A TIMED SEMANTICS FOR A HIERARCHICAL DESIGN NOTATION

Phillip James Brooke

A thesis submitted in partial fulfilment of the requirements
for the Degree of Doctor of Philosophy

University of York
Department of Computer Science

April 1999

Abstract

The software control systems that are embedded in many products are increasingly complex. There are many stages in the typical software design life cycle, and these include testing, and sometimes use formal methods.

This thesis aims to strengthen the design life cycle by demonstrating a pragmatic use of formal methods for an industrially-applicable design method.

We take a design method and notation called DORIS (which is used by British Aerospace), and give a formal syntax for the notation. This then forms the structure into which we place activities (the active processing parts of the system), and intercommunication data areas (IDAs, the 'passive' parts of the system through which activities communicate).

We aim to give an industrially useful semantics for this notation, so we use several models of Timed CSP as the underlying semantic domain. This allows us to represent the many timed and liveness requirements in BAe's work using the well-understood theory for Timed CSP.

For a given system design, we can generate a Timed CSP representation of the system from its DORIS design. This representation is an abstraction of the behaviour of the system modelling the interactions between components in that system. This Timed CSP model can be analyzed and assertions tested against it. However, because of the size of these systems, tool support is necessary.

The final part of this thesis concerns the construction of such tool-supported proofs, using both a bespoke tool for generating Timed CSP from the DORIS design, and two industrial-strength tools, PVS and FDR. We illustrate this with two case studies, one of them a significant model based on a real system.

This work demonstrates that even when using current state-of-the-art tools, it is difficult to apply these methods in practice. We conclude that without significant advances in tool technology, it will continue to be difficult to prove non-trivial properties about large systems at the level of abstraction presented in this thesis.

Contents

List of Figures

Acknowledgements

I would like to thank many people, in particular:

Professor John McDermid, Stephen Paynter (my industrial supervisor), Jim Armstrong, Hugo Simpson, Richard Paige, and Dave Stringer-Calvert, who have offered me much advice and critical feedback on my work.

The examiners, Steve Schneider and Alan Wood.

My supervisor, Jeremy Jacob.

Darren Buttle, Simon Fowler, and Darren Priddin, for being happy to talk about anything not to do with this work.

EPSRC and British Aerospace for funding this work under the ROPA and CASE schemes.

Author's Declaration

This thesis is the result of my own original work (except where explicitly noted), and has not been submitted for any other qualification at any other university.

An earlier version of the work on the pool IDA contained in Chapters 4 and 7 has been presented at the Northern Formal Methods Workshop, and subsequently published in the proceedings of that workshop [9].

Part I

Introduction and Survey

Chapter 1

Introduction

1.1 Motivation

1.1.1 Problem Domain

British Aerospace (BAe) embed large, complex, concurrent computer systems in some of their products. Many of these systems are *safety critical*: a failure could lead to loss of life or other undesirable consequences. The particular computer systems that BAe are immediately concerned with are the *embedded* applications which control aircraft, both piloted and pilotless.

These applications interact with the real world in a complex way, and so they are often very complex systems themselves.

1.1.2 General Approach

When building computer systems, one general problem is to minimise the likelihood of a failure of these systems.

One approach to minimise the failure rate is 'testing'. Testing is an important part of the software engineering process, but cannot find all errors, since it would take an infeasible amount of effort to test every set of inputs to the system.[1] These systems, moreover, are often *reactive*, *i.e.* they respond to events from the environment in which they are embedded. The generation of responses to these events is often required to be within a

[1]Here, we take 'a set of inputs' to mean the entire history of a particular execution. This means that every possible input at each possible state of the system should be tested.

deadline. This makes the testing problem worse; a complex test harness needs to be constructed for these systems.

The problem of building complex computer systems is recognised, and careful design can help reduce the number of faults. As Drew Hamilton has stated, 'no one intentionally writes bad code' [37]. Despite this, faults do occur in practice, so many (defence) contractors are required to conform to a specific software engineering standard (*e.g.* Defence Standard 00-55 [17]). These standards often involve formal methods (techniques and tools based on mathematical modeling and formal logic), in addition to testing.

1.1.3 MASCOT and DORIS

We now turn our attention to the methods for designing such systems. There are a large number of notations and methods available. In this thesis, we concentrate on those methods used by BAe,[2] which include the Modular Approach to Software Construction, Operation, and Test (MASCOT) [67, 107], the Data Oriented Requirements Implementation Scheme (DORIS) [110, 112], and Statecharts [39]. All of these notations are 'hierarchical'; *i.e.* some constructs can be decomposed hierarchically: they can be refined to other constructs which may themselves be decomposed.

The design notation within DORIS (which is very similar to MASCOT-3's design notation) deals with the architectural structure of the system, and consists of activities (processes) communicating with each other and their environment via 'inter-communication data areas' (IDAs, *a.k.a.* routes or protocols). The MASCOT notation has been studied (Paynter produced an abstract formalisation of MASCOT in his thesis [84]), although not in the timed domain.

1.1.4 Reasoning about Designs

The notations mentioned above (MASCOT and Statecharts) use diagrams. When such a diagram is drawn to denote a system, we have some idea of what it means: we understand the semantics of the system, even if only informally. Importantly, the writer may have intended a different meaning

[2]British Aerospace and EPSRC funded the work in this thesis: the overall problem that this thesis addresses is broadly constrained by these interests.

to the meaning that a particular reader infers. If we want to reason more precisely about the system, we need to use mathematical reasoning. To do this, we need to define the semantics of the notation more formally.

It is not trivial to define such a semantics. For instance, over 20 attempts to give a semantics to Harel's Statecharts have been published (von der Beeck has surveyed a number of these [6]), and none of these are entirely satisfactory. Other graphical notations also have problems of ambiguity, illustrated by both Leveson [64] and Baresi [3]. As discussed in an earlier literature survey [10], there are persistent problems.

Many applications must satisfy timing requirements for their functional output to be useful. For example, a fly-by-wire flight control system that generates the correct commands to the control surfaces of an aircraft ten minutes after the pilot's input will result in the loss of the aircraft. The timing requirements, *e.g.* the commands will be generated within (say) 10 milliseconds of input, are an important part of the specification.

This explicit introduction of time presents much of the complication in producing useful models of systems. State machines can be constructed to model systems, but modelling time introduces a very large (or even infinite) state space.

A separate problem concerns the interaction of concurrent processes. Such processes often communicate through shared variables. This interaction complicates analysis of the functional and timing properties of a given system. This is the motivation for the DORIS IDAs 'decoupling' of concurrent processes (*i.e.* separating the description of how and when data is transferred between the processes from the description of what data is transferred).

There is other work on the behaviour of concurrent variables. For instance, Lamport has described three types of shared variable, safe, regular and atomic, which can be considered as particular types of IDAs [59]. In this thesis, we use Lamport's variables as the primitive components out of which we implement the DORIS IDAs.

1.1.5 Application of Formal Methods

As well as defining the semantics of a system, we need to be able to apply these semantics productively. Formal methods addresses this by using specific tools and techniques to determine properties of the system from the semantics.

There is an increasingly wide range of tools. Some are theorem provers (they assist a human user in performing a traditional proof); others are state space exploration tools (they attempt to search the entire state space of the system). Others simulate, or animate, systems — this is especially useful for design prototyping, since a system can only be formally checked against a formal set of requirements.

Finally, we note that formal methods are still not widely accepted in industry [48]. We suggest the following analogy: that the formal notations currently used (say, in Timed CSP [20]), are still at the level of machine code, whereas system designers need more abstract notations, at the level of high-level programming languages.

1.2 Contribution of this Thesis

The aim of the work described in this thesis is to produce a semantics for a hierarchical design notation in such a manner that it is acceptable to a wide range of software engineers, *e.g.* those working on systems like the large case study in this thesis (Section 8.5, Page 185). This must allow the construction of larger components that reflect the application domains where the notation is used, thus enabling functional decomposition of systems. Finally, the semantics should also be effectively machine supported.

1.3 Development of Ideas

This thesis describes the design notation within the DORIS method — it is a hierarchical design notation used in industry.

We give this notation a BNF-style grammar, and outline a Timed CSP-based semantics for the syntax that allows systems to be constructed hierarchically. Each component of a system either includes other components (using a Timed CSP parallel composition operator) , or is a primitive component that can be represented by carefully constructed Timed CSP expressions.

The two types of primitive components in DORIS are the IDAs (shared variables) and activities.

The IDAs are defined by specifying their behaviour. This is not trivial, as the IDAs can have writes and reads occurring concurrently. We investigate the IDAs further by exploring how they can be implemented in terms of Lamport's shared variables, and how compositions of IDAs behave.

Activities are abstractions of the 'processing' parts of a system. In the semantic model, they are Timed CSP expressions that interact with the activity's environment, and represent the passing of time spent calculating results. To aid the modelling of real systems, we define a simple language containing loops, termination, choice, input, output, and 'calculation', which translates into Timed CSP.

At this point, we can take a DORIS design, and translate it into Timed CSP. This is detailed work, and is aided by a tool specifically written for the purpose, named dt. This tool generates output intended to be used with a PVS [15, 79, 80, 96, 105] embedding of Timed CSP semantic models, or for input to FDR [28]. Using these tools, we can attempt to show, by machine proof, or state space exploration, that particular properties hold. These two tools are used as complementary approaches to applying our semantics for DORIS.

1.4 Overview

The thesis is in four parts. The first part contains this introduction, and a brief background survey. More details of relevant literature are given in each chapter as needed.

The second part presents the theoretical semantics for DORIS: Chapter 3 presents an overview of DORIS, and provides the overall framework for the following two chapters, which concern the definition of IDAs (Chapter 4) and activities (Chapter 5).

We then turn our attention to the mechanical support of this work in the third part. This starts with Chapter 6, where our method is described. The mechanical methods devised are then used in Chapter 7 to prove properties about IDAs.

Chapter 8 sets out two case studies. The first is based on a small example by Paynter *et al.* [83]. The second concerns a large design from British Aerospace. Subsequently, we report the analysis of these case studies.

The final part of the thesis consists of Chapters 9 and 10, which summarise this work, and discuss future possibilities.

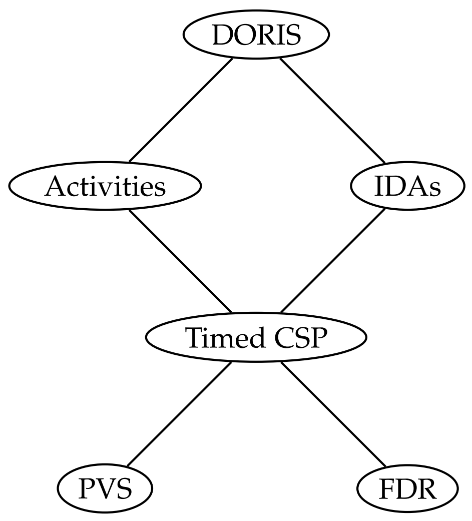


Figure 1.1: Structure of the theory

Figure 1.1 illustrates the structure of the theory in this thesis: DORIS consists of two major parts, the IDAs and activities, both of which are given a semantics in terms of Timed CSP. Finally, mechanical support is provided by the PVS and FDR tools.

A glossary of terms, definitions and abbreviations, a glossary of symbols, and an annotated bibliography conclude the thesis.

Chapter 2

Background Survey

In this chapter, we provide a general introduction and overview to the domain of critical systems engineering and formal methods (FM). We also introduce MASCOT and DORIS, and a process algebra, Timed CSP.

Throughout the rest of the thesis, further material is introduced where it is needed. An annotated bibliography is also provided (Page 255).

2.1 Critical Systems

The systems that BAe typically use DORIS to design are *safety* or *mission critical*. The failure of a safety critical system could result in the loss of life. A mission critical system failing will probably not result in loss of life, but could cause severe difficulties. For example, a weapon system failing to hit a target is mission critical, but the flight control system of a manned aircraft is safety critical.

A system which could potentially damage the wider environment can be considered safety critical: oil distribution systems could pollute a large area quickly; they are dependent on computer control of pumps and valves. Nuclear power plants are heavily dependent on their control systems to ensure safe operation.

Other systems can be described as critical systems: a company may depend on its computers to ensure that orders go to the right place on time: they are *business critical*. If such a system fails, it could cause the company to go out of business. A *security critical* system may allow information into the possession of unauthorised people, which could cause severe repercussions.

We can see that the concept of a critical system can encompass many domains and types of failure. However, they all have one common theme: they must operate reliably and robustly.

In many circumstances, it is impractical to test a critical system in its 'final' environment, *e.g.* an ambulance control system. Then how do we tell whether the system will work?

There are already many software engineering methods aimed at designing systems, *e.g.* SSADM. What is lacking in many of these methods is a way to reliably determine that the final system does exactly what it is meant to. Once designed and built, a system can be tested; however, testing cannot cover all possible circumstances (except for trivial systems).

More can be done: formal methods are still relatively under-used in industry, although defence contractors are often required to conform to safety standards that mandate formal methods.

2.2 Formal Methods

A NASA guidebook states that formal methods consist

> "...of a set of techniques and tools based on mathematical modeling and formal logic that are used to specify and verify requirements and designs for computer systems and software." [75]

There are several important points here:

- formal methods are based on mathematical and logical techniques;
- formal methods consist of techniques and tools (otherwise they are of no use);
- computer systems as well as software are (sometimes) amenable to such analysis; and
- formal analysis of a computer system (or software) is based on a *model* of that computer system.

It is the last point that is particularly important: if a formal model of a washing machine concludes that the washing machine will always complete a programme that has been started, then is the analysis invalidated if the user disconnects the power supply during the programme? In this case, there would be an explicit or (more likely) an implicit assumption that the power supply always remains connected.

The issue here is that we must understand clearly what assumptions are made during the construction of a model: the analysis is not about the 'real' system. Barwise discusses this subject in some depth in a refutation of Fetzer's argument [29]. In his conclusions, Barwise says

> "[Articles critical about computer system correctness] will help to generate an improved understanding among professionals and the public of what such proofs show and what they do not show about the correctness of physical computers operating in the real world." [4]

Jackson makes some similar points in his short articles [52].

Much of the rationale for formal methods in software engineering relies on the distinction between traditional engineering disciplines and the problems with writing software. These problems are twofold:

1. Software is discrete and discontinuous. A small change to source code or system input can have a large effect on the behaviour of the overall system. Most engineering systems (*e.g.* bridges) have a range of tolerances that can be exploited to give a margin of safety.

2. The sheer complexity of software and the bespoke approach to the construction of software allows design flaws to be easily missed. The interaction of even a moderate number of complex components vastly increases the overall complexity of the system.

The reader is recommended to read Rushby's introduction where these issues are more fully discussed [97].

Another useful reference is the two-volume NASA guidebook [75, 76]. This is a wide-ranging, thorough, and quite readable guidebook. It serves well as an overall reference into formal methods techniques.

Although formal methods are not widely used, there is no reason why they should not be used in the development of all software, except for the (perceived) cost of using FM for a system that simply does not warrant the level of assurance required.[1]

2.3 MASCOT and DORIS

Since DORIS draws much of its notation and methodology from MASCOT, it is appropriate to first introduce MASCOT.

2.3.1 MASCOT

The Modular Approach to Software Construction, Operation, and Test (MASCOT) incorporates:

- a means of design representation,
- a method for deriving the design,
- a way of constructing software so that it is consistent with the design,
- a means of executing the constructed software so that the design structure remains visible at run-time, and
- facilities for testing the software in terms of the design structure [67].

MASCOT originated during the early 1970s, with an 'Official Definition of MASCOT' (*a.k.a.* MASCOT-1) being produced in 1978. This came about through work on embedded real-time systems. MASCOT-2 followed in the early 1980s, with the current standard (MASCOT-3) issued in 1987.

One of the motivations for MASCOT is the intangibility of (large) software systems. This causes difficulty in understanding how a particular system works. To address this, MASCOT, in common with many structured methods, approaches the problem top-down; *i.e.* the overall structure of the system is set out, with further refinements as more detail is required. The lowest level of design (in this context) is the source code in the target language.

MASCOT is not just a notation; it is a method. The MASCOT process has three distinct phases:

[1]Of course, this does not prevent people using such software for critical purposes. For example, consider Wiener's example of a surgeon who used a spreadsheet to analyze data from a patient during open-heart surgery [119, Page 128].

Network Design An iterative process which creates a hierarchy of subsystems, IDAs and servers (all described below). This phase identifies the purpose of each component.

Component Design Each of these components is (if necessary) refined into more components (activities). The bottom-level activities are then designed and coded.

Integration and Test Each component is tested individually, then with its neighbours, and so on, until the whole system has been constructed.

