372}, \text{P2373}, \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{P2374}, \text{P2375}, \text{P2380}\}) \end{array}$$

In this chapter, we have seen how FDR has been applied to a problem of industrial relevance. In the next, and final, chapter, we examine some ways of building upon these results, and suggest some ways of taking this research further.

Chapter 8

Conclusions

8.1 A Review

In this thesis, we have seen how CSP, together with mechanical support in the form of FDR, can be applied in a number of ways to the analysis and development of safety-critical systems. This goal has been achieved in three ways: the development of a general approach to the analysis of safety-critical systems with CSP; the development of an approach to the analysis of non-interference properties of safety-critical systems with CSP; and the demonstration that the mechanical verification of geographic data for Solid State Interlocking railway signalling systems is computationally feasible with FDR. These areas were seen to be closely related, in that it is possible to describe SSI safety invariants in terms of mutual exclusion, which is a specific form of non-interference. In this, the final chapter, we provide a brief review of the main points of this thesis, and present some suggestions for possible areas of future work.

8.1.1 A General Approach to Safety

In Chapter 2, we saw how it is possible to move from a hazard analysis, in the form of a finite state automaton, to a CSP representation of that analysis. Although this process of developing a CSP interpretation of a hazard analysis is somewhat informal, it allows us to construct a representation against which a specification can be verified with respect to its (known) hazards. By analysing the composition of a specification and a hazard analysis for divergence, FDR offers an efficient and convenient method of testing for the absence of hazards in CSP specifications. Because we test for hazards with respect to divergence, we can show that safety, in this sense, is preserved under refinement.

As well as testing for the absence or presence of hazards, our general approach to the analysis of safety-critical systems with CSP and FDR allows us to reason about the prevention of faults, and the recovery from failures in such systems. Properties such as these cannot be verified easily by means of testing for refinement or non-divergence: they are described in terms of non-interference.

8.1.2 Safety-Critical Non-Interference

In Chapters 2-4, we formulated a notion of safety-critical non-interference, termed protection, which was based on the property of lazy independence of [RWW94] and [Ros95]. We discussed a number of ways in which safety and security can be related, and argued that this formulation of non-interference, given in terms of information flow, has a role to play in the analysis of safety-critical systems.

There are several advantages to be gained from taking this approach to safety-critical non-interference. First, there has recently been substantial research into the analysis of non-interference in security-critical systems, using this and similar approaches, in CSP. This has resulted in fruitful collaboration between those working on safety and security problems, which, in turn, has led to attempts at developing approaches and properties which are applicable to both types of system. To the best of our knowledge, the research described in this thesis has been the first substantial attempt at reasoning about problems of non-interference in safety-critical systems.

Secondly, our approach to non-interference is based on the CSP concept of determinism. As determinism is central to the theory of the failures-divergences model, which is, in many respects, the 'standard' model of CSP, there is already a well-established body of knowledge related to this property. Thus, we can have confidence in claiming that the theory which underpins our approach to non-interference is well-established.

Finally, as a consequence of the failures-divergences model being regarded as the standard model of CSP, mechanical support is more readily available for this model (and, therefore, our approach to non-interference), than for others. In particular, FDR provides us with a suitable platform for analysing determinism of, and refinement between, relatively complex processes. In many ways, this is a major advantage of our approach — it enables us to test non-interference properties of larger systems than might otherwise be possible if such tool support were not available. It should be noted that much of the work described in this thesis would not have been possible without the aid of FDR.

Having established a notion of non-interference for safety-critical systems, we then developed a formal system for protection, based upon the algebraic laws of CSP and the semantic definitions of the failures-divergences model. Having such a proof system in place affords us the benefit of being able to formally develop CSP processes of which certain non-interference conditions should hold. It also enables us to reduce large proof obligations to a series of smaller proofs of a more manageable size. As we have seen, it is often the case that mechanical proofs have to be reduced in this fashion. In addition, we illustrated that protection is preserved under refinement.

We then identified areas in which our theory could be of benefit. In this respect, we postulated a number of properties and concepts of safety-critical systems which can be described in terms of protection. Such properties included fault-tolerance and dependability. Several of these concepts were illustrated by means of examples. In particular, we showed how properties of a simplified Automatic Train Protection system might be characterised in terms of non-interference. This case study helped to illustrate how the approach can be applied to larger safety-critical systems.

One of the benefits of this work is that it has enabled us to postulate a general approach to the treatment of faults, failures, and hazards for safety-critical systems in CSP. Possible extensions to the theory of safety-critical non-interference are discussed in Section 8.2.

8.1.3 Mutual Exclusion

In Chapter 6, we saw how CSP can be used to represent a number of mutual exclusion algorithms. In addition, we saw how programming constructs, as well as properties of distributed algorithms, can be represented by CSP processes. The purpose of Chapter 6 was neither to investigate mutual exclusion, nor to postulate a new solution to the problem. Rather, Chapter 6 fulfilled three important roles in the context of this thesis.

First, mutual exclusion is a specific form of non-interference. As such, it was necessary for us to investigate how such non-interference can be modelled in terms of CSP. Secondly, mutual exclusion is an elegant case study against which we were able to verify our approach to fault-tolerance. Finally, many of the problems and assumptions associated with our modelling of Solid State Interlocking geographic databases are concerned with mutual exclusion. As such, an introduction to the modelling of mutual exclusion and atomicity in FDR was essential before embarking upon the work of Chapter 7.

8.1.4 Solid State Interlocking

In Chapter 7, we demonstrated that the mechanical verification of SSI geographic databases is feasible with FDR. This is a fundamental problem in the safe application of SSI systems in this and other countries. Currently, such data is, to a large extent, verified manually. Given the complexity of a real interlocking, this is clearly not the most satisfactory state of affairs. As such, a fully (or even semi) automated approach in this area could be of great benefit in terms of time, money, and safety.

We have shown that, for simplified data, certain SSI safety invariants can be verified for real interlockings. Furthermore, we have reason to be confident that these methods, at least for the invariants described in Chapter 7, are applicable to full-scale data.

Thus, Chapter 7 had three main objectives:

1. To investigate the applicability of CSP and FDR to a complex 'real-life' problem which is of genuine interest to industry.
2. To investigate the feasibility of the mechanical verification of geographic data associated with Solid State Interlocking signalling systems.
3. To investigate whether non-interference has a role to play in solving the above problem.

The first goal has been achieved. CSP, together with FDR, has shown itself to be an ideal vehicle for the description of the behaviour of Solid State Interlocking systems.

The second, and possibly most important, goal has also been achieved, albeit for a simplified version of geographic data. As such, we have been able to postulate that the mechanical verification of SSI geographic data is feasible.

Finally, non-interference, in the form of mutual exclusion, certainly did help in solving the problem at hand: the analysis of each of the safety invariants considered relied upon a mutual exclusion property to reduce complexity.

Thus, each of our aims for Chapter 7 were, with varying degrees of success, achieved. However, there is far more work to be carried out in this area — this is discussed in the next section.

It was stated in Section 7.1 that a number of other approaches, such as CCS [Mor91] and HOL [Mor93] have been applied to the problem of verifying SSI geographic databases. This

begs the following question: If this problem can be solved with CSP, can it be solved with any of these other approaches?

Our approach to the modelling of geographic data involves a subset of the CSP operators — specifically, we rely upon parameterised processes, event prefixing, external choice, parallel composition, and partial interleaving. As such, one might conclude that conditional checks might also be modelled with CCS. Recalling that . and + are the CCS equivalents of → and □ respectively, we might write CCS representations of sub-route release data in terms of agents of the form

$$\begin{aligned}\text{UAE-BA}\mathit{locked}(c,f,l) \stackrel{\text{def}}{=}\ & \mathit{circuitState}_{\text{TAE}}(c)\ .\ \text{UAE-BA}\mathit{locked}(c,f,l)\\ &+\\ & \mathit{circuitState}_{\text{TAE}}(o)\ .\ \text{UAE-BA}\mathit{locked}(o,f,l)\\ &+\\ & \mathit{subrouteState}_{\text{UAD-BA}}(f)\ .\ \text{UAE-BA}\mathit{locked}(c,f,l)\\ &+\\ & \mathit{subrouteState}_{\text{UAD-BA}}(l)\ .\ \text{UAE-BA}\mathit{locked}(c,l,l)\\ &+\\ & \mathit{subrouteState}_{\text{UAD-CA}}(f)\ .\\ & \mathit{subrouteState}_{\text{UAE-BA}}(f)\ .\ \text{UAE-BA}\mathit{free}(c,f,f)\\ &+\\ & \mathit{subrouteState}_{\text{UAD-CA}}(l)\ .\ \text{UAE-BA}\mathit{locked}(c,f,l)\end{aligned}$$

There is one aspect of our modelling which does not sit so well with the CCS approach. We tend to use partial interleaving when composing route setting processes as we have no wish for such processes to synchronise upon occurrences of *subrouteState.u.l* for some sub-route $u \in \mathcal{U}$ if two different routes r and r' can lock u. As CCS does not possess such a form of composition, we look for a way to counter this problem. The simplest solution is to associate each observation of a sub-route being locked with the particular route which sets it. So, for example, if sub-route u is locked during the setting of route r, then we might observe *subrouteState.u.r.l*; similarly, *subrouteState.u.r'.l* represents the locking of u for route r'. If we were to adopt this change in notation, then all relevant processes could be composed via the parallel composition operator, ||, (or | for CCS).

If we were to assume that the problem of our approach involving multi-way synchronisation could be overcome with CCS, it follows that our modelling of geographic databases lends itself to that technique. However, the key question that we are concerned with is stated thus: Does our solution to the reduction of the complexity of such databases also work for CCS? The simple answer is no. Recall that it was failures-divergences refinement that allowed us to decompose the problem at hand. More specifically, refinement allows us to state that, given processes P, Q and R, if P is refined by Q, then we can conclude that $P \parallel R$ is refined by $Q \parallel R$. It is precisely this result which allows us to decompose interlockings with CSP. As CCS has no corresponding notion of refinement, we cannot approach the problem in the same way with that notation. The proof of properties with CCS often involves finding the simplest process of an equivalence class — this is far less convenient than refinement for enabling us to decompose systems. Thus, it is entirely due to the concept of failures-divergences refinement that we can claim that our approach to the problem, together with the proffered

solution, is unique to CSP.

To summarise, the two goals that this thesis set out to achieve have been accomplished. First, a theory of safety-critical non-interference based upon the relationship that exists between safety and security has been developed. Secondly, CSP has been applied to the modelling of Solid State Interlocking geographic databases; we have also exhibited the feasibility of the approach scaling up. It is important to note that although CSP is used for modelling, the automatic analysis itself is technology independent — other methods of verification, such as BDDs [Hu95] or SMV [McM92] might equally well be applied.

8.2 Future Work

In this section, we postulate some possible areas of future research which may follow from the work described in this thesis. We start by describing possible ideas for work in the area of non-interference and safety-critical systems, and then discuss the possibility of further automating the process of verifying SSI geographic databases.

8.2.1 Safety-Critical Non-Interference

There has, during the course of the work described in this thesis, been a great deal of input from industrial contacts.[1] As a result of this collaboration, it has been noted that there have been two recurring themes of interest.

1. The need for a greater level of 'user friendliness' from formal methods.[2]
2. The need for the incorporation of probabilistic reasoning into current formal techniques.

The first of these problems is somewhat beyond the scope of the research described in this thesis (although we shall briefly touch upon this issue with respect to the further development of our SSI verification techniques). However, the second problem is a consideration which we can address.

In Chapter 1, we saw that, in the analysis of any safety-critical system, it is always necessary, as far as possible, to verify potential specifications against the possibility of all known failures and hazards. Our approach to safety, as advocated in Chapters 2 and 4, attempts to analyse specifications in this way. However, as far as the failures-divergences model is concerned, the question of safety is a black-and-white issue. Either we can guarantee that a failure (or hazard) will not occur, or we cannot — we have no means of quantifying safety. As such, our approach to safety is somewhat limited in its applicability. In this sense, probabilistic reasoning has an enormous role to play in the formal analysis of CSP descriptions of safety-critical systems, and, if possible, we should extend our approach accordingly.

There has, in recent years been substantial research into extending the semantics of CSP to deal with probability. For example, Probabilistic Communicating Processes [Sei92] is a formalism for the specification of systems involving probabilities. The language of PCSP contains a subset of the CSP operators, as well a number of constructs which reason about

[1] In this respect, we are particularly grateful to both Don Hayward of London Underground Limited and David Mee of British Rail Research (Derby).

[2] See Chapter 1.

probability. By extending our approach to non-interference to such probabilistic models of CSP, it might be possible to discuss non-interference in terms of probabilities, and, as a result, to quantify the possibility of failures leading to hazards. Thus, an important, and potentially fruitful possible area for future research might be to investigate the feasibility of extending our theory in this direction.

Another possible area for future research might be to extend our approach to non-interference to Timed CSP [Sch90, Dav91].[3]

By extending our theory to probabilistic and timed models, we would have a formidable vehicle for the analysis of non-interference properties of complex safety-critical systems. However, without relevant mechanical support for timed probabilistic models, the potential rewards which might be reaped from such developments would appear to be rather limited.

8.2.2 The Mechanical Verification of SSI Geographic Data

The feasibility of mechanically verifying SSI data was demonstrated in Chapter 7. Although much has already been gained from this research, a lot of work remains to be done in this area if our goal of further automating the process of verifying such data is to be achieved. Future areas of work in this field are outlined below, and it is hoped that successful research might lead to a fully automated tool for the verification of SSI geographic data.

We have seen how a number of important safety invariants can be verified for SSI geographic data via FDR. The set of invariants with which we have dealt is by no means exhaustive — the aim of the research was to demonstrate the feasibility of analysing such invariants with FDR. There are many further invariants which should hold of interlockings. However, the requirements that we have investigated are representative of the type of property that must be proved of geographic data for all Solid State Interlockings.