There are a number of component types referred to above:

System This is the 'product' which the method aims to produce.

Subsystem The system is broken down into functionally separate components called *subsystems*.

Server The system interacts with its environment through sensors and actuators via servers (a specialised subsystem).

Activities These are the parts of the system which perform the bulk of the 'work', *e.g.* calculations.

IDAs Intercommunication Data Areas (IDAs) link the activities and servers together by allowing data to flow around the system in a controlled manner.

An example MASCOT-3 system is given in Figure 2.1, taken from Page 2-3 of the MASCOT-3 manual [67]. This example contains a system called 'example_sys', which contains three subsystems (with round cornered boxes) and two IDAs (the square cornered boxes inside example_sys). There are three devices through which the system communicates with its environment.

An important feature to note is the distinction between *ports* and *windows*. The ports are represented by the small disks on the edge of an activity at the end of a path (*e.g.* ap2 in subsys_2). A window is represented by a small filled-in rectangle (*e.g.* aw in sida_2). The action associated with a window is considered to be passive, and is driven by the active end, indicated by the port. In MASCOT and DORIS, paths may only connect a port

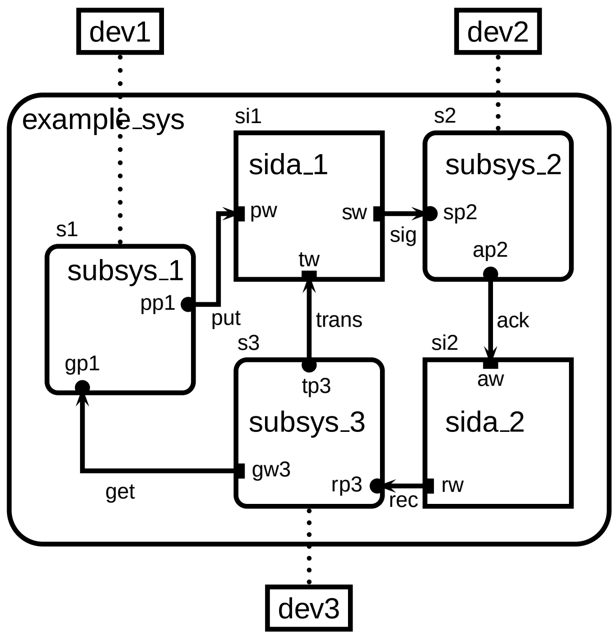


Figure 2.1: An example MASCOT-3 system

to a window or *vice versa*. Similarly, each port and window has a direction of data flow associated with it, which is indicated by the arrow heads.[2]

2.3.2 DORIS

The primary reference for the Data Orientated Requirements Implementation Scheme (DORIS) is Simpson's *Methodological and Notational Conventions in DORIS Real Time Networks* [110].

DORIS is a wide-ranging method primarily intended for applications using real-time networks, and is itself a British Aerospace product. It includes concepts drawn from MASCOT-2 and MASCOT-3, (DORIS is very closely related to MASCOT-3) as well as the Controlled Requirements Expression (CORE) method [16]. DORIS covers a wide span of the computer system and software development process.

This chapter does not describe DORIS any further. Instead, Chapter 3 describes the DORIS notation itself (Page 53).

The IDAs (which can be treated as shared variables) are discussed in Chapter 4. Two broad classes of shared variables are discussed, in Section 4.2 (Page 81): Simpson's (DORIS) IDAs [112] and Lamport's safe, regular and atomic variables [59].

Part of the DORIS notation includes MASCOT-3-like activities. We introduce a simple language to describe these activities in Chapter 5. Section 5.1 (Page 111) describes two alternative approaches: the Activity Description Language (ADL) [83] and Statecharts [40].

2.4 Communicating Sequential Processes

This thesis uses Hoare's Communicating Sequential Processes (CSP) [46] as the semantic domain for the DORIS notation (see Chapter 3, Page 53 for the rationale). CSP is a mathematical approach to the study of concurrency and communication. It has been extended to incorporate real-time in Timed CSP (TCSP) by Reed and Roscoe [92], and further developed by Schneider [100], and Davies [23].

[2]Strictly, path fragments from a port to another port (or one window to another) are allowed. This occurs when hierarchical constructs are involved. Similarly, bidirectional paths are permitted: in this thesis, they are represented as two unidirectional paths.

Davies and Schneider's paper *A Brief History of Timed CSP* [20] describes the untimed and timed models. (Other summaries have also been produced [18].) Roscoe has recently published a text covering untimed CSP [95], and Schneider's forthcoming book covers Timed CSP [102].

2.4.1 Model of Computation

A *program* is a term in the syntax of CSP, and a *process* is an element of a semantic model. There are a number of different models, corresponding to different types of *observations*.

An observation is a record of the entire history of a particular execution of a process. In one of the simplest CSP models,[3] a CSP process interacts with its environment via synchronisations called *events*. They are considered to occur at a specific point in time, and have no duration. They occur only with the agreement of both the process and the environment that it is placed in.

The sequence of events that occur is known as a trace, and traces are the observations in the untimed traces semantic model. (We will use several semantic models of CSP in this thesis.)

There are several assumptions underlying this model of computation:

Maximal progress A program will execute as many internal (hidden) events as possible.

Maximal parallelism We assume that there are no restrictions such as scheduling limits. (Later in the thesis, we will restrict the amount of work each activity can do by explicitly modelling a scheduler.)

Finite variability A program may only undergo finitely many state changes in a finite amount of time.

Synchronous communication Each communication event requires the simultaneous participation of all involved programs.

Instantaneous events The events have no duration.

[3]There are simpler CSP models, but this (untimed traces) model is the simplest commonly used.

$$\begin{array}{rcl} P & ::= & \mathit{Stop} \mid \mathit{Skip} \mid a \rightarrow P \mid P;P \mid P \,\Box\, P \mid P \sqcap P \\ & & \mid a : A \rightarrow P_a \mid f(P) \mid P \backslash A \mid P \| P \mid P ||| P \\ & & \mid P \|_A P \mid \|^i_{A_i} P_i \mid \mu X \bullet F(X) \end{array}$$

Figure 2.2: Syntax for untimed CSP

Our definition of CSP will be drawn primarily from the paper cited above by Davies and Schneider [20], Schneider's more recent description [101], and the two books by Roscoe and Schneider [95, 102].

2.4.2 Untimed CSP

The language of untimed CSP has the syntax in Figure 2.2. We briefly describe this notation.

'Stop' is the program which will do nothing; it will never interact with its environment. Stop can model a broken printer: it will never accept any jobs; it can do nothing at all.

'$\rightarrow$' is called 'prefix'; '$a \rightarrow P$' is the program which engages in the event a, and then behaves as P. For example, $\mathit{coin} \rightarrow \mathit{Stop}$ is the process which engages in a coin event, and then does nothing.

'Skip' does nothing, except terminate successfully by engaging in the special termination event '$\checkmark$' (pronounced 'tick' or 'success'). Sequential composition is denoted by ';'. The sequential composition of two processes, P and Q, ('$P; Q$') behaves at first as P. If P terminates successfully (*i.e.* P engages in '$\checkmark$'), the composed process then behaves as Q. Thus the process $\mathit{coin} \rightarrow \mathit{Skip}; \mathit{coin} \rightarrow \mathit{Stop}$ can engage in two coin events before behaving as Stop again.

'$P \,\Box\, Q$' means that the environment may choose between P and Q (external choice), whereas '$P \sqcap Q$' results in P or Q being chosen non-deterministically (internal choice). The distinction here is important when considering liveness properties of a process: external choice will offer both possible processes; internal choice does not have to.

'$a : A \rightarrow P_a$' (general, or menu choice) offers a first event from the (possibly infinite) set A. The environment can choose any of the events in A: suppose it chooses a. Once the event a has occurred, the process a :

$A \rightarrow P_a$ then behaves as P_a. This allows the parameterisation of processes based on the particular event.

This permits the modelling of communication along channels with additional syntax. If a data channel c can pass values from a set D, then for any $d \in D$, $c!d$ means 'write the value d on channel c', and is defined as $c.d$, where $c.d$ is a compound event. (Compound events are simply events with additional structure: they have no special meaning, except for simplying descriptions. For instance, if we are only interested that some event, $c.d$ occurs, but are not worried what the specific d is, we may write c.)

Similarly, reading from the channel c is written $c?x : D \rightarrow P_{c.x}$, and is defined as $e : \{x : D|c.x\} \rightarrow P_{c.x}$. This use of structured syntax can be useful for structuring models of systems.

'$f(P)$' is a program P, but with the events relabelled as dictated by f. This has a similar purpose to 'generic instantiation' in high-level programming languages.

'$P \backslash A$' is the program P, but with the events in the set A hidden from the environment. Such hidden events do not need the cooperation of the environment, and occur as soon as P can perform them.

The program '$P \|_A Q$' is the parallel combination in which P and Q synchronise on the events in $A \cup \{\checkmark\}$, and interleave all other events. $\checkmark$ is included in the set of events to synchronise on because both processes must agree to terminate successfully. (This particular variant of parallel composition is often known as 'hybrid parallel'.)

The program '$P \| Q$' denotes 'synchronising parallel', where P and Q cooperate on all events in the intersection of the alphabets of P and Q. The asynchronous (shuffle) parallel operator, '$P ||| Q$' is the program where events of P and Q are interleaved, except for $\checkmark$, which P and Q must cooperate on. Both of these parallel operators can be defined in terms of the hybrid parallel operator.

The final parallel operator is the 'network parallel' (*a.k.a.* general(ised), or interface parallel): '$\|^i_{A_i} P_i$'. This takes a number of processes, P_i, each of which has an associated alphabet A_i, called the 'interface'. For a particular event to occur, then every process P_i which has that event in its interface must agree on that occurrence. $\|^i_{A_i} P_i$ is sometimes written in the form $\|_i (P_i, A_i)$ or $\|\{(P_i, A_i)\}$.

Recursive programs, such as '$\mu X \bullet F(X)$', uniquely satisfy $P = F(P)$ (in the untimed models) if F is guarded. ($\mu X \bullet F(X)$ always has a well-defined semantics in the untimed models.) F is guarded in untimed CSP if every free occurrence of X is preceded by at least one observable (*i.e.* non-hidden) event.

There are a number of semantic models for (untimed) CSP. The simplest presented here is based on observations of events engaged in by the program with its environment. This is known as the traces model. For a program P, $\mathit{Traces}(P)$ is the set of all possible sequences of observable events that P may engage in.

By using trace specifications, we can capture safety properties.[4] For example,

$$\forall tr \in \mathit{Traces}(P) \quad \bullet \quad tr = \langle\rangle$$

requires that P never performs any observable event. (This entire expression is often abbreviated to $tr = \langle\rangle$, where it is understood that tr is the typical trace.) The relation P **sat** S means

$$\forall tr \in \mathit{Traces}(P) \quad \bullet \quad S(tr)$$

where S is a predicate, *i.e.* every trace in P satisfies the specification S.

A clock that ticks has a specification with an infinite set of traces.

$$tr \quad \in \quad \{\langle\rangle, \langle tick\rangle, \langle tick, tick\rangle, \ldots\}$$

This can be satisfied by the recursive program

$$\mu X \bullet tick \to X$$

We can use *refusal sets* to capture liveness (readiness) properties. A refusal set records the events that the process will not engage in. This allows us to define specifications that state that under certain circumstances, the process is not allowed to refuse an event. Even if the environment is not prepared to engage in that event, the process cannot itself refuse it.

The typical refusal set is denoted by rf. A failure is a pair (tr, rf), which means that the program has been observed to engage in the trace tr, and will deadlock if only the events in rf are offered by the environment. For example, we may specify a machine that cannot refuse to produce a chocolate, if the last event engaged in was accepting a token:

$$last(tr) = token \quad \Rightarrow \quad chocolate \notin rf$$

[4]In this context, 'safety properties' are not necessarily related to 'safety' in the sense of 'safety critical'. Here, safety properties constrain which traces may occur.

$$\begin{array}{rll} P & ::= & \mathit{Stop} \mid \mathit{Skip} \mid \mathit{Wait}\ t \mid a \to P \mid P ; P \mid P \,\Box\, P \\ & \mid & P \sqcap P \mid a : A \to P_a \mid P \overset{t}{\rhd} P \mid P \swarrow_t P \\ & \mid & f(P) \mid P \backslash A \mid P \| P \mid P ||| P \mid P \|_A P \\ & \mid & \|^i_{A_i} P_i \mid \mu X \bullet F(X) \end{array}$$

Figure 2.3: Syntax for Timed CSP

The semantic model can be extended to include divergences. Divergences can model livelock. In Hoare's text [46], a divergence of a program P is a trace after which the program behaves as the Chaos process — the most nondeterministic process possible. An alternative treatment (originally due to Reed [93]) associates a stability value (0 or ∞) with each trace or failure. If the stability value is ∞, then the program diverges after performing tr.

2.4.3 Timed CSP

In the untimed traces model, the interval between two consecutive events could be from one millisecond to several million years: we simply do not have that information available. Sometimes, we might want to know exactly how long a particular interval was between two such events. Timed CSP (TCSP) is an extension of untimed CSP to include this timing information.

There are a number of variants of Timed CSP. CSP was originally extended by Reed and Roscoe [92, 93]. Schneider's and Davies' doctoral theses build on this work [23, 100]. As noted above, Schneider has a forthcoming book on the subject [102].

Timed CSP has the syntax in Figure 2.3 (similar to the syntax in Figure 2.2). Times, t, are non-negative real numbers. No lower bound is placed on the interval between consecutive events.

There are three new constructs: Wait (delay), $\overset{t}{\rhd}$ (timeout), and $\swarrow_t$ (timed interrupt):

'$\mathit{Wait}\ t$' does nothing for time t, and then behaves as Skip.

The program '$P \overset{t}{\triangleright} Q$' is a timeout: if P engages in no events before time t has passed, then the program behaves as Q. If the first event of P occurs at precisely time t, then the outcome (P or Q) is nondeterministic. Otherwise, the program behaves as P.

'$P \swarrow_t Q$' is the timed interrupt program. It behaves as P until time t, after which it behaves as Q.

Recursive definitions are considered to be guarded (and thus well-defined) if every free occurrence of X in $F(X)$ is preceded by a uniform non-zero delay.

The untimed notions of trace, refusal, and divergence (stability) can be extended to incorporate time. A timed event is a pair (t, a), where $t \in \mathit{Time} = [0, \infty)$, and $a \in \Sigma$ (where Σ is the set of all observable events.) A timed trace is then a sequence of timed events.

Conceptually, timed refusals are sets of timed events, *i.e.* $(t, a) \in \aleph$ (where $\aleph$ is the typical timed refusal set) indicates that event a was refused at time t.

Usually, timed refusal sets are finite unions of refusal tokens. Each refusal token is of the form $I \times A$, where I is a half-open subset of Time, (say, $I = [l, u) \subseteq \mathit{Time}$) and A is a subset of Σ. The meaning of such a token is that all the events A are refused throughout the interval I. (Davies requires that such intervals are finite, so that if two programs have distinct meanings, then this can be determined in finite time by observations.)

A timed failure is a pair $(\tau, \aleph)$ consisting of a timed trace (τ), and the union of a finite set of refusal tokens ($\aleph$).[5]

Finally, stability values in the timed model are not restricted to only 0 or ∞. The stability value is the earliest time by which all internal activity is certain to have ceased.

The much larger amount of information in timed observations results in more complicated expressions for specification. To address this, specification 'macros' are used as a shorthand. Schneider lists several such idioms [101, Page 156]. But first, we need to define the symbols used in those definitions:

[5]In much of the literature, an untimed failure is represented by (tr, rf), and a timed failure by $(s, \aleph)$. τ has been chosen for the timed trace in this thesis to prevent confusion when s is used in other contexts.

$s_1 {}^\frown s_2$	concatenation of (timed) traces
$s_1 \preceq s_2$	subtrace
$s_1 \ll s_2$	prefix of a trace
$s \# A$	number of occurrences of members of A in s
$\mathit{first}(s)$	first event in a (timed) trace
$\mathit{last}(s)$	last event in a (timed) trace
$\mathit{begin}(\tau)$	first time in a timed trace
$\mathit{end}(\tau)$	last time in a timed trace
$\mathit{head}(s)$	first (timed) event in a (timed) trace
$\mathit{foot}(s)$	last (timed) event in a (timed) trace
$s \downarrow A$	subtrace of a (timed) trace whose events occur in A
$\tau \uparrow I$	subtrace of a timed trace where the times lie in the interval I
$\aleph \downarrow A$	timed refusal set containing only events in A

Some useful specification macros are:

$$\begin{aligned}
a \ \mathbf{at}\ t(\tau, \aleph) &= \langle (t, a) \rangle \preceq \tau \\
a \ \mathbf{open}\ t(\tau, \aleph) &= a \ \mathbf{at}\ t \vee (t, a) \in \aleph \\
a \ \mathbf{closed}\ t(\tau, \aleph) &= \neg a \ \mathbf{at}\ t
\end{aligned}$$

a **at** t is true if (t, a) appears in τ. a **open** t and a **closed** t indicate whether or not the environment of the process was prepared to engage in event a at time t.

$$\begin{aligned}
a \ \mathbf{at}\ I(\tau, \aleph) &= \exists t : I \bullet a \ \mathbf{at}\ t \\
A \ \mathbf{at}\ I(\tau, \aleph) &= \exists a : A \bullet a \ \mathbf{at}\ I \\
a \ \mathbf{open}\ I(\tau, \aleph) &= \forall t : I \bullet a \ \mathbf{open}\ t \\
a \ \mathbf{closed}\ I(\tau, \aleph) &= \neg a \ \mathbf{at}\ I
\end{aligned}$$

a **at** I, a **open** I, and a **closed** I are generalisations to time intervals, and A **at** I is a generalisation to sets of events and time intervals together.

$$\begin{array}{rcl}
a \textbf{ live } t(\tau, \aleph) & = & a \textbf{ at } t \vee (t, a) \notin \aleph \\
a \textbf{ live from } t \textbf{ until } A(\tau, \aleph) & = & [t, \mathit{begin}(\tau \uparrow [t, \infty) \downarrow A)) \times \{a\} \cap \aleph = \emptyset \\
a \textbf{ live within } \delta \textbf{ of } t \textbf{ until } A & = & \quad A \textbf{ at } [t, t+\delta) \\
 & & \vee \quad a \textbf{ live from } t+\delta \textbf{ until } A \\
B \textbf{ live from } t \textbf{ until } A(\tau, \aleph) & = & [t, \mathit{begin}(\tau \uparrow [t, \infty) \downarrow A)) \times B \cap \aleph = \emptyset \\
B \textbf{ live within } \delta \textbf{ of } t \textbf{ until } A & = & \quad A \textbf{ at } [t, t+\delta) \\
 & & \vee \quad B \textbf{ live from } t+\delta \textbf{ until } A
\end{array}$$

a **live** t indicates that the process is prepared to perform the event a at time t, and a **live from** t **until** A indicates that a is 'live' until some event in A occurs (usually, $a \in A$, otherwise the specification is unsatisfiable). The formulation a **live within** δ **of** t **until** A allows for the case when there is some uncertainty about when the event becomes live: it gives a time interval. The last two formulæ are again generalisations to sets of events.

2.4.4 Hierarchy and Refinement

Reed presents a hierarchy of models for (Timed) CSP [93]. Figure 2.4 illustrates the hierarchy, and contains 9 models. The first four are untimed: the most basic model here is M_T — the traces model. M_F and M_S are the trace-failures and trace-stability (divergence) models, which are combined in M_{FS}. The four models TM_T, TM_F, TM_S, and TM_{FS} are the timed analogues of the four untimed models. TM^*_{FS} bridges a gap for the time-wise refinement in Schneider's work [100].

This hierarchy of models within a uniform theory is a major advantage of CSP. For a given system, the model most appropriate to the properties that we wish to study can be chosen.

Within each semantic model, we can use the notion of satisfaction (using **sat** — Page 37) to determine whether or not a a process satisfies a specification predicate.

An alternative is to use a (simple, obviously correct) program as a specification. This is approach is used in the FDR tool [28]. We say that the

$$\begin{tikzcd}
 & & TM_{FS} \arrow[ddll] \arrow[d] \arrow[dr] & & \\
 & & M_{FS} \arrow[dl] \arrow[dr] & TM^{*}_{FS} \arrow[dr] & \\
TM_{F} \arrow[r] \arrow[ddrr] & M_{F} \arrow[dr] & & M_{S} \arrow[dl] & TM_{S} \arrow[l] \arrow[ddll] \\
 & & M_{T} & & \\
 & & TM_{T} \arrow[u] & &
\end{tikzcd}$$

Figure 2.4: Models of Timed CSP

program P is refined by the program Q, written

$$P \sqsubseteq Q$$

if and only if every possible observation of Q is an observation of P:

$$P \sqsubseteq Q \quad \Leftrightarrow \quad \mathcal{O}[\![\, Q \,]\!] \subseteq \mathcal{O}[\![\, P \,]\!]$$

(In this expression, $\mathcal{O}[\![\, P \,]\!]$ means the set of observations of P in the semantic model used for the problem.) This could also be described, 'Q refines P', 'Q implements P', or 'Q is an implementation of the specification P'.

2.4.5 Summary and Remarks

The theory behind the various models of CSP is well-understood [18]. Hoare's text includes a proof theory for untimed CSP [46]. Schneider has described an operational semantics for TCSP [103], and a further paper describes the timed trace-failures model with both an algebraic (denotational) and behavioural requirements specification [104].

Examples of CSP and TCSP in use can be found in any of the cited references. Two further papers by Davies and Schneider [21, 22] describe Timed CSP with well-presented examples.

Section 3.1 (Page 53) gives the rationale for using Timed CSP as the semantic domain in this thesis.

2.4.6 Other Notations

There are so many other notations, methods and tools that it is impractical to list them here. Instead, the reader could profitably reference the two NASA manuals [75, 76], and McDermid's text [70].

Some other interesting notations are very briefly described below.

CCS

Milner's Calculus of Communicating Systems (CCS) [73, 74] was a major breakthrough in the mathematical modelling of concurrency. Agents (processes) are denoted by mathematical expressions, and a series of equivalences are defined between those expressions. There are (as expected) notational differences between CCS and CSP, for instance, '$a.P$' in CCS corresponds to '$a \rightarrow P$' in CSP.

An important difference between CSP and CCS concerns the treatment of hiding. Hidden events in CSP leave no trace; CCS hidden events are replaced with the internal transition symbol 'τ'. This can be used to distinguish between divergent processes.

Timed variants of CCS have been developed, and Hennessey has described a variant based on 'actions' with a strictly positive duration [43]. Gurr has combined CCS and Statecharts [35].

Petri Nets

Petri's doctoral thesis in 1962 is the origin of net theory [89]. Reisig's book [94] is a comprehensive introduction to the subject.

Hesketh is currently developing methods for the synthesis of expressions from 'Petri boxes' (labelled Petri nets) [44]. There are many variants and derivatives of Petri nets and net theory.

Methods for translating CCS and TCSP to Petri nets are the concern of Taubner's doctoral thesis [116]. Similarly, the Duration Calculus, (an extension of the Interval Temporal Logic,) has been used by Hansen *et al.* to give an operational semantics to a timed variant of CSP [38]. However, when a 'native' definition of Timed CSP is available (as is the case for this thesis), there is no need to perform further translations.

TLA

Lamport's Temporal Logic of Actions (TLA) [60] is a small logic for specifying and reasoning about concurrent systems. It incorporates a proof theory for refinement. A later text describes 'predicate-action diagrams' which have a meaning in terms of TLA [61]. There are a number of further reports concerning TLA, including Abadi and Lamport's work on conjoining specifications [1], and on proving safety and liveness properties [62].

Other Models

Shaw's Communicating Real-Time State Machines are a "new, complete, and executable notation for specifying concurrent real-time systems" [106].

CRSMs are a graphical notation with an operational semantics and provision for pre- and post-conditions on transitions. However, it is not clear that this notation will scale up to larger problems.

A more mathematical approach to concurrency is Pratt's partially ordered multisets (pomsets) [91]. These are very abstract; however, one example shows that this approach may provide extremely elegant descriptions of some problems. In a similar vein, Goubault has described higher-dimensional automata [31] (which are a generalisation of nondeterministic finite dimensional automata).

2.5 Notational Issues

We briefly examine how formal languages are defined. Schmidt notes that there are three main aspects to a programming language: syntax, semantics and pragmatics [99]:

Syntax concerns the appearance and structure of sentences in the language. Is a given term a member of the language (is it 'legal')? This is usually defined in a notation known as Backus-Naur Form (BNF), which is described in many texts, including Schmidt's.

BNF is used in this thesis to define the syntax for the DORIS notation (Page 62).

Semantics concerns the assignment of meanings to the sentences. There are many methods for giving a language a semantics, including:

Operational semantics are a description of a language in terms of an abstract machine, or an interpreter [99, 114]. Hennessy gives a number of excellent examples [42]. Essentially, the description of the language is written in terms of a more primitive language which is interpreted, for example, the step semantics of some Statechart variants [6]. Milner uses another variant of operational semantics for some definitions of CCS [74].

This raises the general point that syntax and semantics are formulated in formal languages which themselves have a syntax and semantics which must be defined somehow.

Denotational semantics Schmidt's and Stoy's texts provide fine descriptions of denotational semantics [99, 114].

The idea behind denotational semantics is that a semantic function is applied to a legal term in the language, which provides the meaning of the term as a mathematical object. Reasoning about the language is then performed upon the resulting mathematical objects.

Axiomatic semantics take a more abstract view. A consistent collection of axioms are given for the language, and properties derived from them. In particular, the properties of a given term are derived by applying the axioms.

Algebraic semantics An example of an algebraic approach is given in Hoare's 1987 text [47]. The underlying model is based upon an algebra, and provability coincides with equational reasoning.

It has been suggested that an adequate understanding of a language is attained when it possesses several different types of semantics, and that all are consistent with each other. For instance, Schneider notes that

> "in providing an operational semantics for a language [*i.e.* Timed CSP] which is already endowed with a denotational semantics, our intention is to provide an operational intuition for the language constructs, and hence gain some insight..." [103]

In Section 3.1.1 (Page 54), we give the rationale for our choice of semantics.

Pragmatics concerns the usability of the language. Can it be implemented relatively easily? Later in the thesis, this includes: 'Can systems described in this notation be reasoned about (relatively) easily?'

Stoy [114], Schmidt [99], and Dijkstra and Scholten's [25] texts cover these topics in considerable detail. Van Leeuwen's text [63] too, covers these issues, as well as a wide range of other topics in the formal methods domain.

2.6 Mechanical Proving

A complaint often leveled against formal methods is the lack of adequate tool support [8]. While this is to an extent a valid complaint, there is an increasingly wide range of tools. Importantly, understanding of how to exploit these tools is also improving.

The reader is referred once again to the text by Rushby [97] and the NASA guidebooks [75,76] for a thorough coverage of available formal methods tools.

Two contrasting tools, FDR and PVS, are briefly discussed here. We also mention MADGE, a tool designed explicitly for use with MASCOT-3 by British Aerospace.

2.6.1 FDR

FDR is essentially a state exploration tool [28]. It can take two CSP processes defined as programs, determine the state spaces of these two processes, and then determine whether one refines the other in a given model (see Section 2.4.4 for a short description of refinement in CSP). This can be used for checking if a process satisfies a specification (itself defined as a process). Currently, this does not extend to the timed models of CSP.

An example of its use is in the author's analysis of the four-slot implementation of the pool [9]. The FDR model (see the cited paper for details of the model) was checked against the specification below:

$$\begin{aligned} FDRAsyncReader &= sr \rightarrow \left(\bigsqcap d : D \bullet er!d \rightarrow FDRAsyncReader\right) \\ FDRAsyncWriter &= sw?d \rightarrow ew \rightarrow FDRAsyncWriter \\ FDRAsync &= FDRAsyncReader || FDRAsyncWriter \end{aligned}$$

This specification was coded in FDR

```
SAREADING = esr -> (|~| d :  DATAVALUES
                   @ eer!d -> SAREADING)
SAWRITING(d) = esw?d -> eew -> SAWRITING(d)
SASYNC = SAREADING ||| SAWRITING(0)
```

and an assertion given to test this

```
assert SASYNC [F= FOURSLOT
```

(The assertion means: Check whether `FOURSLOT` refines `SASYNC` in the failures model.) In this instance, `FOURSLOT` refined `SASYNC`.

FDR appears to be useful for fairly simple, easily defined processes, and for these, is quite trouble-free. It would appear that it also handles much larger processes. Like any other state-space checking tool, it has the disadvantage of not being able to handle abstract programs that have very large state machine representations.

2.6.2 PVS

PVS is a higher-order logic specification and verification system [79, 80, 105]. It is LISP-based, industrial-strength, and is very flexible, although the Emacs interface and step-wise proof commands are not user-friendly. It is not easy to learn to use this tool.

Dutertre and Schneider have recently carried out work on embedding (the untimed traces model of) CSP in PVS [26, 27]. This thesis includes further work on embedding CSP in PVS, in Chapter 6 (Page 133).

2.6.3 MADGE

The MASCOT-3 Design Generator (MADGE) is a British Aerospace tool to aid the MASCOT method [65, 66]. It has specific extensions to handle Simpson's IDAs (see Chapter 4).

This tool has a graphical interface, version control, and file and project management facilities. Systems can be constructed in the graphical editor. It can also generate source code for several languages, including the SPARK subset of Ada.

This thesis refers to a prototype tool, `dt` (Appendix B). It is hoped that this could be linked into MADGE as an alternative code generator: this would result in CSP being generated, which could then be tested against specifications.

2.6.4 Other Tools

The NASA guidebook referred to at the beginning of this chapter contains a list of formal methods tools and techniques [75, 76]. Although it is not exhaustive (the field is expanding rapidly), it is reasonably thorough.

Once the choice of using Timed CSP as a semantic domain was made (see Section 3.1 for this rationale), it is then relatively easy to choose the tool support. Two tools were chosen to represent a complementary approach: it quickly became apparent that using only one approach (theorem proving or state exploration) would not be sufficiently powerful for the detail and properties examined.

FDR, as a CSP based state exploration tool, was an obvious choice. The choice of theorem provers is slightly harder: HOL and IMPS have both been used for embedding CSP. Chapter 6 contains further details about the mechanical support employed in this thesis.

2.7 Summary of Survey

This survey has

- described the general problem domain (critical systems);
- introduced formal methods;
- described the specific notation for the problem;
- outlined the untimed and timed variants of the language of CSP; and
- introduced the tools used to support our work.

Part II

Theoretical Semantics

Chapter 3

Denotational Semantics for DORIS

MASCOT and DORIS were introduced in Sections 2.3.1 and 2.3.2 (Pages 30 and 33 respectively).

In this thesis, we are concerned only with the meaning of the design notations and the Data Interaction Architecture (DIA). More generally, Simpson states that DORIS 'provides the methodological framework in which the notational conventions of DIA have been developed' [110], *i.e.* the notation exists within a more general software engineering method (although, as we will see in this thesis, it can easily stand apart as a separate entity).

This chapter describes DORIS informally, and introduces some of the terminology that we use. We then give a denotational semantics using Timed CSP as the semantic domain. This is significant because it permits formalization and mechanization of proofs and checks about systems designed using DORIS.

3.1 Rationale

Firstly, why give a denotational semantics (as opposed to a different form of semantics — see Section 2.5, Page 45)? Why are we using a process algebra as the semantic domain, and why Timed CSP in particular?

3.1.1 Denotational Semantics

A denotational semantics performs a translation from one notation to another. In many ways, this is the simplest way to give a notation a consistent semantics.

Operational semantics, although understandable (the concept of an imperative machine is well-understood), tend to be hard to reason with. An axiomatic semantics, while ideal for reasoning, is harder to construct. In particular, the construction of a consistent set of axioms is a non-trivial problem. Our remaining choice, algebraic semantics, is also difficult to construct.

Note that these arguments do not mean that we should never use an alternative form of semantics: indeed, it would be useful to have different, but consistent descriptions of the semantics of the DORIS notation (compare with Hoare and He's book [45]). However, we start with denotational semantics as a compromise for simplicity of description, and for powerful use.

3.1.2 Timed CSP

Process algebras are well-understood mathematical objects. Because they are small, well-defined theories, they are well-suited as semantic domains.

(Timed) CSP is one of the best known of the process algebras, and has the most well-developed timed variant. Since DORIS applications are typically real-time applications, we must be able to model time. Moreover, we need more than just event ordering: we will sometimes need to know exactly how much time passes between specific events.

Further, there is considerable effort being put into making Timed CSP amenable to machine checking. Scattergood has developed a machine-readable language (necessary because CSP has developed as a 'blackboard' language) [98]. The tool FDR uses such a language, and for many applications, is sufficient [28]. Dutertre and Schneider have adopted an approach that is also used later in this thesis, where CSP is embedded in PVS [26, 27].