In this respect, the first question that should be asked is stated: Is it possible to represent all SSI safety invariants in terms of CSP processes? If it is, then it may be possible to verify all such properties automatically, via FDR.

The obvious follow-up question to the above is stated: Is it computationally feasible to analyse all invariants for real interlockings? Again, a major step forward will be achieved if the answer to this question is yes, as we already have a means for analysing such invariants.

It must be remembered that even if the answers to either (or even both) of these questions are no, then there are still benefits to be gained from carrying out verification with respect to the properties and interlockings that we can analyse, as errors can be detected quickly and efficiently by carrying out such checks.

The geographic data introduced in this thesis is a simplified version of actual data: SSI geographic databases are concerned with conditional checks other than those associated with route setting, sub-route release, and points being free to move. Obviously, to provide automatic verification for any safety invariants associated with such conditions, we would have to build new, more complex models of interlockings, which could provide CSP representations of such conditional checks. As such, future research in this area might involve demonstrating that further conditional checks may be represented in terms of CSP processes. It should be noted that adding further components to our current models will in no way affect the complexity of analysing those safety invariants which we have previously described,

[3]See also [Low93] which describes extensions to the language of Timed CSP to reason about probabilities and priorities.

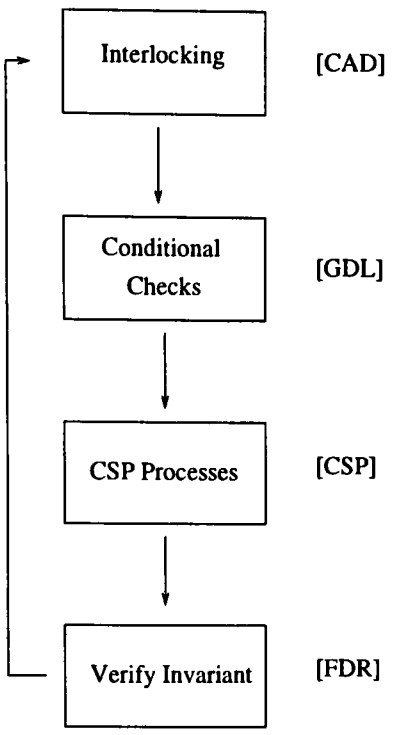

Figure 8.1: The possible integration of FDR with existing CAD technology.

as our extractions for these invariants would remain the same. Indeed, our approaches to determining extractions might be adapted so that smaller processes incorporating our new conditional checks might be chosen.

Currently, the process of translating SSI checks into CSP processes is a manual one. As long as this remains the case, questions can and will (quite correctly) be raised about the soundness of our approach. For example, errors in such translations may lead to the possibility of false positives.

We believe that it is possible to automate the process of this translation. An approach to this task was postulated in Chapter 7, and it is hoped that this shall be implemented sometime in the near future. This would drastically improve the efficiency of the task of testing such data. Of course, it would also offer the advantage of being capable of verifying data for *any* interlocking, providing that the data is in the correct form.

Another task which is currently carried out manually is that of identifying the proof obligations for verifying data. We have, however, outlined a number of algorithms, which, if implemented, could allow us to automate this process. Obviously, further work is needed on establishing the *best* extraction for each property. In addition, further invariants will require further algorithms for establishing proof obligations.

A further task which is currently carried out manually is that of composing the processes which are relevant to checking a given invariant. This is a relatively simple task, and given that the processes (and their alphabets) which are to be composed are known, this task could also be automated. Indeed, the latest version of FDR2 allows one to present it with processes to be composed, and the tool selects the best possible composition.

If we were to build upon the work of Chapter 7 in the ways outlined above, it might be possible to develop a fully automated tool for verifying SSI geographic data, which is capable of integration with FDR.

Given that signal engineers deal with Geographic Data Language and not CSP, it would, of course, be necessary to have the external interface relating information to geographic data and not CSP.

Assuming an interface with GDL constructs, capable of accessing data files for interlockings, there might be options for checks of different scales. For example, we might verify a single invariant for a particular interlocking component, verify a single invariant for a whole interlocking, or verify all invariants for a whole interlocking.

One might visualise the task of the fully automated verification of safety invariants for SSI data in the following way.

After a particular check has been chosen, e.g., safety invariant 1 for track circuit TAA, the necessary geographic checks are chosen for the verification of this invariant. There may be more than one possible combination of checks — one will be chosen. These conditional checks are then translated into the equivalent CSP processes, before being composed in the correct fashion. If this check is successful, then the engineer is informed accordingly. On the other hand, if the check fails, then an appropriate message is returned.

To conclude, in the light of the approach outlined above, it would seem that a fruitful area of future research might be to investigate further the feasibility of fully automating the verification of SSI geographic databases. Ultimately, this might lead to the integration of FDR with existing CAD technology to accomplish this task, as illustrated by Figure 8.1.

Bibliography

[Abr96] J.-R. Abrial. *The B Book.* Cambridge University Press, 1996. To appear.

[AC91] W. Atkinson and J. Cunningham. Proving properties of a safety-critical system in FOREST. *Software Engineering Journal*, 6(2):41–50, 1991.

[All91] P.G. Allen. A comparison of non-interference and non-deducibility using CSP. In *Proceedings of 1991 IEEE Computer Security Workshop.* IEEE Computer Security Press, 1991.

[BB87] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.

[BF93] R.W. Butler and G.B. Finelli. The infeasibility of quantifying the reliability of life-critical real-time software. *IEEE Transactions on Software Engineering*, 19(1):3–12, 1993.

[BFM91] R.E. Bloomfield, P.K.D. Froome, and B.Q. Monahan. Formal methods in the production and assessment of safety critical software. In B. Littlewood and D. Miller, editors, *Software Reliability and Safety*, pages 51–66. Elsevier Applied Science, 1991.

[BH94] J.P Bowen and M.G. Hinchey. Seven more myths of formal methods: Dispelling industrial prejudices. In M. Naftalin, T. Denvir, and M. Bertran, editors, *FME '94: Industrial Benefits of Formal Methods. Proceedings of Second International Symposium of Formal Methods Europe, Barcelona, Spain, October 1994*, pages 105–117. Springer-Verlag Lecture Notes in Computer Science, volume 873, 1994.

[BM92] L. Barroca and J. McDermid. Formal methods: Use and relevance for the development of safety-critical systems. *The Computer Journal*, 35(6):579–599, December 1992.

[BMD92] A. Burns, J. McDermid, and J. Dobson. On the meaning of safety and security. *The Computer Journal*, 35(1):3–15, January 1992.

[BN92] S.M. Brien and J.E. Nicholls. Z base standard version 1.0. Technical monograph PRG-107, Programming Research Group, Oxford University Computing Laboratory, University of Oxford, 1992.

[Bor87] A. Borning. Computer system reliability and nuclear war. *Communications of the ACM*, 30(2):112–131, February 1987.

[BR85] S.D. Brookes and A.W. Roscoe. An improved failures model for communicating processes. In S.D. Brookes, A.W. Roscoe, and G. Winskel, editors, *Proceedings of the NSF-SERC Seminar on Concurrency*, pages 281–305. Springer-Verlag Lecture Notes in Computer Science, volume 197, 1985.

[BRB90] SSI data preparation guide. British Railways Board Issue SSI8003, February 1990.

[BRB94] Automatic Train Protection. Report from British Railways Board to Secretary of State for Transport, March 1994.

[BS93a] J.P. Bowen and V. Stavridou. The industrial take-up of formal methods in safety-critical and other areas: A perspective. In J.C.P. Woodcock and P.G. Larsen, editors, *FME '93: Industrial-Strength Formal Methods*, pages 183–195. Springer-Verlag Lecture Notes in Computer Science, volume 670, 1993.

[BS93b] J.P. Bowen and V. Stavridou. Safety-critical systems, formal methods and standards. *IEE/BCS Software Engineering Journal*, 8(4):189–209, 1993.

[BSC92] R. Barden, S. Stepney, and D. Cooper. The use of Z. In J.E. Nicholls, editor, *Z User Workshop, York 1991*, pages 99–124. Springer-Verlag, 1992.

[CDSW95] D. Cresswell, J.W. Davies, A.C. Simpson, and J.C.P. Woodcock. Formal description and analysis of a media access protocol. Supplementary notes for IGDP Concurrency and Distributed Systems course, Department for Continuing Education, University of Oxford, 1995.

[CGR93] D. Craigen, S. Gerhart, and T. Ralston. An international survey of industrial applications of formal methods. U.S. Department of Commerce, Technology Administration National Institute of Standards and Technology Computer Systems Laboratory, Gaithersburg, MD 20899, March 1993.

[Cha86] R.N. Charette. *Software Engineering Environments: Concepts and Technology.* McGraw-Hill, 1986.

[Cla93] J. Clare. Requirements development for safety-critical systems — a total systems approach. In F. Redmill and T. Anderson, editors, *Safety-Critical Systems: Current Issues, Techniques and Standards*, pages 117–122. Chapman and Hall, 1993.

[CM92] A.H. Cribbens and I.H. Mitchell. The application of advanced computing techniques to the generation and checking of SSI data. In *Railway Engineers' Forum Meeting, London, 1991*, pages 54–66. IRSE, 1992.

[CP95] G.V. Conroy and C. Pulley. Logical methods in the formal verification of safety-critical software. In *The Mathematics of Dependable Systems.* Oxford University Press, 1995.

[CPS89a] R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench. In J. Sifakis, editor, *Proceedings of Workshop on Automatic Verification Methods for Finite State Systems, Grenoble, 1989.* Springer-Verlag Lecture Notes in Computer Science, volume 407, 1989.

[CPS89b] R. Cleaveland, J. Parrow, and B. Steffen. A semantics based verification tool for finite state systems. In *Proceedings of the Ninth International Symposium on Protocol Specification, Testing and Verification*. North Holland, 1989.

[Cri87] A.H. Cribbens. Solid State Interlocking (SSI): An integrated electronic signalling system for mainline railways. *IEE Proceedings*, 134(3):148–158, 1987.

[Dav91] J.W. Davies. Specification and proof in real-time systems. DPhil thesis, Programming Research Group, Oxford University Computing Laboratory, University of Oxford, 1991.

[dB67] J.G. de Bruijn. Additional comments on a problem in concurrent programming control. *Communications of the ACM*, 10(3):137–138, March 1967.

[Den76] D.E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[Dij65] E.W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, September 1965.

[dLSA92] R. de Lemos, A. Saeed, and T. Anderson. A train set as a case study for the requirements analysis of safety-critical systems. *The Computer Journal*, 35(2):30–40, 1992.

[DM94] B. Dehbonei and F. Mejia. Formal methods in the railways signalling industry. In M. Naftalin, T. Denvir, and M. Bertran, editors, *FME '94: Industrial Benefits of Formal Methods. Proceedings of Second International Symposium of Formal Methods Europe, Barcelona, Spain, October 1994*, pages 26–34. Springer-Verlag Lecture Notes in Computer Science, volume 873, 1994.

[DM95] B. Dehbonei and F. Mejia. Formal development of safety-critical software systems in railway signalling. In M.G. Hinchey and J.P. Bowen, editors, *Applications of Formal Methods*, pages 227–252. Prentice-Hall International, 1995.

[DoD85] DOD trusted computer system evaluation criteria. DoD 5200.28-STD, U.S. Department of Defense, 1985.

[Dod93] N. Dodd. Safety-critical system monitoring using default-trained neural networks. In F. Redmill and T. Anderson, editors, *Safety-Critical Systems: Current Issues, Techniques and Standards*, pages 228–237. Chapman and Hall, 1993.

[DS96] J.W. Davies and A.C. Simpson. The mechanical verification of fault-tolerant mutual exclusion algorithms. To appear, 1996.

[DT80] R.W. Doran and L.K. Thomas. Variants of the software solution to mutual exclusion. *Information Processing Letters*, 10(4):206–208, 1980.

[DTI94] Plan for safety analysis & general safety analysis. DTI reports IED4/1/9035, Westinghouse Signals Limited, 1994.

[EFH83] H. Ehrig, W. Fey, and H. Hansen. ACT ONE: An algebraic specification language with two levels of semantics. Technical report, Institut für Software und Theoretische Informatik, Technische Universität Berlin, 1983.

[Fin96] K. Finney. Mathematical notation in formal specification: Too difficult for the masses? *IEEE Transactions on Software Engineering*, 22(2):158–159, February 1996.

[Fol87] S.N. Foley. A universal theory of information flow. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, pages 116–122. IEEE Computer Society Press, 1987.

[Fox93] J. Fox. Engineering safety into expert systems. In F. Redmill and T. Anderson, editors, *Safety-Critical Systems: Current Issues, Techniques and Standards*, pages 202–227. Chapman and Hall, 1993.

[Gal87] A.P. Galton, editor. *Temporal Logics and Their Applications.* Academic Press, 1987.

[Gas88] M. Gasser. *Building a Secure Computer System.* Van Nostrand Reinhold, 1988.

[GC92] J. Graham-Cumming. The formal development of secure systems. DPhil thesis, Programming Research Group, Oxford University Computing Laboratory, University of Oxford, 1992.

[GM82] J.A. Goguen and J. Meseguer. Security policy and security models. In *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Computer Security Press, 1982.

[GM93] M.J.C. Gordon and T.F. Melham, editors. *Introduction to HOL.* Cambridge University Press, 1993.

[GW88] J.A. Goguen and T. Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-9, SRI International, Menlo Park, California, August 1988.

[Hal90] J.A. Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, 1990.

[Hal92] S. Hall. *BR Signalling Handbook.* Ian Allan Publishing, 1992.

[Han94] K.M. Hansen. Validation of a railway interlocking model. In M. Naftalin, T. Denvir, and M. Bertran, editors, *FME '94: Industrial Benefits of Formal Methods. Proceedings of Second International Symposium of Formal Methods Europe, Barcelona, Spain, October 1994*, pages 582–601. Springer-Verlag Lecture Notes in Computer Science, volume 873, 1994.