Timed CSP has a number of operators that are very useful in this type of problem: network parallel allows a number of processes to agree on some events, but not others. The hiding operator is particularly useful for abstraction. The algebraic laws of Timed CSP allow some hope that a usable theory can be developed for DORIS designs.

Finally, Timed CSP has a notion of refinement that can be exploited for different layers. Essentially, we can 'fill in' further detail as we pursue the development process of a system, and show that the more detailed Timed CSP at each point refines the previous point.

3.2 Informal Description of DORIS

The DORIS notation

- is hierarchical;
- captures concurrency; and
- represents data flow.

DORIS does not attempt to describe the detailed behaviour of components of the system: this is typically described in other design documentation. (In later parts of the thesis, these components will be described in Timed CSP, or a simpler language based upon Timed CSP.)

The system consists of components (including activities, servers, routes, and paths), and can be viewed from any of the four layers of the *Data Interaction Architecture*.

3.2.1 Data Interaction Architecture

The four layers of DIA are given in Figure 3.1 (Figure 3 of Simpson's paper [112]).

The functional design describes the behaviour of the system: for instance, a generator passes data to a user via a particular protocol. More abstractly, this layer could be a specification of the system in terms of transactions across the inputs and outputs of the system. The style of specification does not matter as long as the meaning is clear.

The next layer (the design layer) indicates that the generator writes to a reader via a route (an implementation of a protocol). In the rest of the design, the various activities and routes are connected via paths, and are grouped into complex forms.

The design is then distributed (the distribution layer). This involves placing the route on the processor where the reader resides; the writer may

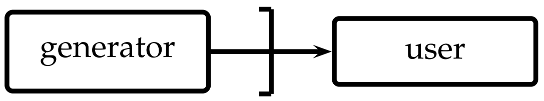
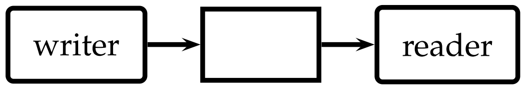
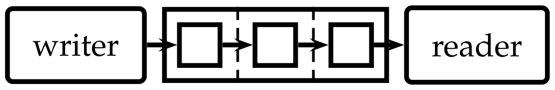
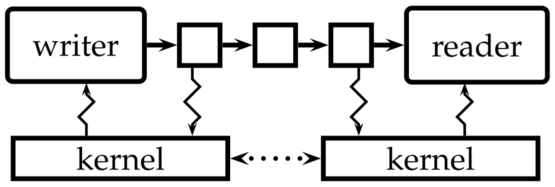
Layer	Representation Example	Focus
Functional		What
Design		How
Distribution		Where
Execution		When

Figure 3.1: Layers of the Data Interaction Architecture

be on the same processor, or on a directly-linked processor, or even on a processor which can only be linked indirectly through a third processor on the network.

The execution layer is the final mapping onto processors running kernels which communicate with each other. This introduces detailed timing information into our model.

Figure 3.1 illustrates this development by starting with a 'generator' connected to a 'user' via a 'pool' (a particular protocol) in the functional layer. An implementation of the pool is chosen in the design layer, and the implementation of the pool is split into three when the design is distributed across the network. Finally, the kernels and the communications infrastructure are added at the execution layer.

This thesis concerns all the layers to some degree. The semantic functions we develop are primarily concerned with the middle two layers, although a notion of refinement between all the layers will be developed towards the end of this chapter (Section 3.9, Page 74).

3.2.2 Entities of a DORIS System

We can now introduce the terminology for the entities that make up a DORIS design. Although this is similar to the MASCOT description (Section 2.3.1, Page 30), it is not identical. Hence, this section serves as the first step in giving a formal semantics to the DORIS notation.

A DORIS design consists of a number of entities. The first of these is the *system*. This is the top of the hierarchy, and contains all the other entities.

At the bottom end of the tree, there are the *activities*, which carry out the work of the system. *Servers* are specialized activities that interact with hardware, such as sensors and actuators. This hides details of the actual hardware interface from the rest of the design process.

The activities and servers are 'active' components and are never connected to each other directly. Instead, they are connected via 'passive' components called *routes*.

A *protocol* defines the dynamic constraints (*e.g.* timing and blocking constraints) on the data transfers. A route is an implementation of a particular protocol. Protocols and routes are jointly referred to as *IDAs*. However, the three terms are often used interchangeably; this is the case in this thesis. Thus a route is a component of a system which transfers data: it may be a channel (buffer), pool, or signal. There are others: Simpson has described a wide range of protocols [112].

The various entities are connected by *paths*: these are the parts of the system that glue the other components together. Paths connect *ports* to *windows*: ports are at the active end of the path, and windows are at the passive end. Activities have only ports; routes have only windows; servers may have both.

These paths are one-way, and are from a window to a port, or from a port to a window. *Complex paths* provide a means for paths to be grouped together.

The system can be hierarchically decomposed into subsystems: these are the *complex servers*, *complex activities*, and *devices*.[1] These complex forms may contain other DORIS entities: they are distinguished by particular restrictions: a device may only contain passive elements and have windows for an interface. Complex activities may have only ports. Complex servers may have both ports and windows.

[1]Note that MASCOT terminology calls the complex forms 'composite'.

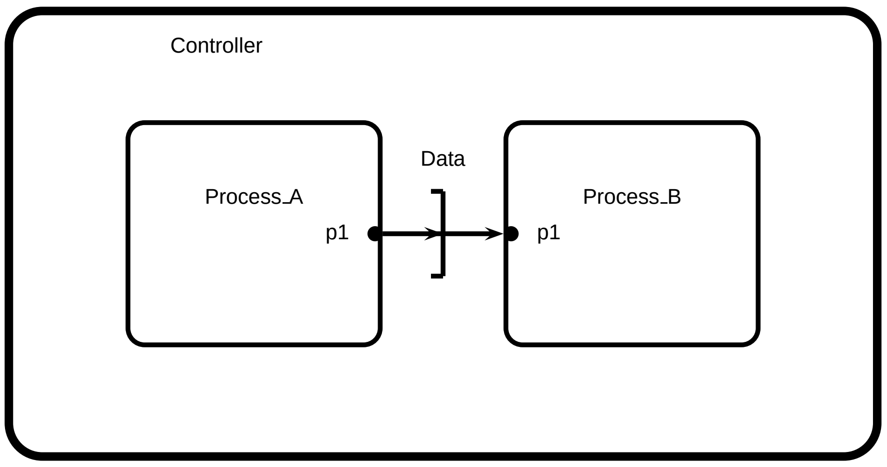
3.3 Syntax

Beyond the syntax defined for MASCOT, there is no published syntax for DORIS. Before we can give a semantics for DORIS, we must first give it a syntax.

The syntax we define here is driven by both the informal description given earlier in this chapter, and how the notation has been used in practice on a large system (in particular, the case study from Section 8.5). At this point, we concentrate on the design and distribution layers only: the design layer concentrates on what should be done when. The distribution layer adds more concrete details: it says what should be done when, and also where is should be done in the network.

We will illustrate the syntax more informally by way of two examples: Figures 3.2 and 3.3. Both examples have a graphical and textual description (although the paths and IDAs are omitted from the second example to avoid cluttering the diagram).

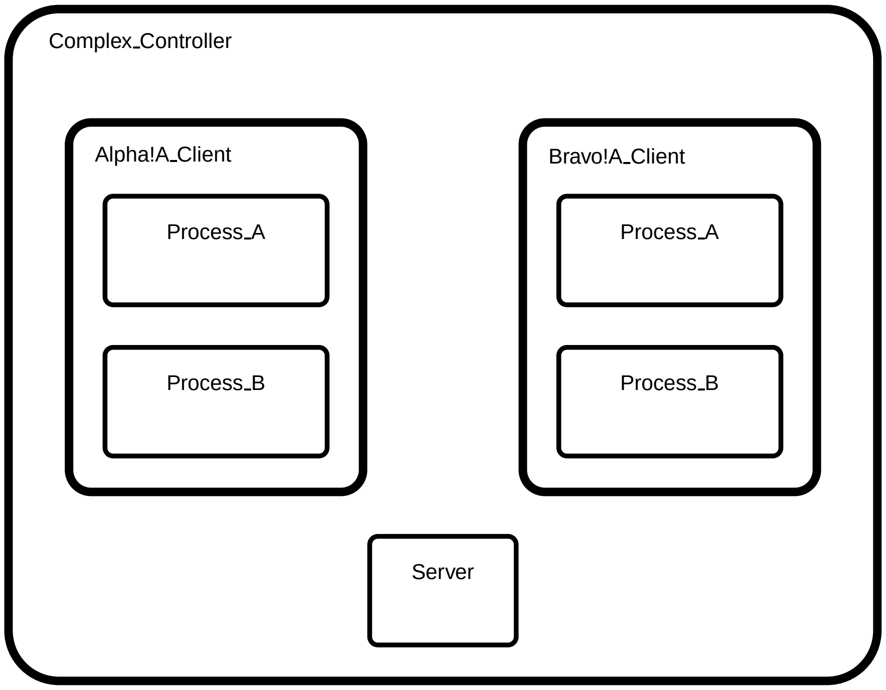
The graphical syntax, with components embedded in other components, is a natural way of drawing hierarchically decomposed systems. The textual notation, however, is based on a strict distinction between hierarchies.



```
(
     (sy, Controller,
                    (sa, Process_A),
                    (sa, Process_B),
                    (rt, Data, Pool),
                    (sp, , Process_A:p1, Data:w1),
                    (sp, , Data:w2, Process_B:p1)
     )
)
```

Code stubs expected are: Process_A
Process_B

Figure 3.2: A simple example of the DORIS notation



```
(
    (sy, Complex_Controller,
                          (ca, Alpha!A_Client),
                          (ca, Bravo!A_Client),
                          (sa, Server)
    ),
    (cc, A_Client,
                (sa, Process_A),
                (sa, Process_B)
    )
)
```

Code stubs expected are: Process_A
Process_B
Server

(Routes and paths not included)

Figure 3.3: A more complex example of the DORIS notation

3.3.1 Tags

Later in this chapter, we will need to identify the different types of components. To do this, we introduce *tags*:

Entity Type	Tag	Description
System	sy	Top-level system: there should be exactly one of these in each system.
Transputer	tp	A single processor in a distributed system.
Direct T-link	td	A bidirectional link between two adjacent transputers.
Indirect T-link	ti	A bidirectional link between two transputers via a third transputer.
Complex server	cs	A server that contains other entities.
Complex activity	ca	An activity that contains other entities.
Simple server	ss	A server that contains no other entities.
Simple activity	sa	An activity that contains no other entities.
Device	dv	A complex route: often contains several routes and paths, or other devices.
Route	rt	A single IDA, with one writer window and one reader window.
Complex path	cp	One or more one-way data paths from a complex window to a complex window, or from a complex port to a complex port.
Simple path	sp	A single one-way data path from a window to a port, or from a port to a window.

Although the large case study is implemented on a transputer network, this semantics is not limited to transputer architectures. However, the word ‘transputer’ will appear throughout the thesis to distinguish a piece of hardware with processing capability. (‘T-link’ is an abbreviation for ‘transputer link’.)

3.3.2 BNF-style Notation

First, we need to identify the notation for defining our syntax. The context-free syntax of the language is described using a simple variant of Backus-Naur Form.

In particular,

$|$ denotes choice;

$\{\ldots\}^+$ denotes one or more instances;

$\{\ldots\}$ denotes zero or more instances; and

$[\ldots]$ denotes zero or one instances.

Round brackets, ‘$(\ldots)$’, are used for grouping terms, and literal text is represented by sans serif text in quotes, ‘“. . . ”’.

3.3.3 Complex Constructs

A DORIS textual design is a set of complex constructs. Each complex construct is a tuple: the first element of each tuple is a tag, and the second element is a name. The remaining elements of each complex construct tuple are basic constructs, which are themselves also tuples. (The basic constructs are described in the next section.)

The first complex construct is the top-level syntactic element, the ‘system’:

$$\begin{array}{rcl}
\langle System \rangle & ::= & (\langle System\_Design' \rangle \mid \langle System\_Distributed' \rangle) \\
 & & \{\langle Complex\_Server' \rangle \\
 & & \ \mid \langle Complex\_Activity' \rangle \\
 & & \ \mid \langle Device' \rangle\} \\
\langle System\_Design' \rangle & ::= & \text{“(sy,”}\langle Name \rangle \\
 & & \{\text{“,”}\langle Component \rangle\}^+\text{“)”}
\end{array}$$

$$\begin{array}{rcl} \langle \mathit{System\_Distributed}' \rangle & ::= & \text{“(sy,”} \langle \mathit{Name} \rangle, \\ & & \{ \text{“,”} \langle \mathit{Transputer} \rangle \mid \text{“,”} \langle \mathit{Direct\_T\_Link} \rangle \\ & & \mid \text{“,”} \langle \mathit{Indirect\_T\_Link} \rangle \mid \text{“,”} \langle \mathit{Component} \rangle \}^{+} \text{“)”} \end{array}$$

- A design-level system may not contain any transputers, nor any direct or indirect transputer links. This type of system has been decomposed functionally, but not across a network.
- A distribution-level system may contain transputers, and direct and indirect transputer links. This is the result of a design-level system being distributed across a network: thus the system requires information about the network.

The remaining complex constructs are the complex server, complex activity, and device:

$$\begin{array}{rcl} \langle \mathit{Complex\_Server}' \rangle & ::= & \text{“(cs,”} \langle \mathit{Name} \rangle, \{ \text{“,”} \langle \mathit{Component} \rangle \}^{+} \text{“)”} \\ \langle \mathit{Complex\_Activity}' \rangle & ::= & \text{“(ca,”} \langle \mathit{Name} \rangle, \{ \text{“,”} \langle \mathit{Component} \rangle \}^{+} \text{“)”} \\ \langle \mathit{Device}' \rangle & ::= & \text{“(dv,”} \langle \mathit{Name} \rangle, \{ \text{“,”} \langle \mathit{Passive} \rangle \}^{+} \text{“)”} \end{array}$$

In turn, the definitions above refer to the components that they contain:

$$\begin{array}{rcl} \langle \mathit{Component} \rangle & ::= & \langle \mathit{Active} \rangle \mid \langle \mathit{Passive} \rangle \\ \langle \mathit{Active} \rangle & ::= & \langle \mathit{Complex\_Server} \rangle \\ & & \mid \langle \mathit{Complex\_Activity} \rangle \\ & & \mid \langle \mathit{Simple\_Server} \rangle \\ & & \mid \langle \mathit{Simple\_Activity} \rangle \\ \langle \mathit{Passive} \rangle & ::= & \langle \mathit{Device} \rangle \\ & & \mid \langle \mathit{Route} \rangle \\ & & \mid \langle \mathit{Complex\_Path} \rangle \\ & & \mid \langle \mathit{Simple\_Path} \rangle \end{array}$$

We will define these in the next section.

3.3.4 Basic Constructs

The basic construct tuples also start with a tag and a name, and then hold further information particular to that tag. Some instances of basic constructs refer to other complex constructs, *e.g.* ‘A_Client’ in Figure 3.3.

The first constructs give information about the network (if applicable):

$$\begin{array}{rcl}
\langle \mathit{Transputer} \rangle & ::= & \text{“(tp,”} \langle \mathit{Name} \rangle \text{“)”} \\
\langle \mathit{Direct\_T\_Link} \rangle & ::= & \text{“(td,”} \langle \mathit{Name} \rangle \text{“,”} \langle \mathit{T\_Source} \rangle \text{“,”} \\
 & & \langle \mathit{T\_Destination} \rangle \text{“)”} \\
\langle \mathit{Indirect\_T\_Link} \rangle & ::= & \text{“(ti,”} \langle \mathit{Name} \rangle \text{“,”} \langle \mathit{T\_Source} \rangle , \\
 & & \langle \mathit{T\_Destination} \rangle \text{“,”} \langle \mathit{T\_Via} \rangle \text{“)”}
\end{array}$$

We can declare instances of the complex and simple forms of servers and activities:

$$\begin{array}{rcl}
\langle \mathit{Complex\_Server} \rangle & ::= & \text{“(cs,”} \langle \mathit{Instance\_Name} \rangle \text{“)”} \\
\langle \mathit{Complex\_Activity} \rangle & ::= & \text{“(ca,”} \langle \mathit{Instance\_Name} \rangle \text{“)”} \\
\langle \mathit{Simple\_Server} \rangle & ::= & \text{“(ss,”} \langle \mathit{Instance\_Name} \rangle \text{“)”} \\
\langle \mathit{Simple\_Activity} \rangle & ::= & \text{“(sa,”} \langle \mathit{Instance\_Name} \rangle \text{“)”}
\end{array}$$

The definition of simple servers and activities will be discussed further in Chapter 5. The passive components, the route and device (a complex route), can also be instantiated here.

$$\begin{array}{rcl}
\langle \mathit{Device} \rangle & ::= & \text{“(dv,”} \langle \mathit{Instance\_Name} \rangle \text{“)”} \\
\langle \mathit{Route} \rangle & ::= & \text{“(rt,”} \langle \mathit{Name} \rangle \text{“,”} \langle \mathit{Route\_Type} \rangle \text{“)”}
\end{array}$$

The routes will be discussed in the next chapter, Chapter 4. Finally, we define paths, which link all the above components together.

$$\begin{array}{rcl}
\langle \mathit{Complex\_Path} \rangle & ::= & \text{“(cp,”} \langle \mathit{Opt\_Name} \rangle \text{“,”} \langle \mathit{Path\_Source} \rangle \text{“,”} \\
 & & \langle \mathit{Path\_Destination} \rangle \text{“)”} \\
\langle \mathit{Simple\_Path} \rangle & ::= & \text{“(sp,”} \langle \mathit{Opt\_Name} \rangle \text{“,”} \langle \mathit{Path\_Source} \rangle \text{“,”} \\
 & & \langle \mathit{Path\_Destination} \rangle \text{“)”}
\end{array}$$

There are a number of smaller definitions used above, *e.g.* a single component can be included within several other components by using the

$\langle Instance\rangle$"!"$\langle Target\rangle$ notation: Figure 3.3 is such an example. (The same consideration applies to using multiple instances of simple servers and simple activities.)

$$\begin{array}{rcl}
\langle Opt\_Name\rangle & ::= & \langle Name\rangle \mid \langle empty\_string\rangle \\
\langle Instance\_Name\rangle & ::= & \langle Target\rangle \mid \langle Instance\rangle\text{“!”}\langle Target\rangle \\
\langle Target\rangle & ::= & \langle Name\rangle \\
\langle Instance\rangle & ::= & \langle Name\rangle \\
\langle Name\rangle & ::= & \langle string\rangle \\
\langle Path\_Source\rangle & ::= & \ldots \\
\langle Path\_Destination\rangle & ::= & \ldots \\
\langle T\_Source\rangle & ::= & \langle string\rangle \\
\langle T\_Destination\rangle & ::= & \langle string\rangle \\
\langle T\_Via\rangle & ::= & \langle string\rangle \\
\langle Route\_Type\rangle & ::= & \langle string\rangle
\end{array}$$

The $\langle Path\_Source\rangle$ and $\langle Path\_Destination\rangle$ will be covered in detail in Section 3.8.1 (Page 73).

It can be seen that this syntax is a set consisting of one system, and any number of complex servers, complex activities and devices.

Section B.1 (Page 223) gives a complete definition of the plain text input language that the DORIS-to-CSP tool, dt, manipulates, with an example in Section B.2 (Page 225).

3.4 The System Tuple

The grammar in Section 3.3 (Page 58) defines a string. This string represents a tuple, which we name $\mathcal{S}$. $\mathcal{S}$ is a tuple of complex constructs:

$$\mathcal{S} = (\mathcal{SY}, \mathcal{CS}_1, \ldots, \mathcal{CS}_{N_S}, \mathcal{CA}_1, \ldots, \mathcal{CA}_{N_A}, \mathcal{DV}_1, \ldots, \mathcal{DV}_{N_D})$$

where

$\mathcal{SY}$	is the $\langle \mathit{System\_Design}' \rangle$ or $\langle \mathit{System\_Distributed}' \rangle$ construct;
$\mathcal{CS}_i$	is the ith $\langle \mathit{Complex\_Server}' \rangle$;
$\mathcal{CA}_i$	is the ith $\langle \mathit{Complex\_Activity}' \rangle$;
$\mathcal{DV}_i$	is the ith $\langle \mathit{Device}' \rangle$;
N_S	is the number of $\langle \mathit{Complex\_Server}' \rangle$ tuples;
N_A	is the number of $\langle \mathit{Complex\_Activity}' \rangle$ tuples; and
N_D	is the number of $\langle \mathit{Device}' \rangle$ tuples.

The syntax arranges that the structure of a particular system is reminiscent of a block-structured programming language: it has a 'main' procedure ($\mathcal{SY}$), and additional procedures (the other components of $\mathcal{S}$). The semantics continues this analogy by defining the meaning of the system in terms of $\mathcal{SY}$, referring to the other components of $\mathcal{S}$ when further information is required.

Each of $\mathcal{SY}$, $\mathcal{CS}_i$, $\mathcal{CA}_i$ and $\mathcal{DV}_i$, is a tuple, and is known as a *complex component*.

The elements within each complex component tuple are denoted by subscripts, *e.g.* the jth element of the tuples $\mathcal{SY}$ and $\mathcal{CS}_i$ are $\mathcal{SY}_j$ and $\mathcal{CS}_{i,j}$ respectively.

3.5 Static Semantics

There are a number of constraints to be applied to this system. For example, if a complex activity is referred to in another complex component, then it must be defined somewhere.

The static semantics for the DORIS notation are given in Appendix A (Page 213). (We omit them here as they enforce some 'reasonable' expectations about the systems we might define.)

3.6 Semantics

We can now give a semantics for this notation by a number of semantic functions on the syntax described above. These functions result in a Timed CSP program for the particular system. (See Section 2.4 and Appendix C for the description of Timed CSP used in this thesis.)

The semantics takes the tuple, $\mathcal{S}$, which represents the whole design, and extracts the single tuple describing the system component, $\mathcal{SY}$. The meaning of $\mathcal{SY}$ is the parallel composition of its own (basic) components' Timed CSP meaning. Part of the calculation of these basic components requires the analogous construction of the meaning of the other complex components in the top-level tuple, $\mathcal{S}$. These components are then linked together using contextual information about ports and windows, which is determined from the paths in the complex components.

3.6.1 Basic Components

The function, $\mathcal{B}$, gives the meaning of a given basic component. It consists of a tuple of three parts:

$$\mathcal{B}[\![\; c \;]\!] = (\mathcal{B}_P[\![\; c \;]\!], \mathcal{B}_I[\![\; c \;]\!], \mathcal{B}_W[\![\; c \;]\!])$$

where c is the basic component that we are interested in.

$\mathcal{B}_P[\![\; c \;]\!]$ is the CSP process representing c;

$\mathcal{B}_I[\![\; c \;]\!]$ is the set of events concerned with IDA communications; and

$\mathcal{B}_W[\![\; c \;]\!]$ is the set of events concerned with scheduling (work).

In the next section, we will see why we have these three-part tuples, rather than just having a single element for the process: briefly, the complex components are constructed by taking the CSP process for each basic component within that complex component. These are composed together using the CSP parallel operator, which also requires an alphabet: the set of IDA communications events. The final part is collected together for the scheduler model.

We are nearly ready to give the definition of $\mathcal{B}$ for each tag. Before that, we need to define several 'helper' functions which link in definitions from later chapters.

We define the functions $\mathcal{NSS}$, $\mathcal{NAS}$ and $\mathcal{NRS}$ to return the three-part meaning for simple servers, simple activities, and simple routes respectively in the next two chapters. Note that we do not define any events in this chapter: the relevant events are also defined in the next two chapters.

The functions, $\mathcal{NCS}$, $\mathcal{NCA}$, and $\mathcal{NDV}$ are defined later in this chapter, and return the three-part meaning for complex servers, complex activities, and devices respectively.

We can summarise these six functions in the following table:

Function	…returns the meaning of a	defined on Page…
$\mathcal{NCS}$	complex server	70
$\mathcal{NCA}$	complex component	70
$\mathcal{NDV}$	device	70
$\mathcal{NSS}$	simple server	120
$\mathcal{NAS}$	simple activity	120
$\mathcal{NRS}$	route	98

The definition of $\mathcal{B}$ for each tag is as follows:

$$\begin{aligned}
\mathcal{B}[\![(\mathsf{tp}, n)]\!] &= (\mathit{Null}, \emptyset, \emptyset) \\
\mathcal{B}[\![(\mathsf{td}, n, x, y)]\!] &= (\mathit{Null}, \emptyset, \emptyset) \\
\mathcal{B}[\![(\mathsf{ti}, n, x, y, z)]\!] &= (\mathit{Null}, \emptyset, \emptyset) \\
\mathcal{B}[\![(\mathsf{cs}, n)]\!] &= \mathcal{NCS}[\![n]\!] \\
\mathcal{B}[\![(\mathsf{ca}, n)]\!] &= \mathcal{NCA}[\![n]\!] \\
\mathcal{B}[\![(\mathsf{ss}, n)]\!] &= \mathcal{NSS}[\![n]\!] \\
\mathcal{B}[\![(\mathsf{sa}, n)]\!] &= \mathcal{NAS}[\![n]\!] \\
\mathcal{B}[\![(\mathsf{dv}, n)]\!] &= \mathcal{NDV}[\![n]\!] \\
\mathcal{B}[\![(\mathsf{rt}, n, t)]\!] &= \mathcal{NRS}[\![n, t]\!] \\
\mathcal{B}[\![(\mathsf{cp}, n, x, y)]\!] &= (\mathit{Null}, \emptyset, \emptyset) \\
\mathcal{B}[\![(\mathsf{sp}, n, x, y)]\!] &= (\mathit{Null}, \emptyset, \emptyset)
\end{aligned}$$

This matches up each component to a particular function depending on its tag. Complex servers, complex activities, and devices are all handled together: semantically, they are the composition of their own basic components (defined in the next section).

The simple servers and simple activities are defined in Chapter 5 by matching them to a 'code stub', which defines how that server or activity interacts with its surroundings. Routes are described in Chapter 4.

Anything to do with transputers or paths is given a 'null' entry: $(\mathit{Null}, \emptyset, \emptyset)$. The CSP process Null is defined as

$$\mathit{Null} = \mathit{Run}_{\{\}}$$

which results in a process that does not affect any other process, since it is the Run process of the null alphabet. This is used in these constructs because they only provide contextual information. Arguably, these constructs could be completely omitted from this section, but this would result in a more complicated expression for calculating the parallel composition of basic components in the next section.

3.6.2 Complex Constructs

The complex components each consist of a 'tag', a $\langle Name \rangle$, and a list of basic components. We can define the function, $\mathcal{C}$, to return the meaning of a particular complex component:

$$\mathcal{C}[\![\ C \ ]\!] = (\mathcal{C}_P[\![\ C \ ]\!], \mathcal{C}_I[\![\ C \ ]\!], \mathcal{C}_W[\![\ C \ ]\!])$$

The process associated with each complex component, $\mathcal{C}_P$, is constructed by taking the program associated with each basic component ($\mathcal{B}_P$) of that complex component, and composing them together with the CSP network parallel operator, giving the set of IDA communication events ($\mathcal{B}_I$) as the interface alphabet.

Finally, any communication events that can be hidden are hidden, because both the activity and route concerned are in the parallel construct. $\mathcal{H}$ specifies the events that can be hidden at this level: it is those events that appear in at least two distinct $\mathcal{B}_I$ sets. (Due to the way that event names are constructed and ports allocated to routes, a particular IDA event will never be in more than two $\mathcal{B}_I$ sets for a particular complex construct.[2])

$$\begin{aligned}
\mathcal{H}[\![\ C \ ]\!] &= \bigcup_{i,j \geq 3, i \neq j} \mathcal{B}_I[\![\ C_i \ ]\!] \cap \mathcal{B}_I[\![\ C_j \ ]\!] \\
\mathcal{C}_P[\![\ C \ ]\!] &= (\|_{i \geq 3}(\mathcal{B}_P[\![\ C_i \ ]\!], \mathcal{B}_I[\![\ C_i \ ]\!])) \setminus \mathcal{H}[\![\ C \ ]\!]
\end{aligned}$$

(The '≥ 3' indices select the basic components out of the complex construct. The first element is the tag, and the second element is the $\langle Name \rangle$.)

The set of events that the particular complex component generates as its own IDA interface are those which are not bound in the previous expression: *i.e.* it is the union of its own basic component IDA interfaces less

[2] The name of every route is unique. The path resolution algorithm ensures that a port is connected to exactly one window and *vice versa*. Therefore, the combination name of route-window appears in exactly two basic components, the route which owns the window, and the server or activity connected to it.

those in $\mathcal{H}$.

$$\mathcal{C}_I[\![\, C \,]\!] \quad = \quad \left(\bigcup_{i \geq 3} \mathcal{B}_I[\![\, C_i \,]\!] \right) \setminus \mathcal{H}[\![\, C \,]\!]$$

The work events for C are simply the union of all its basic components' work events. Thus the work events are collected together and propagated upwards until they are composed with a scheduler process later in this chapter.

$$\mathcal{C}_W[\![\, C \,]\!] \quad = \quad \bigcup_{i \geq 3} \mathcal{B}_W[\![\, C_i \,]\!]$$

Any complex components referred to in a particular complex component are included by the use of $\mathcal{B}$. This results in a structure for the semantics that closely mirrors the hierarchical structure of the design.

We can now define the three functions, $\mathcal{NCS}$, $\mathcal{NCA}$, $\mathcal{NDV}$, referred to in the previous section.

$$\begin{array}{rcll}
\mathcal{NCS}[\![\, n \,]\!] & = & \mathcal{C}[\![\, \mathcal{CS}_i \,]\!] & \text{iff } \mathcal{CS}_{i,2} = n \\
\mathcal{NCA}[\![\, n \,]\!] & = & \mathcal{C}[\![\, \mathcal{CA}_i \,]\!] & \text{iff } \mathcal{CA}_{i,2} = n \\
\mathcal{NDV}[\![\, n \,]\!] & = & \mathcal{C}[\![\, \mathcal{DV}_i \,]\!] & \text{iff } \mathcal{DV}_{i,2} = n
\end{array}$$

The conditional expression matches the name of the complex construct, *e.g.* $\mathcal{CS}_{i,2}$, to the name required, *i.e.* n, of that type of complex construct.

By static constraint 2 (in Appendix A.1), there will be at most one matching complex construct. Static constraints 4 to 6 ensure that there will be at least one matching construct.

3.6.3 Semantics of the System

We can now define the meaning of the system, $\mathcal{M}[\![\, \mathcal{S} \,]\!]$:

$$\mathcal{M}[\![\, \mathcal{S} \,]\!] \quad = \quad \left(\mathcal{C}_P[\![\, \mathcal{SY} \,]\!] \|_{\mathcal{C}_W[\![\, \mathcal{SY} \,]\!]} \mathit{Scheduler} \right) \setminus \mathcal{C}_W[\![\, \mathcal{SY} \,]\!]$$

This definition effectively 'caps' the complex component meaning for the system. The purpose of *Scheduler* is to restrict when events in the set of processing events, $\mathcal{C}_W[\![\, \mathcal{SY} \,]\!]$, may occur. The intention here is that

$Scheduler$ can simulate the behaviour of the scheduler on the final hardware: with limited processing resources, the individual components will not progress simultaneously, instead being allocated resources by the scheduler. Since this could affect the timing of outputs, we need to have the framework to model this action.

Finally, the processing events are hidden from the environment (otherwise the environment could interfere with the events by refusing to engage in them). This means that the only DORIS events visible are unresolved IDA events, and extra events defined in activities and servers.

The definition of $Scheduler$ will be given in Section 5.6 (Page 124).