[Har84] D. Harel. Statecharts: A visual approach to complex systems. Technical Report CS84-05, The Wiezmann Institute of Science, Rehovot, Israel, February 1984.

[Har92] D. Harel. Biting the silver bullet: Towards a brighter future for system development. *IEEE Computer*, 25(1):8, January 1992.

[Hay85] I.J. Hayes. Applying formal specification to software development in industry. *IEEE Transactions on Software Engineering*, SE-11(2):169–178, February 1985.

[Hit92] D.K. Hitchins. *Putting systems to work.* Wiley, 1992.

[HK91] I. Houston and S. King. CICS project report: Experience and results from the use of Z in IBM. In S. Prehn and W.J. Toetenel, editors, *VDM '91, Formal Software Development Methods*, pages 588–603. Springer-Verlag Lecture Notes in Computer Science volume 551, 1991.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, 1985.

[HPSS87] D. Harel, A. Pnueli, J.P. Schmidt, and R. Sherman. On the formal semantics of statecharts. In *Proceedings of the Second IEEE Symposium on Logic in Computer Science*, pages 54–64, 1987.

[Hu95] A.J. Hu. *Efficient Techniques for Formal Verification Using Binary Decision Diagrams.* PhD thesis, Stanford University, 1995.

[Hym66] H. Hyman. Comments on a problem in concurrent programming control. *Communications of the ACM*, 9(1):45, January 1966.

[IM92] M. Ingleby and I.H. Mitchell. Proving safety of a railway signalling system incorporating geographic data. In H.H. Frey, editor, *SAFECOMP '92*, pages 129–134. Pergamon Press, 1992.

[IM95] M. Ingleby and D.J. Mee. A calculus of hazard for railway signalling. In *Proceedings of the IEEE Workshop on Industrial-Strength Formal Specification Techniques*, 1995.

[Ing95] M. Ingleby. A Galois theory of local reasoning in control systems with compositionality. In *The Mathematics of Dependable Systems.* Oxford University Press, 1995.

[ISO88] ISO8807, International Standards Organisation. *Information Processing Systems — Open Systems Interconnection — LOTOS — Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, 1988.

[Jac90] J.L. Jacob. Specifying security properties. In C.A.R. Hoare, editor, *Developments in Concurrency and Communication*, pages 221–237. ACM Press, 1990.

[Jac94] D. Jackson. Let's get back to basics. *Safety Systems: The Safety-Critical Club Newsletter*, 3(2):1–2, 1994.

[Jac95] M.J. Jackson. *Software requirements and specifications: A lexicon of practice, principles, and prejudices.* Addison-Wesley, 1995.

[Jon90] C.B. Jones. *Systematic Software Development using VDM.* Prentice-Hall International, second edition, 1990.

[JT89] D.M. Johnson and F.J. Thayer. Security properties consistent with the testing semantics for communicating processes. In *Proceedings of the 1989 IEEE Computer Security Workshop.* IEEE Computer Society Press, 1989.

[Kin94] T. King. Formalising BR's signalling rules. In M. Naftalin, T. Denvir, and M. Bertran, editors, *FME '94: Industrial Benefits of Formal Methods. Proceedings of Second International Symposium of Formal Methods Europe, Barcelona, Spain, October 1994*, pages 45–54. Springer-Verlag Lecture Computer Science, volume 873, 1994.

[Knu66] D.E. Knuth. Additional comments on a problem in concurrent computing control. *Communications of the ACM*, 9(5):321–322, May 1966.

[Lam86] L. Lamport. The mutual exclusion problem part II — statement and solutions. *Journal of the ACM*, 33(2):327–348, 1986.

[Lap89] J.C. Laprie. Dependability: A unifying concept for reliable computing and fault tolerance. In T. Anderson, editor, *Dependability of Resilient Computers*, pages 1–28. Blackwell Scientific Publications, 1989.

[Lev86] N.G. Leveson. Software safety: why, what and how. *ACM Computing Surveys*, 18(2):125–163, 1986.

[Lev91] N.G. Leveson. Software safety in embedded computer systems. *Communications of the ACM*, 34(2):34–46, February 1991.

[Lev95] N.G. Leveson. *Safeware: System Safety and Computers.* Addison-Wesley, 1995.

[Lit93] B. Littlewood. The need for evidence from disparate sources to evaluate software safety. In F. Redmill and T. Anderson, editors, *Directions in Safety-Critical Systems*, pages 217–231. Springer-Verlag, 1993.

[Low93] G. Lowe. Priorities and probabilities in Timed CSP. DPhil thesis, Programming Research Group, Oxford University Computing Laboratory, University of Oxford, 1993.

[LS87] D. Langley and M. Shain. *Data and Computer Security: Dictionary of Standards, Concepts and Terms.* Macmillan, 1987.

[May90] D. May. The use of formal methods by a silicon manufacturer. In C.A.R. Hoare, editor, *Developments in Concurrency and Communication*, pages 107–129. Addison-Wesley, 1990.

[McC87] D. McCullough. Specifications for multi-level security and a hook-up property. In *Proceedings of the 1987 Symposium on Security and Privacy*, pages 161–166. IEEE Computer Society Press, 1987.

[McC88] D. McCullough. Non-interference and the composability of security properties. In *Proceedings of the 1988 Symposium on Security and Privacy.* IEEE Computer Society Press, 1988.

[McD89] J.A. McDermid, editor. *The Theory and Practice of Refinement: Approaches to the Formal Development of Large Scale Systems.* Butterworths, 1989.

[McD91] J.A. McDermid. Formal methods: Use and relevance for the development of safety critical systems. In P.A. Bennett, editor, *Safety Aspects of Computer Control*, pages 96–153. Butterworth-Heinemann, 1991.

[McD93] J.A. McDermid. Issues in the development of safety-critical systems. In F. Redmill and T. Anderson, editors, *Safety-Critical Systems: Current Issues, Techniques and Standards*, pages 16–42. Chapman and Hall, London, 1993.

[McD94] J.A. McDermid. On dependability, its measurement and management. *High Integrity Systems*, 1(1):17–26, 1994.

[McL90] J. McLean. Security models and information flow. In *Proceedings of the 1990 Symposium on Security and Privacy.* IEEE Computer Society Press, 1990.

[McM92] K.L. McMillan. *Symbolic model checking — an approach to the state explosion problem.* PhD thesis, Carnegie Mellon University, 1992.

[Mee94] D. Mee. Possible examples for safety through security work. Personal Communication, November 1994.

[Mil80] A.J.R.G. Milner. *A Calculus of Communicating Systems.* Springer-Verlag Lecture Notes in Computer Science, volume 92, 1980.

[Mil89] A.J.R.G. Milner. *Communication and Concurrency.* Prentice-Hall International, 1989.

[MoD91a] The procurement of safety critical software in defence equipment (part 1: Requirements, part 2: Guidance). Interim Defence Standard 00-55, Issue 1, Ministry of Defence, Directorate of Standardisation, Kentigem House, Glasgow, 1991.

[MoD91b] Hazard analysis and safety classification of the computer and programmable electronic system elements of defence equipment. Interim Defence Standard 00-56, Issue 1, Ministry of Defence, Directorate of Standardisation, Kentigem House, Glasgow, 1991.

[Mor91] M.J. Morley. Modelling British Rail's interlocking logic: Geographic data correctness. Technical Report ECS-LFCS-91-186, Department of Computer Science, University of Edinburgh, 1991.

[Mor93] M.J. Morley. Safety in railway signalling data: A behavioural analysis. In *6th Annual Workshop on Higher Order Logic and its Applications*, pages 464–474, 1993.

[Nee92] C. Neesham. Safe conduct. *Computing*, pages 18–20, 1992.

[Neu95] P.G. Neumann. *Computer Related Risks.* Addison-Wesley, 1995.

[O'H90] C. M. O'Halloran. A calculus of information flow. In *Proceedings of the 1990 European Symposium on Research in Computer Security*, pages 147–159. AFCET, 1990.

[Par95] D.L. Parnas. Using mathematical models in the inspection of critical software. In M.G. Hinchey and J.P. Bowen, editors, *Applications of Formal Methods*, pages 17–32. Prentice-Hall International, 1995.

[Pet81] G.L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, June 1981.

[Pla93] N. Plat. *Experiments with Formal Methods in Software Engineering.* PhD thesis, Technische Universiteit Delft, 1993.

[PS85] J.L. Peterson and A. Silberschatz. *Operating system concepts.* Addison-Wesley, second edition, 1985.

[Qui93] B. Quirk. The FOREST technique for system specification and validation. In F. Redmill and T. Anderson, editors, *Safety-Critical Systems: Current Issues, Techniques and Standards*, pages 102–116. Chapman and Hall, 1993.

[Ray86] M. Raynal. *Algorithms for Mutual Exclusion.* North Oxford, 1986.

[Rea79] J. Reason. Action not as planned: the price of automization. In G. Underwood and R. Stevens, editors, *Aspects of Consciousness.* Academic Press, 1979.

[Red93] F. Redmill. Software in safety-critical applications — a review of current issues. In F. Redmill and T. Anderson, editors, *Safety-Critical Systems: Current Issues, Techniques and Standards*, pages 1–15. Chapman and Hall, 1993.

[Ree88] G.M. Reed. A uniform mathematical theory for real-time distributed computing. DPhil thesis, Programming Research Group, Oxford University Computing Laboratory, University of Oxford, 1988.

[Rei85] W. Reisig. *Petri nets: An introduction.* Springer-Verlag, 1985.

[RGG+95] A.W. Roscoe, P.H.B. Gardiner, M.H. Goldsmith, J.R. Hulance, D.M. Jackson, and J.B. Scattergood. Hierarchical model-checking for CSP. In *Proceedings of TACAS '95.* Springer-Verlag, 1995.

[Ros94] A.W. Roscoe. Model checking CSP. In A.W. Roscoe, editor, *A Classical Mind: Essays in honour of C.A.R. Hoare*, pages 353–378. Prentice-Hall International, 1994.

[Ros95] A.W. Roscoe. CSP and determinism in security modelling. In *IEEE Symposium on Security and Privacy*, 1995.

[Rus89] J.M. Rushby. Kernels for safety. In T. Anderson, editor, *Safe and Secure Computer Systems*, pages 210–220. Blackwell Scientific Publications, 1989.

[Rus93] J. Rushby. Formal methods and the certification of critical systems. Technical Report SRI-CSL-93-07, Computer Science Laboratory, SRI International, November 1993.

[RWW94] A.W. Roscoe, J.C.P. Woodcock, and L. Wulf. Non-interference through determinism. In *Proceedings of ESORICS '94*, pages 33–53. Springer-Verlag Lecture Notes in Computer Science, volume 875, 1994.

[Rya91] P.Y.A. Ryan. A CSP formulation of non-interference. In *Cipher*, pages 19–27. IEEE Computer Society Press, 1991.

[Sch90] S.A. Schneider. Correctness and communication in real-time systems. DPhil thesis, Programming Research Group, Oxford University Computing Laboratory, University of Oxford, 1990.

[Sei92] K. Seidel. Probabilistic Communicating Processes. DPhil thesis, Programming Research Group, Oxford University Computing Laboratory, University of Oxford, 1992.

[Sen89] C.T. Sennett. *High-integrity Software.* Pitman, 1989.

[Sim94] A.C. Simpson. A formal specification of an Automatic Train Protection system. In M. Naftalin, T. Denvir, and M. Bertran, editors, *FME '94: Industrial Benefits of Formal Methods. Proceedings of Second International Symposium of Formal Methods Europe, Barcelona, Spain, October 1994*, pages 602–617. Springer-Verlag Lecture Notes in Computer Science, volume 873, 1994.

[Sim95a] A.C. Simpson. The application of formal methods to the development of an ATP (Automatic Train Protection) system. In *Colloquium on Communication Networks in Transportation.* IEE, 1995.

[Sim95b] A.C. Simpson. Railway signalling as a case study in safety through determinism. In *Proceedings of Aspect '95.* IRSE, 1995.

[Sim96] A.C. Simpson. Non-interference and safety-critical systems. Supplementary notes for IGDP Critical Systems Engineering course, Department for Continuing Education, University of Oxford, February 1996.

[Spi92] J.M. Spivey. *The Z Notation: A Reference Manual.* Prentice-Hall International, second edition, 1992.

[Sut86] D. Sutherland. A model of information. In *Proceedings of the 9th National Computer Security Conference*, pages 175–183. U.S. National Computer Security Center and U.S. National Bureau of Standards, 1986.

[SW96] A.C. Simpson and J.C.P. Woodcock. Safety through determinism. To appear, 1996.

[SWD96] A.C. Simpson, J.C.P. Woodcock, and J.W. Davies. The mechanical verification of Solid State Interlocking data. To appear, 1996.

[Tho93] M.C. Thomas. The industrial use of formal methods. *Microprocessors and Microsystems*, 17(1):31–36, 1993.

[TRC93] M. Thompson, D. Richardson, and L. Clarke. An information flow model of fault detection. In T. Ostrand and E. Weyuker, editors, *Proceedings of the International Symposium on Software Testing and Analysis*, pages 182–192. ACM Press, June 1993.

[Wal88] D.J. Walker. Analysing mutual exclusion algorithms using CCS. Technical Report ECS-LFCS-88-45, Department of Computer Science, University of Edinburgh, January 1988.

[Wal89a] D.J. Walker. Automated analysis of mutual exclusion algorithms using CCS. *Formal Aspects of Computing*, 1(3):273–292, 1989.

[Wal89b] D.J. Walker. Automated analysis of mutual exclusion algorithms using CCS. Technical Report ECS-LFCS-89-91, Department of Computer Science, University of Edinburgh, August 1989.

[WD96] J.C.P. Woodcock and J.W. Davies. *Using Z: Specification, Refinement, and Proof.* Prentice-Hall International, 1996.