3.7 Example

Recall the example in Figure 3.2 on Page 59. By simply working through the previous definitions, we can see that

$$\mathcal{M}[\![\mathcal{S}]\!] = \big(\mathcal{C}_P[\![\mathcal{SY}]\!] \|_{\mathcal{C}_W[\![\mathcal{SY}]\!]} Scheduler\big) \setminus \mathcal{C}_W[\![\mathcal{SY}]\!]$$

$$\begin{aligned}
\mathcal{C}_P[\![\mathcal{SY}]\!] &= \| \; \{ \; (\mathcal{B}_P[\![(\textsf{sa}, \textsf{Process\_A})]\!], \\
&\qquad\quad \mathcal{B}_I[\![(\textsf{sa}, \textsf{Process\_A})]\!]), \\
&\qquad\quad (\mathcal{B}_P[\![(\textsf{sa}, \textsf{Process\_B})]\!], \\
&\qquad\quad \mathcal{B}_I[\![(\textsf{sa}, \textsf{Process\_B})]\!]), \\
&\qquad\quad (\mathcal{B}_P[\![(\textsf{rt}, \textsf{Data}, \textsf{Pool})]\!], \\
&\qquad\quad \mathcal{B}_I[\![(\textsf{rt}, \textsf{Data}, \textsf{Pool})]\!]), \\
&\qquad\quad (\mathcal{B}_P[\![(\textsf{sp}, , \textsf{Process\_A:p1}, \textsf{Data:w1})]\!], \\
&\qquad\quad \mathcal{B}_I[\![(\textsf{sp}, , \textsf{Process\_A:p1}, \textsf{Data:w1})]\!]), \\
&\qquad\quad (\mathcal{B}_P[\![(\textsf{sp}, , \textsf{Data:w2}, \textsf{Process\_B:p1})]\!], \\
&\qquad\quad \mathcal{B}_I[\![(\textsf{sp}, , \textsf{Data:w2}, \textsf{Process\_B:p1})]\!]) \\
&\qquad \} \; \setminus \mathcal{H}[\![\mathcal{SY}]\!] \\
&= \| \; \{ \; (\mathcal{NAS}_P[\![\textsf{Process\_A}]\!], \\
&\qquad\quad \mathcal{NAS}_I[\![\textsf{Process\_A}]\!]), \\
&\qquad\quad (\mathcal{NAS}_P[\![\textsf{Process\_B}]\!], \\
&\qquad\quad \mathcal{NAS}_I[\![\textsf{Process\_B}]\!]), \\
&\qquad\quad (\mathcal{NRS}_P[\![\textsf{Data}, \textsf{Pool}]\!], \\
&\qquad\quad \mathcal{NRS}_I[\![\textsf{Data}, \textsf{Pool}]\!]), \\
&\qquad\quad (Null, \emptyset) \\
&\qquad\quad (Null, \emptyset) \quad \} \setminus \mathcal{H}[\![\mathcal{SY}]\!]
\end{aligned}$$

$$\begin{aligned} &= \;\; \| \; \{ \; (\mathcal{NAS}_P[\![\, \textsf{Process\_A} \,]\!], \\ &\qquad\quad \mathcal{NAS}_I[\![\, \textsf{Process\_A} \,]\!]), \\ &\qquad\quad (\mathcal{NAS}_P[\![\, \textsf{Process\_B} \,]\!], \\ &\qquad\quad \mathcal{NAS}_I[\![\, \textsf{Process\_B} \,]\!]), \\ &\qquad\quad (\mathcal{NRS}_P[\![\, \textsf{Data, Pool} \,]\!], \\ &\qquad\quad \mathcal{NRS}_I[\![\, \textsf{Data, Pool} \,]\!]), \\ &\qquad \} \; \backslash \mathcal{H}[\![\, \mathcal{SY} \,]\!] \end{aligned}$$

$$\begin{aligned} \mathcal{C}_I[\![\, \mathcal{SY} \,]\!] &= \left(\bigcup_{i \geq 3} \mathcal{B}_I[\![\, \mathcal{SY}_i \,]\!] \right) \backslash \mathcal{H}[\![\, \mathcal{SY} \,]\!] \\ &= (\quad \mathcal{NAS}_I[\![\, \textsf{Process\_A} \,]\!] \\ &\quad \cup \; \mathcal{NAS}_I[\![\, \textsf{Process\_B} \,]\!] \\ &\quad \cup \; \mathcal{NRS}_I[\![\, \textsf{Data, Pool} \,]\!] \quad) \\ &\quad \backslash \mathcal{H}[\![\, \mathcal{SY} \,]\!] \end{aligned}$$

$$\begin{aligned} \mathcal{C}_W[\![\, \mathcal{SY} \,]\!] &= \bigcup_{i \geq 3} \mathcal{B}_W[\![\, \mathcal{SY}_i \,]\!] \\ &= \quad \mathcal{NAS}_W[\![\, \textsf{Process\_A} \,]\!] \\ &\quad \cup \; \mathcal{NAS}_W[\![\, \textsf{Process\_B} \,]\!] \\ &\quad \cup \; \mathcal{NRS}_W[\![\, \textsf{Data, Pool} \,]\!] \end{aligned}$$

This has simply rewritten the definitions into other expressions: we will return to this example later in the thesis to illustrate how these definitions eventually resolve into concrete statements.

3.8 Contextual Information

A considerable amount of contextual information is used in these semantics — although so far, we have given none of it explicitly.

There are two parts: the transputer-scheduler part, and the path-interface part. Firstly, (in the distributed layer) we can use the information in the transputer link components to determine which route is allocated to which transputer. Further, each simple activity and server resides on exactly one transputer. This information can then be used for scheduler modelling. (A further note on the use of transputer information is given in Section 3.8.2, Page 74.) Secondly, the paths determine which port connects to which window.

We consider that the semantic functions implicitly carry the appropriate contextual information with them. This has been omitted to prevent cluttering the previous definitions. However, access is required to this information in the next two chapters.

It is useful in the next two chapters to define the function $\mathcal{F}$ to return a (fully-qualified) unique name (FQUN) for a particular instance of a construct. For example, the two instances of Process_A in the example in Figure 3.3 (Page 60) above may be given the FQUNs

Complex_Controller.Alpha_A_Client.Process_A
Complex_Controller.Bravo_A_Client.Process_A

Further information for the definition of $\mathcal{NSS}$ and $\mathcal{NAS}$ is given by the paths and routes. The paths define the ports and windows in the system. When each path is traced through the simple and complex paths in the network, then it can be replaced by a simple path from a path to a window (or from a window to a path).

Ultimately, the references to ports within any simple server or activity are replaced by the event within the route or server at the other end of the path. This exploits the CSP event synchronizations: the components at each end of the path must agree on that event occurring.

We define a function, $\mathcal{R}$, which for a port of a server or activity, returns the FQUN of the associated route (or server) and window number. This, like $\mathcal{F}$, is used in the next two chapters.

The next section describes path and interface resolution in further detail.

3.8.1 Path Resolution

In the previous section, we identified a function, $\mathcal{R}$, which hides the detail of resolving which servers, activities, and routes are connected. Instead, $\mathcal{R}$ simply identifies which window and which route or server that a particular port is connected to. This section briefly describes the rules used for determining that information — Appendix A.2 (Page 216) gives further details of the algorithm for path resolution.

Figures 3.4 and 3.5 (Pages 76 and 77) can be used to illustrate the problem. The simple interfaces defined by these paths (including the full hierarchical names) are:

A.B.C:w1	A.B.C:p1	A.G.H.I:p1
A.D.F:w1	A.D.F:w2	A.D.E:p1

These are resolved thus:

A.D.F:w2	$\rightarrow$	A.D.E:p1	because fe links them directly
A.B.C:p1	$\rightarrow$	A.D.F:w1	via cb, bd, and df
A.G.H.I:p1	$\rightarrow$	A.B.C:w1	via ih, hg, gb, and bc on complex channel a

Appendix A.2 repeats this example with further details.

3.8.2 Allocation to Transputers and Links

Although not explicitly included in the syntax, the mechanical implementation of the syntax includes a field for indicating which transputer a particular component resides on.

The rule in this case is that if a transputer is indicated, then the component resides on that transputer. Otherwise (*i.e.* if the field is empty), then the component resides on the same transputer as its parent. It is an error (at the distribution level) if a simple server, simple activity or route does not have a transputer indicated in this way.

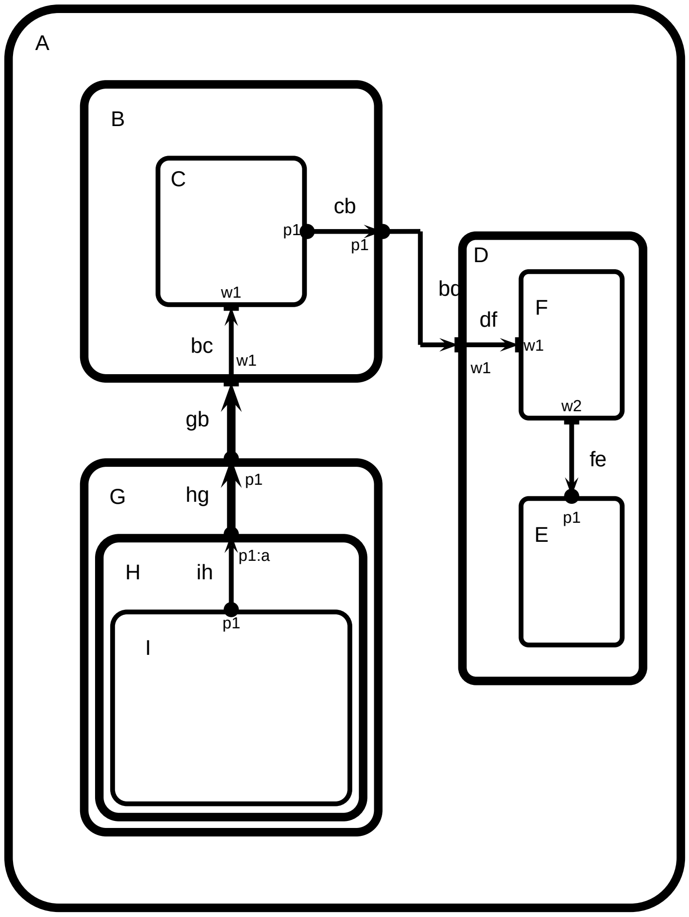
Ideally, this information (along with the rest of the transputer information) should be kept separate from the design level information. Other information pertaining to scheduling can also be stored here, *e.g.* activity priorities.

3.9 Refinement

Refinement[3] within this framework is easy to describe. The functional level requirements can be defined as timed safety and liveness predicates using the TCSP notation (see Chapter 8 for examples from the case studies). The TCSP meaning of the design and distribution levels has been given above. Finally, the meaning of the execution layer is given by the parallel composition of the distribution level meaning, and the scheduler (see Section 5.6).

[3]This section is an aside. It is not used in the rest of this thesis, but is relevant for future work and industrial use.

Since this means that all four layers are defined within the framework of Timed CSP, then the Timed CSP notion of refinement can be used. Further details on Timed CSP refinement can be found in the considerable volume of literature on Timed CSP (see Section 2.4.4, Page 41).



Described on Page 73

Textual version in Figure 3.5 (Page 77)

Figure 3.4: Illustration of path resolution algorithm

```
(       (sy, A,
                (cs, B),
                (cs, D),
                (ca, G),
                (cp, gb, G:p1, B:w1),
                (sp, bd, B:p1, D:w1))
        (cs, B,
                (ss, C),
                (sp, bc, w1:a, C:w1),
                (sp, cb, C:p1, p1))
        (cs, D,
                (sa, E),
                (rt, F, Channel),
                (sp, df, w1, F:w1),
                (sp, fe, F:w2, E:p1))
        (ca, G,
                (ca, H),
                (cp, hg, H:p1, p1))
        (ca, H,
                (sa, I),
                (sp, ih, I:p1, p1:a))       )
```

Code stubs expected are: simple server C
simple activity E
simple activity I
route type Channel

Described on Page 73

Graphical version in Figure 3.4 (Page 76)

Figure 3.5: Textual statement of path resolution example

Chapter 4

Specification of IDAs

4.1 Overview

The previous chapter identified and described the overall structure of the notation and semantics for describing a DORIS system. The next chapter will describe the activities within a DORIS system. This leaves one major part: the IDAs.

In this chapter, we will describe how to

- specify IDAs;
- compose IDAs; and
- implement IDAs.

We can then give the definition of the semantic function $\mathcal{NRS}$. In Chapter 7, some assertions about IDAs will be tested using techniques introduced in Chapter 6.

4.1.1 General Model

Although MASCOT-3 has a notion of general 'access interfaces' (multiple procedure-like interfaces), we adopt a simpler one reader-one writer model in this thesis.

The simplification eases the basic modelling at the expense of requiring more work to simulate access interfaces in this framework. The access interfaces can be simulated by constructing devices (or even complex servers if the object is particularly complex) to abstract away from the one reader-one writer IDAs.

The CSP model is that each reader engages in one or more events in a given order; the final event returns the value read. The writer engages in one or more events in a given order; all the writer events carry the value written. (It is easier to include the value written all the time, rather than try to code up specification predicates to capture the value at a later stage.)

4.1.2 Distribution Issues

The distribution level of a DORIS design includes the allocation of activities to individual processors within a transputer network. The 'processes' implementing the activities are then distributed according to this scheme:

- IDAs written to and read from by activities allocated to the same processor are known as *private* and are allocated to that same processor. This is the simplest form.

- When the reading and writing activities are allocated to two different processors that are directly linked, the IDA is known as a *shared* IDA. It is allocated to the processor where the reading activity resides, and an extra 'process' added on the writing processor to arrange for writes to be forwarded across the link.

- The final case occurs when there is no direct link between the reading and writing processors: this results in a *remote* IDA. Again, the IDA is allocated to the reading processor, and forwarding processes are placed on two or more processors. (In this thesis, we will only come across the case where two forwarding processes are required. The theory described in Section 4.9, Page 103, is easily extended to more than two forwarding processors.)

This arrangement is described in more detail for the system modelled in the large case study (Section 8.5) in Grigg's technical report [34]. The forwarding processes used in this thesis are described in Section 4.9.

4.2 Types of Shared Variable

Two broad classes of shared variable concern us. Firstly, Lamport's classification of shared variables into *safe*, *regular*, and *atomic* is mathematically elegant, and can be used to construct larger shared variables [59]. Secondly, Simpson's definitions of routes are a more industrially motivated concept [112].

4.2.1 Lamport's Variables

It is particularly well-known in database systems that many operations appear to be atomic, *i.e.* their interaction with the environment can be regarded as a single point in time. However, such operations really consist of several atomic operations, *e.g.* the transfer of funds from one bank account to another; this involves the debit of one account, and the credit of another.

Lamport defines three classes of shared variable [59]: safe, regular, and atomic. All will return the 'correct' value held in the register if no writes overlap with the action of reading. Here, the correct value is the most recently written value. The difference between them is the value returned when there is a concurrent write:

- A safe register will return any type-correct value.
- A regular register will return the value held before any concurrent writes, or the value written by any concurrent write.
- An atomic register behaves as if all reads and writes occur in some definite, non-overlapping order.

The example on Page 20 (and Figure 5) of Lamport's 'On Interprocess Communication' illustrates these three types [59], and is repeated in Figure 4.1.

- If the variable is safe, then Read 1 will get the value 5. Reads 2 and 3 will nondeterministically get any possible value that the variable can hold.
- A regular variable will get the value 5 for Read 1. Reads 2 and 3 may obtain either 5 or 6.

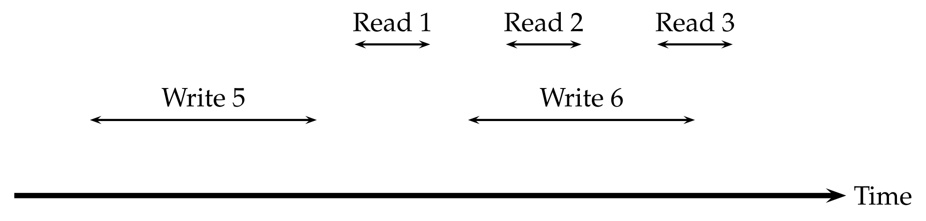


Figure 4.1: Lamport's shared variables (Figure 5 [59])

- An atomic variable will again get value 5 for Read 1, and the other two reads may read any of the following pairs:

Read 2	Read 3
5	5
5	6
6	6

  but not the pair 6 then 5.

Lamport then continues to classify all single-writer variables as having three 'coordinates':

- safe, regular, or atomic;
- boolean or multi-valued; and
- single-reader or multi-reader.

Further, Lamport asserts that the weakest type of variable, the safe boolean single-reader variable, is easily implementable in hardware. He then notes that all the other classes can be constructed from weaker classes, with the exception of atomic multi-reader variables. (Lamport comments that requiring mutual exclusion for some operations will, at some lower level, require the problem of concurrent access to be addressed.)

Later in this chapter, we will implement one of Simpson's routes with Lamport's variables.

4.2.2 Simpson's Variables

Simpson has defined a number of *protocols* for engineering purposes. For instance, the 'pool' is an IDA where the writer can always write, and the reader can always read. Conversely, both the reader or writer can be held up when accessing a channel (respectively when there is no data to read, and when there is no space in the channel). Figure 4.2 sets out the various protocols and their symbols as defined by Simpson [112]. The basic protocols can be given characteristics as in Figure 4.3. Simpson identifies these four protocols with four 'kinds' of data that may be passed in a real-time system:

Event data is passed by a *signal*: data is overwritten if it isn't first read by the reader.

Reference data is written to a *pool*, and may be continuously overwritten while the reader continuously reads. (This requires special mechanisms to prevent timing interference — see the paper by Brooke *et al.* [9].)

Message data is passed in a *channel*: this is the classic FIFO.

Configuration data is *constant* in the system. This is never written to (or if it is, then it never affects the reader).

Each of the four basic protocols only ever 'holds' one datum.

'Flash data' and 'rendezvous' are the two interlocked protocols, which are variants of the signal and channel respectively. They require closer synchronization of the reader and writer: they hold 'zero' data in storage. Similarly, queued protocols can hold more than one datum: the number that may be held is given by the number n on the diagram. These definitions will be discussed later in this chapter.

Void value protocols are derived from earlier protocols: they serve only to pass a 'null' message. Note that in the large case study, we ignore the values passed regardless of protocol.

The integral response and separate response protocols are really two instances of (derivatives of) channels connected together, and are not discussed further in this thesis.

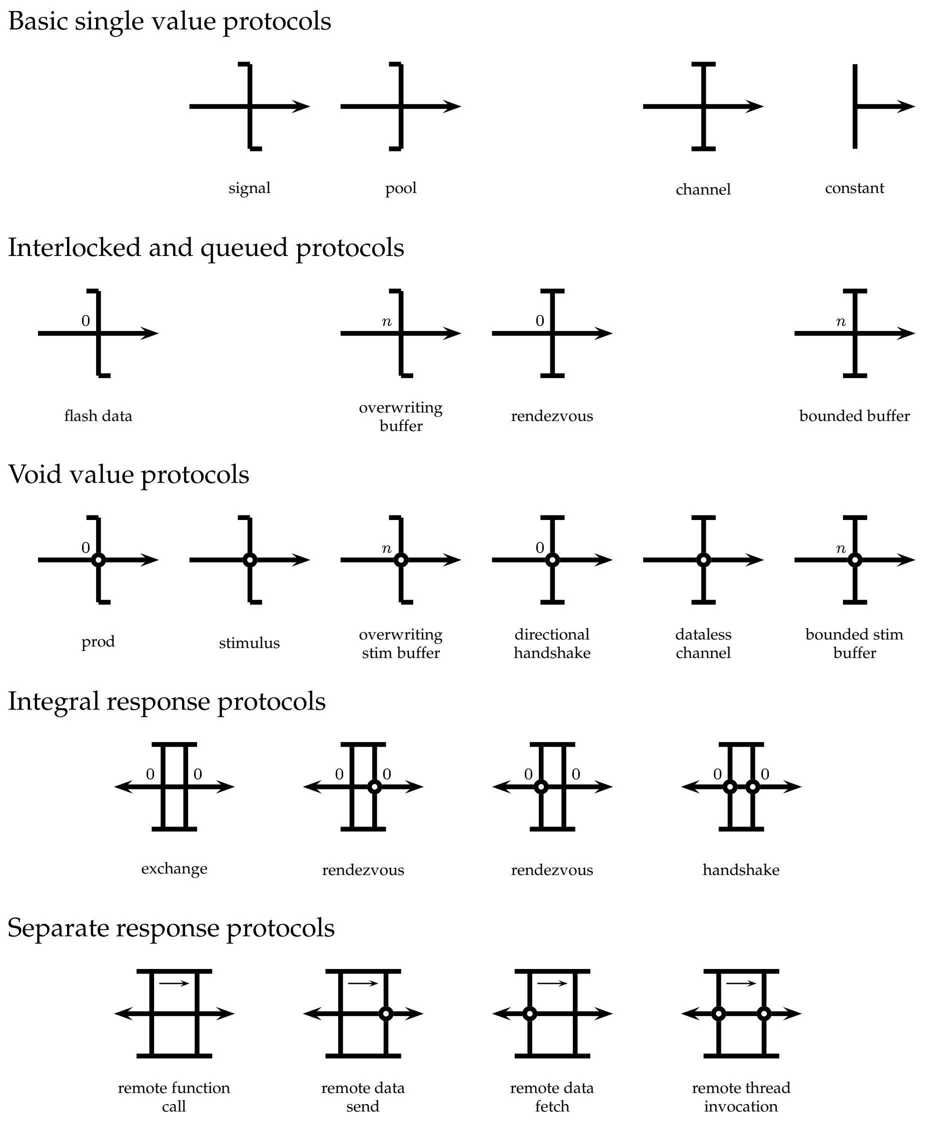


Figure 4.2: DORIS protocol symbols

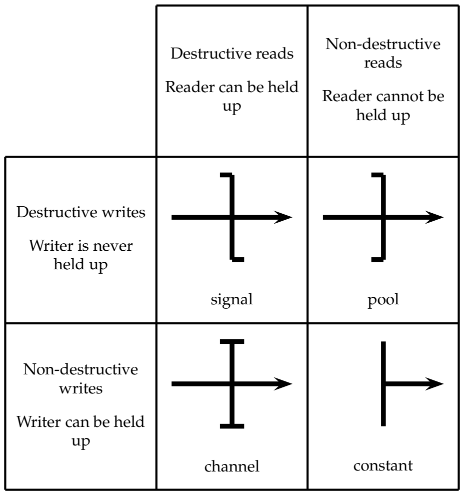


Figure 4.3: DORIS basic protocols

4.3 Untimed Two-Point Models

We now examine how to model the protocols used in DORIS. First, an untimed 'two-point' model and the problem with it is given. In the next section, possible solutions are enumerated.

A CSP process interacts with its environment via atomic events. These events identify points in time. To model an interval, we use two events: the start and end points of the interaction (hence the name 'two-point').

	Reader	Writer
Start	sr	sw
End	er	ew

Thus, an activity reading from a route of FQUN R engages in the two events $R.sr$, and $R.er$ in that order. Similarly, the writer engages in $R.sw$, and $R.ew$ in that order. (In the sequel, we will assume that IDA events are constrained to occur in the specified order.) Three of these events (er, sw, and ew) are really channels,[1] although this is often ignored when only the

[1]Recall from Page 36 that channels allow data to be communicated in the form of CSP

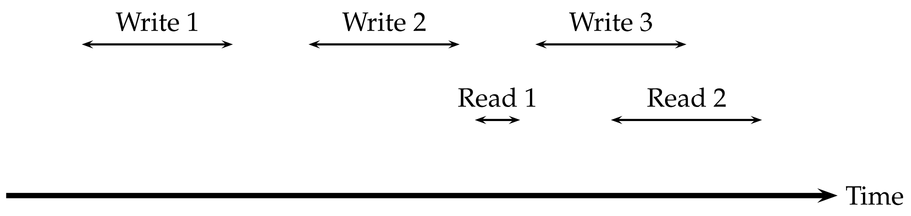


Figure 4.4: Fresh data: short reads

synchronization matters.

We would then expect the following trace to be typical of a variable (say, R):

$$\langle R.sw.5 \rightarrow R.ew.5 \rightarrow R.sr \rightarrow R.er.5\rangle$$

i.e. the value '5' is written to (stored in) the variable; it is then read out.

In the paper, *An Analysis of the Four-slot Mechanism* [9], an implementation of one of the most interesting routes, the pool, is examined. This mechanism is called the 'four-slot mechanism' because it uses four data 'slots' to store data passing through the mechanism while allowing both the reader and writer to work without timing constraints [110].

Essentially, this implementation uses some boolean-valued regular variables to direct the reader and writer to different slots within the mechanism, so that a corrupt datum can never be returned to the reader. However, it is then necessary to prove that the data returned is 'fresh'. This leads to the philosophical (and necessary) point of how to identify fresh data. Consider Figures 4.4 and 4.5. There are three reads: what value does each obtain?

- Any variable should return the value of Write 2 for Read 1.

- Read 2 is less obvious: A Lamport-safe variable would return any possible value for that type of variable; a regular variable would return the value of Write 2 or Write 3.

compound events.

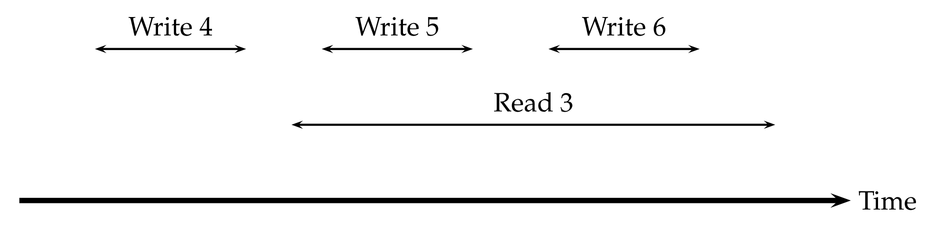


Figure 4.5: Fresh data: a long read

- Read 3 is similar to Read 2. But what if Read 3 was taking a very long time? Should it really return the value of Write 4? Surely we should call such a value 'stale' (*i.e.* not fresh)?

Abstractly, there is no problem with returning stale values, since non-determinism can be very helpful in specifying a system without giving too much detail too early.

In 'real' systems, however, especially real-time systems (which is the prime domain for DORIS), this approach is not acceptable. Data that is 'too stale' could adversely affect the performance of a system.

The work described in the four-slot analysis paper uses a two-point model in untimed failures CSP,[2] and concluded that the four-slot mechanism does not satisfy the formal formulation of freshness given in the paper.

4.4 Fixing the Two-Point Pool

The conclusion above is not satisfactory: the counter-example traces given in the paper were in some ways absurd, since they relied on very long periods of work by (say) the writer without the reader engaging in a single event. This was partially due to the particular semantic model chosen, but mostly due to the formulation of freshness given under that model.

[2]Untimed failures CSP has a sequential notion of time, *i.e.* event a happened before event b, but it doesn't say how long the interval was. The word 'failures' refers to the model being able to express both safety and readiness properties.

In order to more satisfactorily model shared variables such as the four-slot implementation of the pool we could

- use real-time (*i.e.* identify the time that events happen, and not just the order that they occur in); or
- use more points in the model, say, a three-point model.

The first case is the most attractive at first impression: however, it could interact poorly with scheduling. Consider a definition of a variable which says, 'The value read will have been written within the last twenty seconds if there is one, otherwise it will be the last value written', and a scheduler which regularly deschedules the writer (and hence the value written) for thirty seconds at a time.

Of course, a method of counting the 'scheduled time' (like the scheduler in Section 5.6) could be used, but this so complicates accounting which activities and routes have used time that it is simpler (and more elegant) to consider the second case. Further, we would like to consider time in as few places as possible in the semantics, to aid analysis. Therefore the best place for timing is inside the activities.

The second case could be called the 'inappropriate abstraction' argument: we could say that we have missed the essential point of the definition of the pool (and also of the four-slot mechanism), and that there is some middle event that marks a change of phase between the start and end points. It is this case that we develop.

So we then identify six events associated with a particular route of FQUN R: $R.sr$, $R.mr$, $R.er$, $R.sw$, $R.mw$ and $R.ew$. Again, some of these events are really channels (er, sw, mw and ew).

	Reader	Writer
Start	sr	sw
Middle	mr	mw
End	er	ew

The middle event can be identified with different types of 'event' within different IDAs. It simply gives an external synchronization. Generally, a writer takes the steps

$$sw \rightarrow \text{write} \rightarrow mw \rightarrow \text{indicate} \rightarrow ew$$

whereas the reader follows

$$sr \rightarrow \text{indicate} \rightarrow mr \rightarrow \text{read} \rightarrow er$$

'Indication' is the process where, in the implementation, internal variables are updated (which can, for example, be used as 'advisory locks'). Again, the IDA events should be constrained to occur in the given order.

4.5 Three-Point IDA Specifications

We now give the three-point specifications for each of the constant,[3] channel, signal and pool. These specifications will take the form of a number of conjoined predicates over failures.

Note that these are untimed failures predicates: Section 4.11 addresses the issue of timed failure specifications.

4.5.1 Preliminaries

Suppose that the (nonempty) set of possible data values of the variable is D, and that a distinguished value $d_0 \in D$ is the initial value (if appropriate for the particular IDA).

We define several sets to group the events:

$$\begin{aligned}
ER(D) &= \{d : D \bullet er.d\} \\
SW(D) &= \{d : D \bullet sw.d\} \\
MW(D) &= \{d : D \bullet mw.d\} \\
EW(D) &= \{d : D \bullet ew.d\} \\
\\
\alpha R(D) &= \{sr, mr\} \cup ER(D) \\
\alpha W(D) &= SW(D) \cup MW(D) \cup EW(D) \\
\\
\alpha IDA(D) &= \alpha R(D) \cup \alpha W(D)
\end{aligned}$$

The first four sets refer to the events that carry data values. The fifth and sixth sets are the events for readers and writers respectively. The final set is the alphabet of any IDA with data values D.

[3] This is useful in real systems: it can be viewed as static configuration data.

We now define some simple predicates defining the state of the reader and writer. In these predicates, 'tr' is a free variable representing a trace of a process. '$s\#A$', where s is a trace and A is a set of events, means 'count the number of occurrences of events in A in s'. A single event version, '$s\#a$' can be defined

$$s\#a = s\#\{a\}$$

The first predicates constrain the reader to be in exactly one of three states:

$$\begin{aligned}
\mathit{Reading}_A(D) &= (tr\#sr - tr\#mr = 1) \wedge (tr\#mr = tr\#ER(D))\\
\mathit{Reading}_B(D) &= (tr\#sr = tr\#mr) \wedge (tr\#mr - tr\#ER(D) = 1)\\
\mathit{NotReading}(D) &= (tr\#sr = tr\#mr = tr\#ER(D))\\
\mathit{ReaderSequence}(D) &= \mathit{NotReading}(D) \vee \mathit{Reading}_A(D) \vee \mathit{Reading}_B(D)
\end{aligned}$$

This constrains the reader so that there can only be one read in progress. For example, the reader can be either not reading ($\mathit{NotReading}$), or a start-read has occurred ($\mathit{Reading}_A$), or a middle-read has occurred ($\mathit{Reading}_B$).

The next predicates perform a similar function for the writer.

$$\begin{aligned}
\mathit{Writing}_A(D) &= \quad (tr\#SW(D) - tr\#MW(D) = 1)\\
&\quad \wedge (tr\#MW(D) = tr\#EW(D))\\
\mathit{Writing}_B(D) &= \quad (tr\#SW(D) = tr\#MW(D))\\
&\quad \wedge (tr\#MW(D) - tr\#EW(D) = 1)\\
\mathit{NotWriting}(D) &= (tr\#SW(D) = tr\#MW(D) = tr\#EW(D))\\
\mathit{WriterSequence}(D) &= \mathit{NotWriting}(D) \vee \mathit{Writing}_A(D) \vee \mathit{Writing}_B(D)
\end{aligned}$$

The writer is further constrained by $\mathit{ConsistentWrite}$ so that the value written is the same at each step of the same write. s is a pattern-matching trace: for example, we write '$tr = s^\frown\langle ew.d\rangle$' rather than further clutter the statements by writing expressions like '$\exists s, d \bullet tr = s^\frown\langle ew.d\rangle$'. (We will use this approach often in the thesis.)

$$\begin{aligned}
\mathit{ConsistentWrite}(D) &= \quad (tr \downarrow \alpha W(D) = s^\frown\langle sw.d_1, mw.d_2\rangle \Rightarrow d_1 = d_2)\\
&\quad \wedge (tr \downarrow \alpha W(D) = s^\frown\langle mw.d_2, ew.d_3\rangle \Rightarrow d_2 = d_3)
\end{aligned}$$

The '$\downarrow$' symbol is the restriction operator. $s \downarrow A$ returns the trace s with only the events in A: all other events are discarded.

All IDAs have these basic properties, so they are combined into one definition:

$$\begin{aligned} \mathit{Basic}(D) \quad = \quad & \mathit{ReaderSequence}(D) \\ & \wedge\ \mathit{WriterSequence}(D) \\ & \wedge\ \mathit{ConsistentWrite}(D) \end{aligned}$$

Sometimes, we require that one or more events cannot be refused. In the following predicates, we introduce rf, the set of refusals for the process. This is always a free variable (like tr). $last(s)$ returns the last event in a non-empty trace s — therefore, we have to be careful to only apply $last$ to non-empty traces.