[Web96] M. Weber. Combining statecharts and Z for the design of safety-critical control systems. In M-C. Gaudel and J.C.P. Woodcock, editors, *FME '96: Industrial Benefits and Advances in Formal Methods. Proceedings of Third International Symposium of Formal Methods Europe, Oxford, UK, March 1996*, pages 307–326. Springer-Verlag Lecture Computer Science, volume 1057, 1996.

[WGH94] J.C.P. Woodcock, P.H.B. Gardiner, and J.R. Hulance. The formal specification in Z of Defence Standard 00-56. In J.P. Bowen and J.A. Hall, editors, *Proceedings of Eighth Z User Workshop, Cambridge.* Springer-Verlag, 1994.

[Win90] J.M. Wing. A specifier's introduction to formal methods. *IEEE Computer*, 23(9):8–22, September 1990.

[WKC91] D.R. Wallace, D.R. Kuhn, and J.C. Cherniavsky. Report of the NIST workshop of standards for the assurance of high integrity software. NIST Special Publication 500-190, Computer Systems Laboratory, National Institute of Standards and Technology, Gaithersburg, Maryland, August 1991.

[Wul94] L. Wulf. Non-interference and determinism in system specification. MSc to DPhil transfer dissertation, Programming Research Group, Oxford University Computing Laboratory, University of Oxford, December 1994.

[WW93] D. Weber-Wulf. Selling formal methods to industry. In J.C.P. Woodcock and P.G. Larsen, editors, *FME '93: Industrial-Strength Formal Methods*, pages 671–678. Springer-Verlag Lecture Notes in Computer Science, volume 670, 1993.

[WZ91] J.M. Wing and A.M. Zaremski. Unintrusive ways to integrate formal specifications in practice. In S. Prehn and W.J. Toetenel, editors, *VDM '91, Formal Software Development Methods*, pages 547–569. Springer-Verlag Lecture Notes in Computer Science, volume 551, 1991.

Appendix A

Related Proofs

In this appendix, we present proofs related to the laws of Chapter 3.

General Laws

Law A.1

$$\frac{s_1 \nleftarrow s_2\ [P] \qquad s_3 \subseteq s_1}{s_3 \nleftarrow s_2\ [P]}$$

Proof. Assume $s_1 \nleftarrow s_2\ [P]$ and $s_3 \subseteq s_1$. By Definition 2.6,

$$\exists\, s : \mathbb{P}\Sigma \bullet s \cup s_1 \Leftarrow (\alpha P - (s \cup s_1)) \cup s_2\ [P]$$

As we can find a set $s \in \mathbb{P}\Sigma$ such that the above holds and $s_3 \subseteq s_1$, then it follows that, as $\alpha P - s_3$ is a superset of $\alpha P - s_1$, we can establish a set $s \in \mathbb{P}\Sigma$ such that

$$\exists\, s : \mathbb{P}\Sigma \bullet s \cup s_3 \Leftarrow (\alpha P - (s \cup s_3)) \cup s_2\ [P]$$

□

Law A.2

$$\frac{s_1 \nleftarrow s_2\ [P] \qquad s_3 \subseteq s_2}{s_1 \nleftarrow s_3\ [P]}$$

Proof. By a similar argument as that for Law A.1. □

Law A.3

$$\frac{s_1 \leftarrow\!\!+ s_2\ [P] \quad s_3 \subseteq s_1 \quad s_4 \subseteq s_2}{s_3 \leftarrow\!\!+ s_4\ [P]}$$

Proof.

$s_1 \leftarrow\!\!+ s_2\ [P]$	[Assumption]	(1)
$s_3 \subseteq s_1$	[Assumption]	(2)
$s_3 \leftarrow\!\!+ s_2\ [P]$	[1, 2, Law A.1]	(3)
$s_4 \subseteq s_2$	[Assumption]	(4)
$s_3 \leftarrow\!\!+ s_4\ [P]$	[3, 4, Law A.2]	(5)

□

Proposition A.1

$$\forall P : PROC;\ s_1, s_2 : \mathbb{P}\,\Sigma \bullet (\mathbf{det}(P \,|||\, RUN_{s_1}) \wedge \mathbf{det}(P \,|||\, RUN_{s_2})) \Rightarrow \mathbf{det}(P \,|||\, RUN_{s_1 \cup s_2})$$

Proof. Trivial. □

Law A.4

$$\frac{s_1 \not\Leftarrow s_2\ [P] \quad s_1 \not\Leftarrow s_3\ [P]}{s_1 \leftarrow\!\!+ s_2 \cup s_3\ [P]}$$

Proof.

$s_1 \not\Leftarrow s_2\ [P]$	[Assumption]	(1)
$\alpha P \subseteq s_1 \cup s_2 \wedge s_1 \cap s_2 = \varnothing \wedge \mathbf{det}(P \,|||\, RUN_{s_2})$	[1, Definition 2.5]	(2)
$s_1 \not\Leftarrow s_3\ [P]$	[Assumption]	(3)
$\alpha P \subseteq s_1 \cup s_3 \wedge s_1 \cap s_3 = \varnothing \wedge \mathbf{det}(P \,|||\, RUN_{s_3})$	[3, Definition 2.5]	(4)
$\alpha P \subseteq s_1 \cup (s_2 \cup s_3) \wedge s_1 \cap (s_2 \cup s_3) = \varnothing \wedge$ $\mathbf{det}(P \,|||\, RUN_{s_2}) \wedge \mathbf{det}(P \,|||\, RUN_{s_3})$	[2, 4, Trivial]	(5)
$\alpha P \subseteq s_1 \cup (s_2 \cup s_3) \wedge$ $s_1 \cap (s_2 \cup s_3) = \varnothing \wedge \mathbf{det}(P \,|||\, RUN_{s_2 \cup s_3})$	[4, 5, Proposition A.1]	(6)
$s_1 \not\Leftarrow s_2 \cup s_3\ [P]$	[6, Definition 2.5]	(7)
$s_1 \leftarrow\!\!+ s_2 \cup s_3\ [P]$	[7, Proposition 2.4]	(8)

□

Law A.5

$$\frac{s_1 \Leftarrow\!\!\!\!\!\!+\; s_3\ [P] \qquad s_2 \Leftarrow\!\!\!\!\!\!+\; s_3\ [P]}{s_1 \cup s_2 \Leftarrow\!\!\!\!\!\!+\; s_3\ [P]}$$

Proof.

$s_1 \Leftarrow\!\!\!\!\!\!+\; s_3\ [P]$	[Assumption]	(1)
$\alpha P \subseteq s_1 \cup s_3 \wedge s_1 \cap s_3 = \varnothing \wedge \mathbf{det}(P \,\|\|\|\, RUN_{s_3})$	[1, Definition 2.5]	(2)
$s_2 \Leftarrow\!\!\!\!\!\!+\; s_3\ [P]$	[Assumption]	(3)
$\alpha P \subseteq s_2 \cup s_3 \wedge s_2 \cap s_3 = \varnothing \wedge \mathbf{det}(P \,\|\|\|\, RUN_{s_3})$	[3, Definition 2.5]	(4)
$\alpha P \subseteq (s_1 \cup s_2) \cup s_3 \wedge (s_1 \cup s_2) \cap s_3 = \varnothing \wedge \mathbf{det}(P \,\|\|\|\, RUN_{s_3})$	[2, 4, Trivial]	(5)
$s_1 \cup s_2 \Leftarrow\!\!\!\!\!\!+\; s_3\ [P]$	[5, Definition 2.5]	(6)
$s_1 \cup s_2 \Leftarrow\!\!\!\!\!\!+\; s_3\ [P]$	[6, Proposition 2.4]	(7)

□

Event Prefixing

Proposition A.2

$$\forall\, P : PROC;\ s : \mathbb{P}\,\Sigma;\ e : \Sigma \bullet \mathbf{det}(P \,|||\, RUN_s) \wedge e \notin s \Rightarrow \mathbf{det}(e \rightarrow P \,|||\, RUN_s)$$

Proof. Trivial. □

Law A.6

$$\frac{s_1 \Leftarrow\!\!\!\!\!\!+\; s_2\ [P] \qquad e \notin s_2}{s_1 \Leftarrow\!\!\!\!\!\!+\; s_2\ [e \rightarrow P]}$$

Proof.

$s_1 \Leftarrow\!\!\!\!\!\!+\; s_2\ [P]$	[Assumption]	(1)
$\alpha P \subseteq s_1 \cup s_2 \wedge s_1 \cap s_2 = \varnothing \wedge \mathbf{det}(P \,\|\|\|\, RUN_{s_2})$	[1, Definition 2.5]	(2)
$e \notin s_2$	[Assumption]	(3)
$\alpha P \subseteq s_1 \cup s_2 \wedge s_1 \cap s_2 = \varnothing \wedge \mathbf{det}(e \rightarrow P \,\|\|\|\, RUN_{s_2})$	[2, 3, Proposition A.2]	(4)
$\alpha(e \rightarrow P) = \alpha P$	[Semantics of $\rightarrow$]	(5)
$\alpha(e \rightarrow P) \subseteq s_1 \cup s_2 \wedge s_1 \cap s_2 = \varnothing \wedge \mathbf{det}(e \rightarrow P \,\|\|\|\, RUN_{s_2})$	[4, 5, Substitution]	(6)
$s_1 \Leftarrow\!\!\!\!\!\!+\; s_2\ [e \rightarrow P]$	[6, Definition 2.5]	(7)

□

Internal Choice

Law A.7

$$\frac{\begin{array}{c} s_1 \cap s_2 = \varnothing \\ \alpha(P \sqcap Q) \subseteq s_2 \\ \mathbf{nondiv}(P \sqcap Q) \end{array}}{s_1 \Leftarrow s_2\ [P \sqcap Q]}$$

Proof.

$s_1 \cap s_2 = \varnothing$	[Assumption]	(1)
$\alpha(P \sqcap Q) \subseteq s_2$	[Assumption]	(2)
$\mathbf{nondiv}(P \sqcap Q)$	[Assumption]	(3)
$\mathbf{det}(P \sqcap Q \mid\mid\mid RUN_{s_2})$	[2, 3, Proposition 3.3]	(4)
$\alpha(P \sqcap Q) \subseteq s_1 \cup s_2 \wedge$		
$s_1 \cap s_2 = \varnothing \wedge \mathbf{det}(P \sqcap Q \mid\mid\mid RUN_{s_2})$	[1, 2, 4, $\wedge$ introduction]	(5)
$s_1 \Leftarrow s_2\ [P \sqcap Q]$	[5, Definition 2.5]	(6)

□

External Choice

Proposition A.3

$$\begin{array}{l} \forall\, P, Q : PROC;\ s : \mathbb{P}\,\Sigma;\ e : \Sigma \bullet \\ \quad (\mathbf{det}(P \mid\mid\mid RUN_s) \wedge \mathbf{det}(Q \mid\mid\mid RUN_s) \wedge \\ \quad (e \in inits(P) \cap inits(Q) \Rightarrow P \textbf{ after } \langle e \rangle = Q \textbf{ after } \langle e \rangle)) \Rightarrow \\ \qquad \mathbf{det}(P \,\Box\, Q \mid\mid\mid RUN_s) \end{array}$$

Proof. Trivial. □

As a corollary to the above, we have the following.

Proposition A.4

$$\begin{array}{l} \forall\, P : PROC;\ s : \mathbb{P}\,\Sigma;\ e : \Sigma \bullet \\ \quad \mathbf{det}(P \mid\mid\mid RUN_s) \wedge e \notin inits(P) \Rightarrow \mathbf{det}(P \,\Box\, e \rightarrow P \mid\mid\mid RUN_s) \end{array}$$

Proof. Trivial. □

Law A.8

$$\frac{\begin{array}{c} s_1 \not\Leftarrow s_2\ [P] \\ s_1 \not\Leftarrow s_2\ [Q] \\ \forall\, e : \Sigma \bullet e \in \mathit{inits}(P) \cap \mathit{inits}(Q) \Rightarrow P \textbf{ after } \langle e \rangle = Q \textbf{ after } \langle e \rangle \end{array}}{s_1 \not\Leftarrow s_2\ [P \,\Box\, Q]}$$

Proof.

$s_1 \not\Leftarrow s_2\ [P]$	[Assumption]	(1)
$\alpha P \subseteq s_1 \cup s_2 \wedge s_1 \cap s_2 = \varnothing \wedge \textbf{det}(P \,\|\|\|\, RUN_{s_2})$	[1, Definition 2.5]	(2)
$s_1 \not\Leftarrow s_2\ [Q]$	[Assumption]	(3)
$\alpha Q \subseteq s_1 \cup s_2 \wedge s_1 \cap s_2 = \varnothing \wedge \textbf{det}(Q \,\|\|\|\, RUN_{s_2})$	[3, Definition 2.5]	(4)
$\alpha(P \,\Box\, Q) \subseteq s_1 \cup s_2 \wedge s_1 \cap s_2 = \varnothing \wedge$ $\textbf{det}(P \,\|\|\|\, RUN_{s_2}) \wedge \textbf{det}(Q \,\|\|\|\, RUN_{s_2})$	[2, 4, $\wedge$ introduction]	(5)
$\forall\, e : \Sigma \bullet$ $e \in \mathit{inits}(P) \cap \mathit{inits}(Q) \Rightarrow P \textbf{ after } \langle e \rangle = Q \textbf{ after } \langle e \rangle$	[Assumption]	(6)
$\alpha(P \,\Box\, Q) \subseteq s_1 \cup s_2 \wedge$ $s_1 \cap s_2 = \varnothing \wedge \textbf{det}(P \,\Box\, Q \,\|\|\|\, RUN_{s_2})$	[5, 6, Proposition A.3]	(7)
$s_1 \not\Leftarrow s_2\ [P \,\Box\, Q]$	[7, Definition 2.5]	(8)

□

Law A.9

$$\frac{\begin{array}{c} s_1 \not\Leftarrow s_2\ [P] \\ e \notin \mathit{inits}(P) \end{array}}{s_1 \not\Leftarrow s_2\ [P \,\Box\, e \rightarrow P]}$$

Proof.

$s_1 \not\Leftarrow s_2\ [P]$	[Assumption]	(1)
$\alpha P \subseteq s_1 \cup s_2 \wedge s_1 \cap s_2 = \varnothing \wedge \textbf{det}(P \,\|\|\|\, RUN_{s_2})$	[1, Definition 2.5]	(2)
$e \notin \mathit{inits}(P)$	[Assumption]	(3)
$\alpha P \subseteq s_1 \cup s_2 \wedge s_1 \cap s_2 = \varnothing \wedge \textbf{det}(P \,\Box\, e \rightarrow P \,\|\|\|\, RUN_{s_2})$	[2, 3, Proposition A.4]	(4)
$\alpha(P \,\Box\, e \rightarrow P) = \alpha P$	[Semantics of $\rightarrow$ and $\Box$]	(5)
$\alpha(P \,\Box\, e \rightarrow P) \subseteq s_1 \cup s_2 \wedge s_1 \cap s_2 = \varnothing \wedge$ $\textbf{det}(P \,\Box\, e \rightarrow P \,\|\|\|\, RUN_{s_2})$	[4, 5, Proposition A.4]	(6)
$s_1 \not\Leftarrow s_2\ [P \,\Box\, e \rightarrow P]$	[6, Definition 2.5]	(7)

□

Law A.10

$$\frac{\begin{array}{c} s_1 \Leftarrow s_2\ [P] \\ s_1 \Leftarrow s_2\ [Q] \\ x, y \notin s_2 \\ x \neq y \end{array}}{s_1 \Leftarrow s_2\ [x \rightarrow P \,\Box\, y \rightarrow Q]}$$

Proof.

$s_1 \Leftarrow s_2\ [P]$	[Assumption]	(1)
$\alpha P \subseteq s_1 \cup s_2 \wedge s_1 \cap s_2 = \varnothing \wedge \mathbf{det}(P \,\|\|\|\, RUN_{s_2})$	[1, Definition 2.5]	(2)
$x, y \notin s_2$	[Assumption]	(3)
$\alpha(x \rightarrow P) \subseteq s_1 \cup s_2 \wedge$		
$s_1 \cap s_2 = \varnothing \wedge \mathbf{det}(x \rightarrow P \,\|\|\|\, RUN_{s_2})$	[2, 3, Proposition A.2]	(4)
$s_1 \Leftarrow s_2\ [Q]$	[Assumption]	(5)
$\alpha Q \subseteq s_1 \cup s_2 \wedge s_1 \cap s_2 = \varnothing \wedge \mathbf{det}(Q \,\|\|\|\, RUN_{s_2})$	[5, Definition 2.5]	(6)
$\alpha(y \rightarrow Q) \subseteq s_1 \cup s_2 \wedge$		
$s_1 \cap s_2 = \varnothing \wedge \mathbf{det}(y \rightarrow Q \,\|\|\|\, RUN_{s_2})$	[3, 5, Proposition A.2]	(7)
$x \neq y$	[Assumption]	(8)
$\alpha(x \rightarrow P \,\Box\, y \rightarrow Q) \subseteq s_1 \cup s_2 \wedge$		
$s_1 \cap s_2 = \varnothing \wedge \mathbf{det}(x \rightarrow P \,\Box\, y \rightarrow Q \,\|\|\|\, RUN_{s_2})$	[4, 7, 8, Proposition A.3]	(9)
$s_1 \Leftarrow s_2\ [x \rightarrow P \,\Box\, y \rightarrow Q]$	[9, Definition 2.5]	(10)

□

Law A.11

$$\frac{\begin{array}{c} s_1 \Leftarrow s_2\ [P] \\ X \cap \mathit{inits}(P) = \varnothing \end{array}}{s_1 \Leftarrow s_2 [(\Box_{x \in X}\, x \rightarrow P) \,\Box\, P]}$$

Proof.

$s_1 \Leftarrow s_2\ [P]$	[Assumption]	(1)
$X \cap \mathit{inits}(P) = \varnothing$	[Assumption]	(2)
$\forall x : X \bullet s_1 \Leftarrow s_2\ [x \rightarrow P \,\Box\, P]$	[1, 2, Law A.9]	(3)
$\forall x : X \bullet s_1 \Leftarrow s_2\ [\Box_{x \in X}(x \rightarrow P \,\Box\, P)]$	[3, Law A.8]	(4)
$s_1 \Leftarrow s_2\ [(\Box_{x \in X}\, x \rightarrow P) \,\Box\, P]$	[4, associativity & commutativity of $\Box$]	(5)

□

Law A.12

$$\frac{s_1 \Leftarrow s_2\ [P] \qquad X \cap s_2 = \varnothing}{s_1 \Leftarrow s_2\ [\Box_{x \in X}\, x \rightarrow P]}$$

Proof.

$s_1 \Leftarrow s_2\ [P]$	[Assumption]	(1)
$X \cap s_2 = \varnothing$	[Assumption]	(2)
$\forall x : X \bullet s_1 \Leftarrow s_2\ [x \rightarrow P]$	[1, 2, Law A.6]	(3)
$s_1 \Leftarrow s_2\ [\Box_{x \in X}\, x \rightarrow P]$	[3, Law A.8]	(4)

□

Partial Interleaving

Lemma A.1

$$\forall P, Q : PROC;\ A : \mathbb{P}\,\Sigma;\ tr : \mathrm{seq}\,\Sigma \mid tr \in traces\,[\![P \,[\![A]\!]\, Q]\!] \bullet$$
$$\overline{inits}(P \,[\![A]\!]\, Q \textbf{ after } tr) =$$
$$\{x : \Sigma \mid \exists\, t_1 \in traces\,[\![P]\!];\ t_2 \in traces\,[\![Q]\!] \bullet$$
$$tr \in t_1 \,[\![A]\!]\, t_2 \wedge x \in \overline{inits}((P, Q)_A \textbf{ after } (t_1, t_2))\}$$

Proof. By our definition of the $[\![A]\!]$ function on traces. □

Proposition A.5

$$\forall P, Q : PROC;\ A, s : \mathbb{P}\,\Sigma \bullet$$
$$(\textbf{det}(P \,|||\, RUN_s) \wedge \textbf{det}(Q \,|||\, RUN_s) \wedge \textbf{det}(P \,[\![A]\!]\, Q)) \Rightarrow \textbf{det}((P \,[\![A]\!]\, Q) \,|||\, RUN_s)$$

Proof. Assume $\textbf{det}(P \,[\![A]\!]\, Q)$. By Definition 1.1, $\textbf{nondiv}(P \,[\![A]\!]\, Q)$.
Assume $\textbf{det}(P \,|||\, RUN_s)$. By Proposition 3.1, $\textbf{nondiv}(P)$ and

$$\forall t_1, t_2 : \mathrm{seq}\,\Sigma;\ e : \Sigma \bullet$$
$$t_1 \frown t_2 \in traces\,[\![P]\!] \wedge t_1 \frown \langle e \rangle \frown t_2 \in traces\,[\![P]\!] \wedge e \in s \Rightarrow$$
$$(\Sigma - s) \cap \overline{inits}(P \textbf{ after } t_1 \frown \langle e \rangle \frown t_2) = (\Sigma - s) \cap \overline{inits}(P \textbf{ after } t_1 \frown t_2)$$
$$\wedge$$
$$t_1 \frown t_2 \in traces\,[\![P]\!] \wedge t_1 \frown \langle e \rangle \frown t_2 \in traces\,[\![P]\!] \wedge e \notin s \Rightarrow$$
$$(t_1, \{e\}) \notin failures\,[\![P]\!]$$

Assume $\textbf{det}(Q \,|||\, RUN_s)$. By Proposition 3.1, $\textbf{nondiv}(Q)$ and

$$\forall t_1, t_2 : \mathrm{seq}\,\Sigma;\ e : \Sigma \bullet$$
$$t_1 \frown t_2 \in traces\,[\![Q]\!] \wedge t_1 \frown \langle e \rangle \frown t_2 \in traces\,[\![Q]\!] \wedge e \in s \Rightarrow$$
$$(\Sigma - s) \cap \overline{inits}(Q \textbf{ after } t_1 \frown \langle e \rangle \frown t_2) = (\Sigma - s) \cap \overline{inits}(Q \textbf{ after } t_1 \frown t_2)$$
$$\wedge$$
$$t_1 \frown t_2 \in traces\,[\![Q]\!] \wedge t_1 \frown \langle e \rangle \frown t_2 \in traces\,[\![Q]\!] \wedge e \notin s \Rightarrow$$
$$(t_1, \{e\}) \notin failures\,[\![Q]\!]$$

Assume $t_1 \frown \langle e \rangle \in traces\,[\![P \,[\![A]\!]\, Q]\!]$. There are two cases to consider: $e \in s$ and $e \notin s$.

Case 1 Assume $e \in s$. For every trace t_2 such that

$$t_2 \in \mathit{traces}[\![(P \,|[\, A \,]|\, Q) \textbf{ after } t_1 \frown \langle e \rangle]\!] \cap \mathit{traces}[\![(P \,|[\, A \,]|\, Q) \textbf{ after } t_1]\!]$$

there must be traces $s_1 \in \mathit{traces}[\![P]\!]$, $s_2 \in \mathit{traces}[\![P \textbf{ after } s_1 \frown \langle e \rangle]\!]$, $u_1 \in \mathit{traces}[\![Q]\!]$ and $u_2 \in \mathit{traces}[\![Q \textbf{ after } u_1 \frown \langle e \rangle]\!]$, such that $t_1 \in s_1 \,|[\, A \,]|\, u_1$ and $t_2 \in s_2 \,|[\, A \,]|\, u_2$. It follows that

$$\begin{array}{l}
(\Sigma - s) \cap \overline{\mathit{inits}}(P \,|[\, A \,]|\, Q \textbf{ after } t_1 \frown \langle e \rangle \frown t_2) \\
= \\
(\Sigma - s) \cap (\overline{\mathit{inits}}(P \textbf{ after } s_1 \frown \langle e \rangle \frown s_2) \cap A \\
\qquad\qquad \cup \\
\qquad\qquad \overline{\mathit{inits}}(Q \textbf{ after } u_1 \frown \langle e \rangle \frown u_2) \cap A \\
\qquad\qquad \cup \\
\qquad\qquad \overline{\mathit{inits}}(P \textbf{ after } s_1 \frown \langle e \rangle \frown s_2) \cap \overline{\mathit{inits}}(Q \textbf{ after } u_1 \frown \langle e \rangle \frown u_2))
\end{array}$$

Distributing $\cap$ through $\cup$,

$$\begin{array}{l}
(\Sigma - s) \cap \overline{\mathit{inits}}(P \,|[\, A \,]|\, Q \textbf{ after } t_1 \frown \langle e \rangle \frown t_2) \\
= \\
(\Sigma - s) \cap \overline{\mathit{inits}}(P \textbf{ after } s_1 \frown \langle e \rangle \frown s_2) \cap A \\
\cup \\
(\Sigma - s) \cap \overline{\mathit{inits}}(Q \textbf{ after } u_1 \frown \langle e \rangle \frown u_2) \cap A \\
\cup \\
(\Sigma - s) \cap \overline{\mathit{inits}}(P \textbf{ after } s_1 \frown \langle e \rangle \frown s_2) \cap \overline{\mathit{inits}}(Q \textbf{ after } u_1 \frown \langle e \rangle \frown u_2)
\end{array}$$

By our assumptions,

$$\begin{array}{rl}
& (\Sigma - s) \cap \overline{\mathit{inits}}(P \,|[\, A \,]|\, Q \textbf{ after } t_1 \frown \langle e \rangle \frown t_2) \\
= & (\Sigma - s) \cap \overline{\mathit{inits}}(P \textbf{ after } s_1 \frown s_2) \cap A \\
& \cup \\
& (\Sigma - s) \cap \overline{\mathit{inits}}(Q \textbf{ after } u_1 \frown u_2) \cap A \\
& \cup \\
& (\Sigma - s) \cap \overline{\mathit{inits}}(P \textbf{ after } s_1 \frown s_2) \cap \overline{\mathit{inits}}(Q \textbf{ after } u_1 \frown u_2)
\end{array}$$

Again, through the distributivity of $\cap$ through $\cup$,

$$\begin{array}{rl}
& (\Sigma - s) \cap \overline{\mathit{inits}}(P \,|[\, A \,]|\, Q \textbf{ after } t_1 \frown \langle e \rangle \frown t_2) \\
= & (\Sigma - s) \cap (\overline{\mathit{inits}}(P \textbf{ after } s_1 \frown s_2) \cap A \\
& \qquad \cup \\
& \qquad \overline{\mathit{inits}}(Q \textbf{ after } u_1 \frown u_2) \cap A \\
& \qquad \cup \\
& \qquad \overline{\mathit{inits}}(P \textbf{ after } s_1 \frown s_2) \cap \overline{\mathit{inits}}(Q \textbf{ after } u_1 \frown u_2))
\end{array}$$

Finally,

$$\begin{array}{rl}
& (\Sigma - s) \cap \overline{\mathit{inits}}(P \,|[\, A \,]|\, Q \textbf{ after } t_1 \frown \langle e \rangle \frown t_2) \\
= & (\Sigma - s) \cap \overline{\mathit{inits}}(P \,|[\, A \,]|\, Q \textbf{ after } t_1 \frown t_2)
\end{array}$$

Case 2 Assume $e \notin s$. Because the process $P \,[\![\, A \,]\!]\, Q$ is deterministic, it follows that

$$(t_1, \{e\}) \notin \mathit{failures}\,[\![P \,[\![\, A \,]\!]\, Q]\!]$$

Trivially therefore,

$$e \notin s \Rightarrow (t_1, \{e\}) \notin \mathit{failures}\,[\![P \,[\![\, A \,]\!]\, Q]\!]$$

Therefore, by Proposition 3.1, $\mathbf{det}((P \,[\![\, A \,]\!]\, Q) \,|||\, RUN_s)$. □

It follows that our non-interference law for partial interleaving is stated thus:

Law A.13

$$\frac{\begin{array}{c} s_1 \not\Leftarrow s_2\ [P] \\ s_1 \not\Leftarrow s_2\ [Q] \\ \mathbf{det}(P \,[\![\, A \,]\!]\, Q) \end{array}}{s_1 \not\Leftarrow s_2\ [P \,[\![\, A \,]\!]\, Q]}$$

Proof.

$s_1 \not\Leftarrow s_2\ [P]$	[Assumption]	(1)
$\alpha P \subseteq s_1 \cup s_2 \wedge s_1 \cap s_2 = \varnothing \wedge \mathbf{det}(P \,\|||\, RUN_{s_2})$	[1, Definition 2.5]	(2)
$s_1 \not\Leftarrow s_2\ [Q]$	[Assumption]	(3)
$\alpha Q \subseteq s_1 \cup s_2 \wedge s_1 \cap s_2 = \varnothing \wedge \mathbf{det}(Q \,\|||\, RUN_{s_2})$	[3, Definition 2.5]	(4)
$\alpha(P \,[\![\, A \,]\!]\, Q) \subseteq s_1 \cup s_2 \wedge s_1 \cap s_2 = \varnothing \wedge$ $\mathbf{det}(P \,\|||\, RUN_{s_2}) \wedge \mathbf{det}(Q \,\|||\, RUN_{s_2})$	[2, 4, $\wedge$ introduction]	(5)
$\mathbf{det}(P \,[\![\, A \,]\!]\, Q)$	[Assumption]	(6)
$\alpha(P \,[\![\, A \,]\!]\, Q) \subseteq s_1 \cup s_2 \wedge$ $s_1 \cap s_2 = \varnothing \wedge \mathbf{det}((P \,[\![\, A \,]\!]\, Q) \,|||\, RUN_{s_2})$	[5, 6, Proposition A.5]	(7)
$s_1 \not\Leftarrow s_2\ [P \,[\![\, A \,]\!]\, Q]$	[7, Definition 2.5]	(8)

□

Parallel Composition

Proposition A.6

$$\forall P, Q : PROC;\ s : \mathbb{P}\,\Sigma \bullet$$
$$\mathbf{det}(P \,|||\, RUN_s) \wedge \mathbf{det}(Q \,|||\, RUN_s) \Rightarrow \mathbf{det}((P \parallel Q) \,|||\, RUN_s)$$

Proof. Assume $\mathbf{det}(P \,|||\, RUN_s)$. By Proposition 3.1, $\mathbf{nondiv}(P)$ and

$$\forall t_1, t_2 : \mathrm{seq}\,\Sigma;\ e : \Sigma \bullet$$
$$t_1 \frown t_2 \in \mathit{traces}\,[\![P]\!] \wedge t_1 \frown \langle e \rangle \frown t_2 \in \mathit{traces}\,[\![P]\!] \wedge e \in s \Rightarrow$$
$$(\Sigma - s) \cap \overline{\mathit{inits}}(P \ \mathbf{after}\ t_1 \frown \langle e \rangle \frown t_2) = (\Sigma - s) \cap \overline{\mathit{inits}}(P \ \mathbf{after}\ t_1 \frown t_2)$$
$$\wedge$$
$$t_1 \frown t_2 \in \mathit{traces}\,[\![P]\!] \wedge t_1 \frown \langle e \rangle \frown t_2 \in \mathit{traces}\,[\![P]\!] \wedge e \notin s \Rightarrow$$
$$(t_1, \{e\}) \notin \mathit{failures}\,[\![P]\!]$$

Assume $\mathbf{det}(Q \mid\mid\mid RUN_s)$. By Proposition 3.1, $\mathbf{nondiv}(Q)$ and

$$\begin{array}{l}\forall\, t_1, t_2 : \mathrm{seq}\,\Sigma;\ e : \Sigma \bullet \\ \quad t_1 \frown t_2 \in \mathit{traces}[\![Q]\!] \wedge t_1 \frown \langle e \rangle \frown t_2 \in \mathit{traces}[\![Q]\!] \wedge e \in s \Rightarrow \\ \quad\quad (\Sigma - s) \cap \overline{\mathit{inits}}(Q \ \mathbf{after}\ t_1 \frown \langle e \rangle \frown t_2) = (\Sigma - s) \cap \overline{\mathit{inits}}(Q \ \mathbf{after}\ t_1 \frown t_2) \\ \quad \wedge \\ \quad t_1 \frown t_2 \in \mathit{traces}[\![Q]\!] \wedge t_1 \frown \langle e \rangle \frown t_2 \in \mathit{traces}[\![Q]\!] \wedge e \notin s \Rightarrow \\ \quad\quad (t_1, \{e\}) \notin \mathit{failures}[\![Q]\!]\end{array}$$

As $\mathbf{nondiv}(P)$ and $\mathbf{nondiv}(Q)$, $\mathbf{nondiv}(P \parallel Q)$.

Assume $t_1 \frown \langle e \rangle \in \mathit{traces}[\![P \parallel Q]\!]$. There are two cases to consider: $e \in s$ and $e \notin s$.

Case 1 Assume $e \in s$. It follows that

$$\begin{array}{l}\quad (\Sigma - s) \cap \overline{\mathit{inits}}(P \parallel Q \ \mathbf{after}\ t_1 \frown \langle e \rangle \frown t_2) \\ = (\Sigma - s) \cap \overline{\mathit{inits}}(P \ \mathbf{after}\ t_1 \frown \langle e \rangle \frown t_2 \upharpoonright \alpha P) \\ \quad \cup \\ \quad (\Sigma - s) \cap \overline{\mathit{inits}}(Q \ \mathbf{after}\ t_1 \frown \langle e \rangle \frown t_2 \upharpoonright \alpha Q)\end{array}$$

As $\mathbf{det}(P \mid\mid\mid RUN_s)$ and $\mathbf{det}(Q \mid\mid\mid RUN_s)$, by Proposition 3.1,

$$\begin{array}{l}\quad (\Sigma - s) \cap \overline{\mathit{inits}}(P \parallel Q \ \mathbf{after}\ t_1 \frown \langle e \rangle \frown t_2) \\ = (\Sigma - s) \cap \overline{\mathit{inits}}(P \ \mathbf{after}\ t_1 \frown t_2 \upharpoonright \alpha P) \\ \quad \cup \\ \quad (\Sigma - s) \cap \overline{\mathit{inits}}(Q \ \mathbf{after}\ t_1 \frown t_2 \upharpoonright \alpha Q)\end{array}$$

It follows that

$$\begin{array}{l}\quad (\Sigma - s) \cap \overline{\mathit{inits}}(P \parallel Q \ \mathbf{after}\ t_1 \frown \langle e \rangle \frown t_2) \\ = (\Sigma - s) \cap \overline{\mathit{inits}}(P \parallel Q \ \mathbf{after}\ t_1 \frown t_2)\end{array}$$

Case 2 Assume $e \notin s$. By Proposition 3.1,

$$\mathbf{det}(P \mid\mid\mid RUN_s) \Rightarrow (t_1 \upharpoonright \alpha P, \{e\}) \notin \mathit{failures}[\![P]\!]$$

and

$$\mathbf{det}(Q \mid\mid\mid RUN_s) \Rightarrow (t_1 \upharpoonright \alpha Q, \{e\}) \notin \mathit{failures}[\![Q]\!]$$

It follows that $(t_1, \{e\}) \notin \mathit{failures}[\![P \parallel Q]\!]$.

By Proposition 3.1, $\mathbf{det}((P \parallel Q) \mid\mid\mid RUN_s)$. □

Law A.14

$$\frac{s_1 \Leftarrow s_2\ [P] \qquad s_1 \Leftarrow s_2\ [Q]}{s_1 \Leftarrow s_2\ [P \parallel Q]}$$

Proof.

$s_1 \Leftarrow s_2\ [P]$	[Assumption]	(1)
$\alpha P \subseteq s_1 \cup s_2 \wedge s_1 \cap s_2 = \varnothing \wedge \mathbf{det}(P \mid\mid\mid RUN_{s_2})$	[1, Definition 2.5]	(2)
$s_1 \Leftarrow s_2\ [Q]$	[Assumption]	(3)
$\alpha Q \subseteq s_1 \cup s_2 \wedge s_1 \cap s_2 = \varnothing \wedge \mathbf{det}(Q \mid\mid\mid RUN_{s_2})$	[3, Definition 2.5]	(4)
$\alpha(P \parallel Q) \subseteq s_1 \cup s_2 \wedge s_1 \cap s_2 = \varnothing \wedge$ $\mathbf{det}(P \mid\mid\mid RUN_{s_2}) \wedge \mathbf{det}(Q \mid\mid\mid RUN_{s_2})$	[2, 4, $\wedge$ introduction]	(5)
$\alpha(P \parallel Q) \subseteq s_1 \cup s_2 \wedge$ $s_1 \cap s_2 = \varnothing \wedge \mathbf{det}((P \parallel Q) \mid\mid\mid RUN_{s_2})$	[5, Proposition A.6]	(6)
$s_1 \Leftarrow s_2\ [P \parallel Q]$	[6, Definition 2.5]	(7)

□

Law A.15

$$\frac{s_1 \Leftarrow s_2\ [P] \qquad \alpha Q \subseteq s_1 \qquad \mathbf{det}(Q)}{s_1 \Leftarrow s_2\ [P \parallel Q]}$$

Proof.

$s_1 \Leftarrow s_2\ [P]$	[Assumption]	(1)
$\alpha P \subseteq s_1 \cup s_2 \wedge s_1 \cap s_2 = \varnothing \wedge \mathbf{det}(P \mid\mid\mid RUN_{s_2})$	[1, Definition 2.5]	(2)
$\alpha Q \subseteq s_1$	[Assumption]	(3)
$\mathbf{det}(Q)$	[Assumption]	(4)
$\mathbf{det}(Q \mid\mid\mid RUN_{s_2})$	[2, 3, 4, Proposition 3.2]	(5)
$\alpha Q \subseteq s_1 \cup s_2 \wedge s_1 \cap s_2 = \varnothing \wedge \mathbf{det}(Q \mid\mid\mid RUN_{s_2})$	[2, 3, 5, $\wedge$ introduction]	(6)
$s_1 \Leftarrow s_2\ [Q]$	[6, Definition 2.5]	(7)
$s_1 \Leftarrow s_2\ [P \parallel Q]$	[1, 7, Law A.14]	(8)

□

Law A.16

$$\frac{s_1 \not\Leftarrow s_2\ [P] \quad \alpha Q \subseteq s_2 \quad \mathbf{nondiv}(Q)}{s_1 \not\Leftarrow s_2\ [P \parallel Q]}$$

Proof.

$s_1 \not\Leftarrow s_2\ [P]$	[Assumption]	(1)
$\alpha P \subseteq s_1 \cup s_2 \wedge s_1 \cap s_2 = \varnothing \wedge \mathbf{det}(P \mid\mid\mid RUN_{s_2})$	[1, Definition 2.5]	(2)
$\alpha Q \subseteq s_2$	[Assumption]	(3)
$\mathbf{nondiv}(Q)$	[Assumption]	(4)
$\mathbf{det}(Q \mid\mid\mid RUN_{s_2})$	[3, 4, Proposition 3.3]	(5)
$\alpha Q \subseteq s_1 \cup s_2 \wedge s_1 \cap s_2 = \varnothing \wedge\wedge \mathbf{det}(Q \mid\mid\mid RUN_{s_2})$	[2, 3, 5, $\wedge$ introduction]	(6)
$s_1 \not\Leftarrow s_2\ [Q]$	[6, Definition 2.5]	(7)
$s_1 \not\Leftarrow s_2\ [P \parallel Q]$	[1, 7, Law A.14]	(8)

□

Interleaving

Law A.17

$$\frac{s_1 \not\Leftarrow s_2\ [P] \quad s_1 \not\Leftarrow s_2\ [Q] \quad \mathbf{det}(P \mid\mid\mid Q)}{s_1 \not\Leftarrow s_2\ [P \mid\mid\mid Q]}$$

Proof.

$s_1 \not\Leftarrow s_2\ [P]$	[Assumption]	(1)
$s_1 \not\Leftarrow s_2\ [Q]$	[Assumption]	(2)
$\mathbf{det}(P \mid\mid\mid Q)$	[Assumption]	(3)
$P \mid\mid\mid Q = P \,[\![\, \varnothing \,]\!]\, Q$	[By definition]	(4)
$\mathbf{det}(P \,[\![\, \varnothing \,]\!]\, Q)$	[3, 4, Substitution]	(5)
$s_1 \not\Leftarrow s_2\ [P \,[\![\, \varnothing \,]\!]\, Q]$	[1, 2, 5, Law A.13]	(6)
$s_1 \not\Leftarrow s_2\ [P \mid\mid\mid Q]$	[4, 6, Substitution]	(7)

□

Sequential Composition

Proposition A.7 For any processes $P, Q \in PROC$, and any set of events $s \in \mathbb{P}\,\Sigma$, if

- $\mathbf{det}(P \;|||\; RUN_s)$,
- $\mathbf{det}(Q \;|||\; RUN_s)$,
- $\forall\, t_1 : traces[\![P]\!];\ t_2 : traces[\![Q]\!] \mid t_1 \frown t_2 \in traces[\![P]\!] \wedge t_1 \frown t_2 \in traces[\![P \fatsemi Q]\!] \bullet (\Sigma - s) \cap \overline{inits}(P \textbf{ after } t_1 \frown t_2) = (\Sigma - s) \cap \overline{inits}(Q \textbf{ after } t_2)$, and
- $\forall\, tr : \mathrm{seq}\,\Sigma;\ e : \Sigma \bullet tr \frown \langle\checkmark\rangle \in traces[\![P]\!] \wedge \langle e\rangle \in traces[\![Q]\!] \wedge e \notin s \Rightarrow (tr, \{e\}) \notin failures[\![P]\!]$,

then $\mathbf{det}(P \fatsemi Q \;|||\; RUN_s)$.

Proof. By the semantics of $\fatsemi$. □

Law A.18

$$\frac{\begin{array}{l} s_1 \Leftarrow\!\!\!\!= s_2\ [P] \\ s_1 \Leftarrow\!\!\!\!= s_2\ [Q] \\ \forall\, t_1 : traces[\![P]\!];\ t_2 : traces[\![Q]\!] \mid t_1 \frown t_2 \in traces[\![P]\!] \wedge t_1 \frown t_2 \in traces[\![P \fatsemi Q]\!] \bullet \\ (\Sigma - s_2) \cap \overline{inits}(P \textbf{ after } t_1 \frown t_2) = (\Sigma - s_2) \cap \overline{inits}(Q \textbf{ after } t_2) \\ \quad \forall\, tr : \mathrm{seq}\,\Sigma;\ e : \Sigma \bullet \\ \quad tr \frown \langle\checkmark\rangle \in traces[\![P]\!] \wedge \langle e\rangle \in traces[\![Q]\!] \wedge e \notin s_2 \Rightarrow (tr, \{e\}) \notin failures[\![P]\!] \end{array}}{s_1 \Leftarrow\!\!\!\!= s_2\ [P \fatsemi Q]}$$

Proof.

$s_1 \Leftarrow\!\!\!\!= s_2\ [P]$	[Assumption]	(1)
$\alpha P \subseteq s_1 \cup s_2 \wedge s_1 \cap s_2 = \varnothing \wedge \mathbf{det}(P \;|||\; RUN_{s_2})$	[1, Definition 2.5]	(2)
$s_1 \Leftarrow\!\!\!\!= s_2\ [Q]$	[Assumption]	(3)
$\alpha Q \subseteq s_1 \cup s_2 \wedge s_1 \cap s_2 = \varnothing \wedge \mathbf{det}(Q \;|||\; RUN_{s_2})$	[3, Definition 2.5]	(4)
$\alpha(P \fatsemi Q) \subseteq s_1 \cup s_2 \wedge s_1 \cap s_2 = \varnothing \wedge$ $\mathbf{det}(P \;|||\; RUN_{s_2}) \wedge \mathbf{det}(Q \;|||\; RUN_{s_2})$	[2, 4, $\wedge$ introduction]	(5)
$\forall\, t_1 : traces[\![P]\!];\ t_2 : traces[\![Q]\!] \mid$ $t_1 \frown t_2 \in traces[\![P]\!] \wedge t_1 \frown t_2 \in traces[\![P \fatsemi Q]\!] \bullet$ $(\Sigma - s_2) \cap \overline{inits}(P \textbf{ after } t_1 \frown t_2)$ $= (\Sigma - s_2) \cap \overline{inits}(Q \textbf{ after } t_2)$	[Assumption]	(6)
$\forall\, tr : \mathrm{seq}\,\Sigma;\ e : \Sigma \bullet$ $tr \frown \langle\checkmark\rangle \in traces[\![P]\!] \wedge \langle e\rangle \in traces[\![Q]\!] \wedge e \notin s_2 \Rightarrow$ $(tr, \{e\}) \notin failures[\![P]\!]$	[Assumption]	(7)
$\alpha(P \fatsemi Q) \subseteq s_1 \cup s_2 \wedge$ $s_1 \cap s_2 = \varnothing \wedge \mathbf{det}(P \fatsemi Q \;|||\; RUN_{s_2})$	[5, 6, 7, Proposition A.7]	(8)
$s_1 \Leftarrow\!\!\!\!= s_2\ [P \fatsemi Q]$	[8, Definition 2.5]	(9)

□

Concealment

Proposition A.8

$$\forall P : PROC;\ X, s : \mathbb{P}\,\Sigma \bullet (\mathbf{det}(P \,|||\, RUN_{X \cup s}) \wedge \mathbf{nondiv}(P \setminus X)) \Rightarrow \mathbf{det}(P \setminus X \,|||\, RUN_s)$$

Proof. See [Ros95]. □

Law A.19

$$\frac{s_1 \Leftarrow\!\!= s_2 \cup X\ [P] \qquad \mathbf{nondiv}(P \setminus X)}{s_1 \Leftarrow\!\!= s_2\ [P \setminus X]}$$

Proof.

$s_1 \Leftarrow\!\!= s_2 \cup X\ [P]$	[Assumption]	(1)
$\alpha P \subseteq s_1 \cup (s_2 \cup X) \wedge$		
$s_1 \cap (s_2 \cup X) = \varnothing \wedge \mathbf{det}(P \,|||\, RUN_{s_2 \cup X})$	[1, Definition 2.5]	(2)
$\alpha(P \setminus X) \subseteq s_1 \cup s_2 \wedge$		
$s_1 \cap s_2 = \varnothing \wedge \mathbf{det}(P \,|||\, RUN_{s_2 \cup X})$	[2, Set theory]	(3)
$\mathbf{nondiv}(P \setminus X)$	[Assumption]	(4)
$\alpha(P \setminus X) \subseteq s_1 \cup s_2 \wedge$		
$s_1 \cap s_2 = \varnothing \wedge \mathbf{det}(P \setminus X \,|||\, RUN_{s_2})$	[3, 4, Proposition A.8]	(5)
$s_1 \Leftarrow\!\!= s_2\ [P \setminus X]$	[5, Definition 2.5]	(6)

□

Event Renaming

Proposition A.9

$$\forall P : PROC;\ s : \mathbb{P}\,\Sigma;\ f : \Sigma \nrightarrow \Sigma \mid f \in \alpha P \rightarrowtail \Sigma \bullet \mathbf{det}(P \,|||\, RUN_s) \Rightarrow \mathbf{det}(f(P) \,|||\, RUN_{f(s)})$$

Proof. As f is a total injection, and by the semantics of renaming. □

Law A.20

$$\frac{\begin{array}{c} s_1 \Leftarrow\!\!\!= s_2\ [P] \\ f \in \alpha P \rightarrowtail \Sigma \end{array}}{f(s_1) \Leftarrow\!\!\!= f(s_2)\ [f(P)]}$$

Proof.

$s_1 \Leftarrow\!\!\!= s_2\ [P]$	[Assumption]	(1)
$\alpha P \subseteq s_1 \cup s_2 \wedge s_1 \cap s_2 = \varnothing \wedge \mathbf{det}(P \mid\mid\mid RUN_{s_2})$	[1, Definition 2.5]	(2)
$f \in \alpha P \rightarrowtail \Sigma$	[Assumption]	(3)
$\alpha(f(P)) \subseteq f(s_1) \cup f(s_2) \wedge$ $f(s_1) \cap f(s_2) = \varnothing \wedge \mathbf{det}(P \mid\mid\mid RUN_{s_2})$	[2, 3, injectivity of f]	(4)
$\alpha(f(P)) \subseteq f(s_1) \cup f(s_2) \wedge$ $f(s_1) \cap f(s_2) = \varnothing \wedge \mathbf{det}(f(P) \mid\mid\mid RUN_{f(s_2)})$	[3, 4, Proposition A.9]	(5)
$f(s_1) \Leftarrow\!\!\!= f(s_2)\ [f(P)]$	[5, Definition 2.5]	(6)

□

Interrupts

Proposition A.10 For any processes $P, Q \in PROC$ and any set of events $s \in \mathbb{P}\,\Sigma$, if

- $\mathbf{det}(P \mid\mid\mid RUN_s)$,
- $\mathbf{det}(Q \mid\mid\mid RUN_s)$,
- $\forall\, t_1 : traces[\![P]\!];\ t_2 : traces[\![Q]\!] \mid t_1 \frown t_2 \in traces[\![P]\!] \bullet$ $\overline{inits}(Q \textbf{ after } t_2) \cap (\Sigma - s) = \overline{inits}(P \textbf{ after } t_1 \frown t_2) \cap (\Sigma - s)$, and
- $\forall\, t_1 : traces[\![P]\!];\ e : \Sigma \mid \langle e \rangle \in traces[\![Q]\!] \bullet$ $e \notin s \Rightarrow (t_1, \{e\}) \notin failures[\![P]\!]$

then $\mathbf{det}(P \triangle Q \mid\mid\mid RUN_s)$.

Proof. By the semantics of $\triangle$. □

Law A.21

$$\frac{\begin{array}{c} s_1 \not\Leftarrow s_2\ [P] \\ s_1 \not\Leftarrow s_2\ [Q] \\ \forall\, t_1 : \mathit{traces}[\![P]\!];\ t_2 : \mathit{traces}[\![Q]\!] \mid t_1 \frown t_2 \in \mathit{traces}[\![P]\!] \bullet \\ \quad \overline{\mathit{inits}}(Q \textbf{ after } t_2) \cap (\Sigma - s_2) = \overline{\mathit{inits}}(P \textbf{ after } t_1 \frown t_2) \cap (\Sigma - s_2) \\ \forall\, t_1 : \mathit{traces}[\![P]\!];\ e : \Sigma \mid \langle e \rangle \in \mathit{traces}[\![Q]\!] \bullet \\ e \notin s_2 \Rightarrow (t_1, \{e\}) \notin \mathit{failures}[\![P]\!] \end{array}}{s_1 \not\Leftarrow s_2\ [P \triangle Q]}$$

Proof.

$s_1 \not\Leftarrow s_2\ [P]$	[Assumption]	(1)
$\alpha P \subseteq s_1 \cup s_2 \wedge s_1 \cap s_2 = \varnothing \wedge \textbf{det}(P \mid\mid\mid RUN_{s_2})$	[1, Definition 2.5]	(2)
$s_1 \not\Leftarrow s_2\ [Q]$	[Assumption]	(3)
$\alpha Q \subseteq s_1 \cup s_2 \wedge s_1 \cap s_2 = \varnothing \wedge \textbf{det}(Q \mid\mid\mid RUN_{s_2})$	[3, Definition 2.5]	(4)
$\forall\, t_1 : \mathit{traces}[\![P]\!];\ t_2 : \mathit{traces}[\![Q]\!] \mid$ $t_1 \frown t_2 \in \mathit{traces}[\![P]\!] \bullet$ $\overline{\mathit{inits}}(Q \textbf{ after } t_2) \cap (\Sigma - s_2)$ $= \overline{\mathit{inits}}(P \textbf{ after } t_1 \frown t_2) \cap (\Sigma - s_2)$	[Assumption]	(5)
$\forall\, t_1 : \mathit{traces}[\![P]\!];\ e : \Sigma \mid \langle e \rangle \in \mathit{traces}[\![Q]\!] \bullet$ $e \notin s_2 \Rightarrow (t_1, \{e\}) \notin \mathit{failures}[\![P]\!]$	[Assumption]	(6)
$\alpha(P \triangle Q) \subseteq s_1 \cup s_2 \wedge$ $s_1 \cap s_2 = \varnothing \wedge \textbf{det}(P \triangle Q \mid\mid\mid RUN_s)$	[2, 4, 5, 6, Proposition A.10]	(7)
$s_1 \not\Leftarrow s_2\ [P \triangle Q]$	[7, Definition 2.5]	(8)

□

Geographic Data for the Open ALVEY

The geographic data presented in this appendix is associated with the Open ALVEY interlocking of Figure 7.1, and was kindly supplied by British Rail Research (Derby).

B.1 Route Setting Data

Q10A **if** P201 cfr UAB-AC f UBA-AB f
then R10A s P201 cr UAB-CA l UBA-BA l

Q10B **if** P201 cfn UAB-BC f UAC-AB f
then R10B s P201 cn UAB-CB l UAC-BA l

Q11A **if** P202 cfn UAD-BA f UAC-BA f
then R11A s P202 cn UAD-AB l UAC-AB l

Q11B **if** P202 cfr UAD-CA f UBA-BA f
then R11B s P202 cr UAD-AC l UBA-AB l

Q12 **if** P202 cfr UAE-AB f UAD-AC f
then R12 s P202 cr UAD-CA l UAE-BA l

Q13 **if** P201 cfr UAA-BA f UAB-CA f
then R13 s P201 cr UAB-AC l UAA-AB l UAK-AB l

Q14 **if** P202 cfn UAE-AB f UAD-AB f
then R14 s P202 cn UAD-BA l UAE-BA l

Q15 **if** P201 cfn UAA-BA f UAB-CB f
then R15 s P201 cn UAB-BC l UAA-AB l UAK-AB l

Q20A **if** P203 cfr UAG-BA f UCA-BA f
then R20A s P203 cr UAG-AB l UCA-AB l

Q20B **if** P203 cfn UAG-CA f UAH-BA f
then R20B s P203 cn UAG-AC l UAH-AB l

Q21A **if** P204 cfn UAJ-AC f UAH-AB f
then R21A s P204 cn UAJ-CA l UAH-BA l

Q21B **if** P204 cfr UAJ-BC f UCA-AB f
then R21B s P204 cr UAJ-CB l UCA-BA l

Q22 **if** P204 cfr UAJ-CB f UAK-AB f
then R22 s P204 cr UAJ-BC l UAK-BA l UAA-BA l

Q23 **if** P203 cfn UAG-AC f UAF-AB f
then R23 s P203 cn UAG-CA l UAF-BA l

Q24 **if** P204 cfn UAJ-CA f UAK-AB f
then R24 s P204 cn UAJ-AC l UAK-BA l UAA-BA l

Q25 **if** P203 cfr UAG-AB f UAF-AB f
then R25 s P203 cr UAG-BA l UAF-BA l

B.2 Sub-route Release Data

UAA-AB f **if** TAA c UAB-AC f UAB-BC f

UAA-BA f **if** TAA c UAK-BA f

UAB-AC f **if** TAB c R13 xs

UAB-BC f **if** TAB c R15 xs

UAB-CA f **if** TAB c R10A xs

UAB-CB f **if** TAB c R10B xs

UAC-AB f **if** TAC c UAD-AB f

UAC-BA f **if** TAC c UAB-CB f

UAD-AB f **if** TAD c R11A xs

UAD-AC f **if** TAD c R11B xs

UAD-BA f **if** TAD c R14 xs

UAD-CA f **if** TAD c R12 xs

UAE-BA f **if** TAE c UAD-BA f UAD-CA f

UAF-BA f **if** TAF c UAG-BA f UAG-CA f

UAG-AB f **if** TAG c R20A xs

UAG-AC f **if** TAG c R20B xs

UAG-BA f **if** TAG c R25 xs

UAG-CA f **if** TAG c R23 xs

UAH-AB f **if** TAH c UAG-AC f

UAH-BA f **if** TAH c UAJ-CA f

UAJ-AC f **if** TAJ c R24 xs

UAJ-BC f **if** TAJ c R22 xs

UAJ-CA f **if** TAJ c R21A xs

UAJ-CB f **if** TAJ c R21B xs

UAK-AB f **if** TAK c UAA-AB f

UAK-BA f **if** TAK c UAJ-AC f UAJ-BC f

UBA-AB f **if** TBA c UAD-AC f

UBA-BA f **if** TBA c UAB-CA f

UCA-AB f **if** TCA c UAG-AB f

UCA-BA f **if** TCA c UAJ-CB f

B.3 Points Free to Move Data

P201N TAB c UAB-AC f UAB-CA f
P201R TAB c UAB-BC f UAB-CB f

P202N TAD c UAD-AC f UAD-CA f
P202R TAD c UAD-AB f UAD-BA f

P203N TAG c UAG-AB f UAG-BA f
P203R TAG c UAG-AC f UAG-CA f

P204N TAJ c UAJ-BC f UAJ-CB f
P204R TAJ c UAJ-AC f UAJ-CA f

Appendix C

Geographic Data for the Closed ALVEY

The geographic data presented in this appendix is associated with the Closed ALVEY interlocking of Figure 7.4, and was kindly supplied by British Rail Research (Derby).

C.1 Route Setting Data

Q10A **if** P201 cfr UAB-AC f UBA-AB f
then R10A s P201 cr UAB-CA l UBA-BA l

Q10B **if** P201 cfn UAB-BC f UAC-AB f
then R10B s P201 cn UAB-CB l UAC-BA l

Q11A **if** P202 cfn UAD-BA f UAC-BA f
then R11A s P202 cn UAD-AB l UAC-AB l

Q11B **if** P202 cfr UAD-CA f UBA-BA f
then R11B s P202 cr UAD-AC l UBA-AB l

Q12 **if** P202 cfr UAE-AB f UAD-AC f
then R12 s P202 cr UAD-CA l UAE-BA l UAF-AB l

Q13 **if** P201 cfr UAA-BA f UAB-CA f
then R13 s P201 cr UAB-AC l UAA-AB l UAK-AB l

Q14 **if** P202 cfn UAE-AB f UAD-AB f
then R14 s P202 cn UAD-BA l UAE-BA l UAF-AB l

Q15 **if** P201 cfn UAA-BA f UAB-CB f
then R15 s P201 cn UAB-BC l UAA-AB l UAK-AB l

Q20A **if** P203 cfr UAG-BA f UCA-BA f
then R20A s P203 cr UAG-AB l UCA-AB l

Q20B **if** P203 cfn UAG-CA f UAH-BA f
then R20B s P203 cn UAG-AC l UAH-AB l

Q21A **if** P204 cfn UAJ-AC f UAH-AB f
then R21A s P204 cn UAJ-CA l UAH-BA l

Q21B **if** P204 cfr UAJ-BC f UCA-AB f
then R21B s P204 cr UAJ-CB l UCA-BA l

Q22 **if** P204 cfr UAJ-CB f UAK-AB f
then R22 s P204 cr UAJ-BC l UAK-BA l UAA-BA l

Q23 **if** P203 cfn UAG-AC f UAF-AB f
then R23 s P203 cn UAG-CA l UAF-BA l UAE-AB l

Q24 **if** P204 cfn UAJ-CA f UAK-AB f
then R24 s P204 cn UAJ-AC l UAK-BA l UAA-BA l

Q25 **if** P203 cfr UAG-AB f UAF-AB f
then R25 s P203 cr UAG-BA l UAF-BA l UAE-AB l

C.2 Sub-route Release Data

UAA-AB f **if** TAA c UAB-AC f UAB-BC f

UAA-BA f **if** TAA c UAK-BA f

UAB-AC f **if** TAB c R13 xs

UAB-BC f **if** TAB c R15 xs

UAB-CA f **if** TAB c R10A xs

UAB-CB f **if** TAB c R10B xs

UAC-AB f **if** TAC c UAD-AB f

UAC-BA f **if** TAC c UAB-CB f

UAD-AB f **if** TAD c R11A xs

UAD-AC f **if** TAD c R11B xs

UAD-BA f **if** TAD c R14 xs

UAD-CA f **if** TAD c R12 xs

UAE-AB f **if** TAE c UAF-BA f

UAE-BA f **if** TAE c UAD-BA f UAD-CA f

UAF-AB f **if** TAF c UAE-BA f

UAF-BA f **if** TAF c UAG-BA f UAG-CA f

UAG-AB f **if** TAG c R20A xs

UAG-AC f **if** TAG c R20B xs

UAG-BA f **if** TAG c R25 xs

UAG-CA f **if** TAG c R23 xs

UAH-AB f **if** TAH c UAG-AC f

UAH-BA f **if** TAH c UAJ-CA f

UAJ-AC f **if** TAJ c R24 xs

UAJ-BC f **if** TAJ c R22 xs

UAJ-CA f **if** TAJ c R21A xs

UAJ-CB f **if** TAJ c R21B xs

UAK-AB f **if** TAK c UAA-AB f

UAK-BA f **if** TAK c UAJ-AC f UAJ-BC f

UBA-AB f **if** TBA c UAD-AC f

UBA-BA f **if** TBA c UAB-CA f

UCA-AB f **if** TCA c UAG-AB f

UCA-BA f **if** TCA c UAJ-CB f

C.3 Points Free to Move Data

P201N TAB c UAB-AC f UAB-CA f
P201R TAB c UAB-BC f UAB-CB f

P202N TAD c UAD-AC f UAD-CA f
P202R TAD c UAD-AB f UAD-BA f

P203N TAG c UAG-AB f UAG-BA f
P203R TAG c UAG-AC f UAG-CA f

P204N TAJ c UAJ-BC f UAJ-CB f
P204R TAJ c UAJ-AC f UAJ-CA f

Appendix D

The Complexity of Analysing SSI Safety Invariants

In this appendix, we present a number of tables which illustrate the complexity of analysing SSI safety invariants with FDR.

Track circuit	States
TAA	14592
TAB	3532
TAC	232
TAD	3312
TAG	3312
TAH	232
TAJ	3532
TAK	14592
TBA	232
TCA	232

Table D.1: The complexity of verifying invariant 1 for the Open ALVEY.

Track circuit	States
TAA	14592
TAB	3352
TAC	232
TAD	3352
TAE	14592
TAF	14592
TAG	3352
TAH	232
TAJ	3352
TAK	14592
TBA	232
TCA	232

Table D.2: The complexity of verifying invariant 1 for the Closed ALVEY.

Sub-routes	States
(UAD-AB, UAD-AC)	128
(UAD-AB, UAD-BA)	16
(UAD-AB, UAD-CA)	128
(UAD-AC, UAD-BA)	128
(UAD-AC, UAD-CA)	16
(UAD-BA, UAD-CA)	64

Table D.3: The complexity of verifying invariant 1 for track circuit TAD.

Route 1	Route 2	States
R13	R22	720
R13	R24	720
R15	R22	720
R15	R24	720

Table D.4: The complexity of verifying mutual exclusion for the Open ALVEY.

Sub-route	States
UAB-AC	1804
UAB-BC	1592
UAB-CA	1616
UAB-CB	1432

Table D.5: The complexity of verifying invariant 2 for point P201.

Sub-route	States
UAD-AB	1432
UAD-AC	1616
UAD-BA	1432
UAD-CA	1616

Table D.6: The complexity of verifying invariant 2 for point P202.

Route	States
R10A	108
R10B	108
R11A	108
R11B	108
R12	236
R13	332
R14	236
R15	332
R20A	108
R20B	108
R21A	108
R21B	108
R22	332
R23	236
R24	332
R25	236

Table D.7: The complexity of verifying invariant 3 for the Open ALVEY.

Route	States
R10A	108
R10B	108
R11A	108
R11B	108
R12	332
R13	332
R14	332
R15	332
R20A	108
R20B	108
R21A	108
R21B	108
R22	332
R23	332
R24	332
R25	332

Table D.8: The complexity of verifying invariant 3 for the Closed ALVEY.

Route	States
R11A	2672
R11B	2532
R12	3324
R14	3464

Table D.9: The complexity of verifying invariant 4 for point P202.

Route	Sub-routes	States
R10A	(UAB-CA, UBA-BA)	124
R10B	(UAB-CB, UAC-BA)	124
R11A	(UAD-AB, UAC-AB)	124
R11B	(UAD-AC, UBA-AB)	124
R12	(UAD-CA, UAE-BA)	252
R13	(UAB-AC, UAA-AB)	288
R13	(UAA-AB, UAK-AB)	96
R14	(UAD-BA, UAE-BA)	252
R15	(UAB-BC, UAA-AB)	288
R15	(UAA-AB, UAK-AB)	96
R20A	(UAG-AB, UCA-AB)	124
R20B	(UAG-AC, UAH-AB)	124
R21A	(UAJ-CA, UAH-BA)	124
R21B	(UAJ-CB, UCA-BA)	124
R22	(UAJ-BC, UAK-BA)	288
R22	(UAK-BA, UAA-BA)	96
R23	(UAG-CA, UAF-BA)	252
R24	(UAJ-AC, UAK-BA)	288
R24	(UAK-BA, UAA-BA)	96
R25	(UAG-BA, UAF-BA)	252

Table D.10: The complexity of verifying invariant 5 for the Open ALVEY.

Appendix E

A Glossary of Terms

Absolute safety A guarantee of the impossibility of (known) hazards occurring in a given system.

Availability "Measure of proper service delivery with respect to the alternation proper-improper service." [Lap89]

Benign events Observable events of a system which are non-critical.

Benign failure "Where the consequences [of failure] are of the same order of magnitude (generally in terms of cost) as the benefit provided by proper service delivery." [Lap89]

Catastrophic failure "Where the consequences [of failure] are incommensurable with the benefit provided by proper service delivery." [Lap89]

Causitive failures Single-event failures which may occur in the context of a functional specification.

Curative maintenance "Removing faults which have produced (an) error(s) and have been reported." [Lap89]

Dependability "Property of a computing system which allows reliance to be justifiably placed on the service it delivers." [Lap89]

Fail-initial A failure mode which, in the event of a failure, returns a system to its initial state.

Fail-operational A failure mode which, in the event of a failure, ensures that a system's critical and non-critical behaviour are adversely affected neither by that failure, nor by subsequent measures taken to recover from it.

Fail-safe A failure mode which, in the event of a failure, ensures that a system's critical behaviour is not adversely affected by that failure.

Fail-soft A failure mode which, in the event of a failure, ensures that a system's critical behaviour is adversely affected neither by that failure, nor by subsequent measures taken to recover from it.

Fail-stop A failure mode which, in the event of a failure, ensures that the system's behaviour terminates, and that the system is left in a safe state.

Failure "When the delivered service deviates from conditions stated in the service specification." [Lap89]

Failure mode A system's required behaviour in the event of a failure.

Fault

1. "Adjudged or hypothesised cause of an error." [Lap89]
2. "Error cause which is intended to be avoided or tolerated." [Lap89]
3. "Consequence for a system of the failure of another system which has interacted or is interacting with the considered system." [Lap89]

Fault-tolerance "How to provide, by redundancy, a service complying with a specification in spite of faults." [Lap89]

Firewalling A method of separating system components, such that a failure in one component cannot have an adverse affect upon the safety-critical behaviour of others.

Hazard analysis The investigation of causes, and possible prevention of, a system's hazards.

Hazards "[A] set of conditions (i.e., a state) that can lead to an accident given certain environmental conditions." [Lev91]

Human factors The analysis of human behaviour with respect to software-based systems.

Information flow The ability of one system component to exert an influence over the behaviour of another.

Initiating event The first in a chain of events leading to a hazard.

Malicious behaviour Human behaviour which has the specific aim of decreasing the level of safety of a system.

Neutral behaviour Human behaviour which can have no influence upon the level of safety of a system.

Non-interference A system's absence of information flow with respect to a statement of security or safety.

Observational failures Failures which occur following the occurrence of several events *in a specific order.*

Preventative measures Events designed to eliminate the possibility of faults.

Protection A guarantee of non-interference between specific sets of events in a CSP process.

Random failures Failures which occur as consequences of hardware degradation.

Recovery events Observable events of a system which can be classed as recovery measures.

Recovery measures Events designed to limit the effects of failures.

Reliability

1. "Dependability with respect to the continuity of service." [Lap89]
2. "Measure of continuous proper service delivery." [Lap89]
3. "Measure of the time to failure." [Lap89]

Risk A function of the probability of a hazard occurring and the perceived loss caused by that hazard [Lev86].

Risk analysis The investigation and calculation of the cost of potential hazards.

Safety core A method of separating components such that safety-critical components are non-interfering with non-safety-critical components.

Safety-critical events Those events of a safety-critical system which, if performed incorrectly or not at all, might lead to a failure.

Safety-critical transaction non-interference The application of the property of transaction non-interference to the analysis of safety-critical properties.

Systematic failures Failures which occur as consequences of design faults.

Transaction non-interference The property that a user's view of a system is identical both before and after the occurrence of another user's transaction.

Trusted behaviour Human behaviour which has the specific aim of maintaining or increasing the level of safety of a system.