The first six predicates are of the form 'if the last event is this event's predecessor, then this event cannot be refused'.

$$\begin{aligned} \mathit{May}_{sr}(D) \quad = \quad & \quad (tr \downarrow \alpha R(D) = \langle\rangle \vee last(tr \downarrow \alpha R(D)) = er) \\ & \Rightarrow\ sr \notin rf \\ \mathit{May}_{mr}(D) \quad = \quad & (last(tr \downarrow \alpha R(D)) = sr) \Rightarrow mr \notin rf \\ \mathit{May}_{er}(D) \quad = \quad & (last(tr \downarrow \alpha R(D)) = mr) \Rightarrow ER(D) \setminus rf \neq \emptyset \end{aligned}$$

$$\begin{aligned} \mathit{May}_{sw}(D) \quad = \quad & \quad (tr \downarrow \alpha W(D) = \langle\rangle \vee last(tr \downarrow \alpha W(D)) = ew) \\ & \Rightarrow\ SW(D) \cap rf = \emptyset \\ \mathit{May}_{mw}(D) \quad = \quad & (last(tr \downarrow \alpha W(D)) = sw.d) \Rightarrow mw.d \notin rf \\ \mathit{May}_{ew}(D) \quad = \quad & (last(tr \downarrow \alpha W(D)) = mw.d) \Rightarrow ew.d \notin rf \end{aligned}$$

The first two are simple: these unadorned events cannot be refused. The third states that there is at least one value that can be returned by the end-read event. The start-write term states that no start-write event can be refused (since the variable cannot dictate that a certain value may not be written). The last two predicates state that the middle-write or end-write event of the appropriate value cannot be refused.

Again, these are combined into two larger predicates ($\mathit{MayRead}$ and $\mathit{MayWrite}$).

$$\begin{aligned} \mathit{MayRead}(D) \quad = \quad & \mathit{May}_{sr}(D) \wedge \mathit{May}_{mr}(D) \wedge \mathit{May}_{er}(D) \\ \mathit{MayWrite}(D) \quad = \quad & \mathit{May}_{sw}(D) \wedge \mathit{May}_{mw}(D) \wedge \mathit{May}_{ew}(D) \end{aligned}$$

For some IDAs, further conditions apply to when an event can occur. In this thesis, only the mr and ew events are constrained in this way. In the following two predicates, c is a predicate with tr free. Note that some further restrictions apply to c: the occurence of the constrained event should not cause c to become false.

$$\begin{aligned} MayBlock_{mr}(D, c) &= \quad (c \wedge last(tr \downarrow \alpha R(D)) = sr) \Rightarrow mr \notin rf \\ &\quad \wedge (tr = s^\frown \langle mr \rangle) \Rightarrow c \\ MayBlock_{ew}(D, c) &= \quad (c \wedge last(tr \downarrow \alpha W(D)) = mw.d) \Rightarrow ew.d \notin rf \\ &\quad \wedge (tr = s^\frown \langle ew.d \rangle) \Rightarrow c \end{aligned}$$

The interpretation of these predicates is that the event concerned should only occur when the predicate c is true. Moreover, if c is true, then the event should not be refused (provided it is the next event in the sequence).

4.5.2 Constant

We are now in a position to define the $Constant$ IDA. A constant is a route which always returns the value d_0, and is always prepared to allow the reader to read.

$$\begin{aligned} Constant(D, d_0) &= \quad Basic(D) \\ &\quad \wedge MayRead(D) \\ &\quad \wedge ConstantValue(d_0) \end{aligned}$$

where

$$ConstantValue(d_0) = (last(tr) = er.d) \Rightarrow d = d_0$$

This says nothing about what a writer can or cannot do: it simply implies that the writer cannot affect the reader.

4.5.3 Pool

The pool is a particularly interesting IDA, since the definition of which value should be returned is the motivation behind using three-point definitions, and not two-point definitions.

A pool allows both the reader and writer to always read and write. So we can write:

$$\begin{aligned} Pool(D, d_0) \quad = \quad & Basic(D) \\ & \wedge\ MayRead(D) \\ & \wedge\ MayWrite(D) \\ & \wedge\ PoolValue(D, d_0) \end{aligned}$$

It means that as well as having all the common properties of an IDA ($Basic$), neither the reader nor the writer is ever blocked. $PoolValue$ identifies which values may be returned by an end-read. The value returned is any of the following:

- the value of the last ew before the sr if there is a complete write preceding the sr;
- d_0 if there is no complete write preceding the sr; or
- the value of any mw between the last ew before the sr, and the mr.

The first two cases cover the obvious possible value: the last completed write before the read. The last case allows the implementation some slack in the values that may be returned: the intention is that this is used to allow the readers and writers to continue with less mutual interference.

$PoolValue$, in turn uses $PoolValues$, which returns a set of values representing these legal values. The first half of the union ($PoolValuesSR$) captures those events from before the start-read, and the second half ($PoolValuesSMR$) handles events between the start-read and the middle-read.

$$\begin{aligned} PoolValue(D, d_0) \quad = \quad & last(tr) = er.d \\ & \Rightarrow\ d \in PoolValues(D, d_0, tr) \end{aligned}$$

$$\begin{aligned} PoolValues(D, d_0, s) \quad = \quad & PoolValuesSR(D, d_0, s) \\ & \cup\ PoolValuesSMR(D, d_0, s) \end{aligned}$$

$$PoolValuesSR(D, d_0, s) \quad = \quad \{d_1, d_2\}$$

$$\begin{array}{lll} & \text{where} & \mathit{last}(\langle mw.d_0\rangle^\frown \mathit{PoolSliceSR}(s) \downarrow MW(D)) \\ & & \quad = mw.d_1 \\ & & \wedge \quad \mathit{last}(\langle ew.d_0\rangle^\frown \mathit{PoolSliceSR}(s) \downarrow EW(D)) \\ & & \quad = ew.d_2 \end{array}$$

$$\begin{array}{lll} \mathit{PoolValuesSMR}(D, d_0, s) & = & \{d \mid \\ & & mw.d \in \sigma(\mathit{PoolSliceSMR}(s) \downarrow MW(D))\} \end{array}$$

In the expressions above, 'σ' means 'set of': it turns a trace into a set of the values appearing in that trace.

The two final expressions are functions that take a trace, and return a trace. They 'slice' up the trace concerned to return the interesting portions of the trace to be manipulated further (above).

$$\begin{array}{lll} \mathit{PoolSliceSR}(s) & = & s' \\ \text{where} & & s = s'^\frown\langle sr\rangle^\frown s_1 \\ & & \wedge\ sr \notin \sigma(s_1) \\ \\ \mathit{PoolSliceSMR}(s) & = & s'' \\ \text{where} & & s = s_2{}^\frown\langle sr\rangle^\frown s''^\frown\langle mr\rangle^\frown s_3 \\ & & \wedge\ sr \notin \sigma(s''^\frown s_3) \\ & & \wedge\ mr \notin \sigma(s_3) \end{array}$$

Note that these expressions are only partially defined: there are traces where they are not defined.

Although the internal choice operator ($\sqcap$) does not appear explicitly (this is a predicate rather than a program), the specification uses nondeterminism to represent that as long as at least one 'legal' value is offered for an end-read, then all the legal values do not have to be offered (some may be refused). This is represented by the conjunction of the *MayRead* clause, and the safety condition that the value returned is legal.

An alternative, and arguably more understandable specification of *PoolValue* is given on Page 161.

4.5.4 Channel Family

The channel protocol has several derivatives, and it is more elegant to illustrate how they relate to each other by giving a general definition.

A (derivative of a) channel may hold up both the writer and the reader. Both the reader and writer may always start. However, the middle event is not permitted for the reader until there is an item to read. The writer cannot engage in an end-write until there is space to write the item.

The following predicate is given in terms of $ChannelState$, which takes the trace, and returns a sequence of values representing the 'current' state of the channel for that trace.

$$\begin{aligned} ChannelFamily(D, n) \quad = \quad & Basic(D) \\ & \land\ May_{sr}(D) \\ & \land\ MayBlock_{mr}(D, \\ & \qquad\qquad \#(ChannelState(tr)) \geq 1) \\ & \land\ May_{er}(D) \\ & \land\ May_{sw}(D) \\ & \land\ May_{mw}(D) \\ & \land\ MayBlock_{ew}(D, \\ & \qquad\qquad \#(ChannelState(tr)) \leq n) \\ & \land\ ChannelValue \end{aligned}$$

The $ChannelFamily$ does not block except in two circumstances: the middle-read and end-write events are not allowed to occur if there is no item, or no space for an item respectively. (The choice of which event to block has been taken from Simpson's work on standardising the IDAs [112].) However, these events have liveness conditions attached to them for when space or items are available.

Note that the predicate given to the conditional refusal predicates ($MayBlock_{mr}$ and $MayBlock_{ew}$) is true when the event may proceed: thus there must be at least one event for the reader to continue.

$ChannelValue$, together with $ChannelState$, determines which value is returned by a given read. $ChannelState$ recursively constructs a list of data values from the trace, and $ChannelValue$ uses this to determine what value a given end-read should return.

$$ChannelValue \quad = \quad tr = s^\frown\langle er.e\rangle$$

$$\Rightarrow \; e = \mathit{head}(\mathit{ChannelState}(s))$$

$$\begin{array}{rcl}
\mathit{ChannelState}(\langle\rangle) & = & \langle\rangle \\
\mathit{ChannelState}(s'^\frown\langle mw.d\rangle) & = & \mathit{ChannelState}(s')^\frown\langle d\rangle \\
\mathit{ChannelState}(s'^\frown\langle er.e\rangle) & = & \mathit{tail}(\mathit{ChannelState}(s')) \\
\mathit{ChannelState}(s'^\frown\langle x\rangle) & = & \mathit{ChannelState}(s') \qquad \text{when} \quad \begin{array}{rl} & x \neq mw.d \\ \wedge & x \neq er.e \end{array}
\end{array}$$

where for s a sequence, and n a positive integer, s_n denotes the nth member of the sequence s. *tail* returns all but the first element of a sequence.

The channel protocol is easily defined as a single instance of *ChannelFamily*; the interlocked variant of the channel is the rendezvous, and the queued variant is the bounded buffer:

$$\begin{array}{rcll}
\mathit{Rendezvous}(D) & = & \mathit{ChannelFamily}(D, 0) & \\
\mathit{Channel}(D) & = & \mathit{ChannelFamily}(D, 1) & \\
\mathit{BoundedBuffer}(D, n) & = & \mathit{ChannelFamily}(D, n) & \qquad (n > 1)
\end{array}$$

4.5.5 Signal Family

The signal protocol, like the channel, has several derivatives. A signal can hold up the reader, but it never holds up the writer. If the signal runs out of space for the writer, it overwrites old data.

$$\begin{array}{rcl}
\mathit{SignalFamily}(D, n) & = & \quad \mathit{Basic}(D) \\
& & \wedge\ \mathit{May}_{sr}(D) \\
& & \wedge\ \mathit{MayBlock}_{mr}(D, \\
& & \qquad\qquad \#(\mathit{SignalState}(n, tr)) \geq 1) \\
& & \wedge\ \mathit{May}_{er}(D) \\
& & \wedge\ \mathit{MayWrite}(D) \\
& & \wedge\ \mathit{SignalValue}(D, n)
\end{array}$$

SignalFamily uses blocking and liveness conditions in a similar fashion to *ChannelFamily*.

SignalValue is a slightly cumbersome definition because the value returned by an end-read is the value at the front of the *SignalState* queue at the point of the middle-read event, *i.e.* the value to be returned by er is indicated by mr.

$$\begin{array}{rcl} \mathit{SignalValue}(D,n) & = & \quad \mathit{last}(tr) = er.d \\ & & \Rightarrow\ d = \mathit{head}(\mathit{SignalState}(n,s')) \\ & & \text{where} \quad tr = s'^\frown\langle mr\rangle^\frown s'' \\ & & \qquad\quad \wedge\ \ mr \notin \sigma(s'') \end{array}$$

Because writes can overwrite unread values when there is no space, $\mathit{SignalState}$ has extra terms covering end-writes: if there is no space when the ew event occurs, then the head of the list is discarded. Correspondingly, the value returned by a read is determined, and removed from the $\mathit{SignalState}$ sequence, at the mr event.

$$\begin{array}{rcl} \mathit{SignalState}(n,\langle\rangle) & = & \langle\rangle \\ \mathit{SignalState}(n,s'^\frown\langle mr\rangle) & = & \mathit{tail}(\mathit{SignalState}(n,s')) \\ \mathit{SignalState}(n,s'^\frown\langle mw.d\rangle) & = & \mathit{SignalState}(n,s')^\frown\langle d\rangle \\ \mathit{SignalState}(n,s'^\frown\langle ew.d\rangle) & = & \mathit{SignalState}(n,s') \\ & & \qquad \text{when } \#(\mathit{SignalState}(n,s')) \le n \\ \mathit{SignalState}(n,s'^\frown\langle ew.d\rangle) & = & \mathit{tail}(\mathit{SignalState}(n,s')) \\ & & \qquad \text{when } \#(\mathit{SignalState}(n,s')) > n \\ \mathit{SignalState}(n,s'^\frown\langle x\rangle) & = & \mathit{SignalState}(n,s') \quad \text{when} \quad x \ne mr.d \\ & & \qquad\qquad\qquad\qquad\qquad \wedge\ \ x \ne mw.d \\ & & \qquad\qquad\qquad\qquad\qquad \wedge\ \ x \ne ew.d \end{array}$$

The signal protocol is now defined as a single instance of $\mathit{SignalFamily}$; the interlocked variant of the signal is the flash data protocol, and the queued variant is the overwriting buffer:

$$\begin{array}{rcl} \mathit{FlashData}(D) & = & \mathit{SignalFamily}(D,0) \\ \mathit{Signal}(D) & = & \mathit{SignalFamily}(D,1) \\ \mathit{OverwritingBuffer}(D,n) & = & \mathit{SignalFamily}(D,n) \qquad (n > 1) \end{array}$$

4.5.6 Void Protocols

The void protocols are variants of the previously defined protocols which pass no information. An alternative view is that the set D contains exactly one member, so this is the only value that may be written or read. We will use the symbol '$\Diamond$' to denote this single value.

$$\begin{array}{rcll}
\mathit{Prod} & = & \mathit{FlashData}(\{\Diamond\}) & \\
\mathit{Stimulus} & = & \mathit{Signal}(\{\Diamond\}) & \\
\mathit{OverwritingStimBuffer}(n) & = & \mathit{OverwritingBuffer}(\{\Diamond\}, n) & (n > 1) \\
\mathit{DirectionalHandshake} & = & \mathit{Rendezvous}(\{\Diamond\}) & \\
\mathit{DatalessChannel} & = & \mathit{Channel}(\{\Diamond\}) & \\
\mathit{BoundedStimBuffer}(n) & = & \mathit{BoundedBuffer}(\{\Diamond\}, n) & (n > 1)
\end{array}$$

4.5.7 Integral and Separate Response Protocols

The integral and separate response protocols are specific examples of devices, or complex routes. We will not consider these particular instances further in this thesis since the issues surrounding them are dealt with in Section 4.7 (*Composition*).

4.6 IDA Semantics

In Section 3.6.1 (Page 67), we postponed the definition of $\mathcal{NRS}[\![\, n, t \,]\!]$, the semantic function for a route name n of type t. We are now able to give that definition.

Like the function $\mathcal{B}$ (Page 67), $\mathcal{NRS}$ is a three-tuple:

$$\mathcal{NRS}[\![\, n, t \,]\!] = (\mathcal{NRS}_P[\![\, n, t \,]\!], \mathcal{NRS}_I[\![\, n, t \,]\!], \mathcal{NRS}_W[\![\, n, t \,]\!])$$

which gives the CSP process, the IDA communication events, and the processor scheduling events. Since in this model, we do not allow the scheduler to directly interfere with IDA events (see Section 4.4, Page 87):

$$\mathcal{NRS}_W[\![\, n, t \,]\!] = \emptyset$$

i.e. the IDA events have no effect on the scheduler.

We now define 'alphabet relabelling' for a name X and alphabet A:

$$X.A = \{e \mid e = X.e' \wedge e' \in A\}$$

and relabel the IDA alphabet αIDA with the fully-qualified unique name, $\mathcal{F}$ (Page 73), to define $\mathcal{NRS}_I$:

$$\mathcal{NRS}_I[\![\, n, t \,]\!] = \mathcal{F}.\alpha IDA$$

i.e. that instance of the IDA is given a unique alphabet. (We will assume that the set of IDA data values D is constant across a given system, and thus ignore it here.)

Finally, we can define $\mathcal{NRS}_P$ to be any process that satisfies the specification of the indicated IDA type (*e.g.* $Pool$, $Channel$,...), and is renamed to match the alphabet given by $\mathcal{NRS}_I$.

4.7 Composition

What would be the result of the connection of two or more routes?

To examine this, we use a composition process (a simple activity in the DORIS syntax). Suppose the two routes are called A and B, where A has alphabet D and B has an alphabet that is a superset of D. We use one of two processes to connect the two routes:

Read in-write out This process accepts a full transaction from A (reading), and then writes it out to B before accepting another start-read from A:

$$\begin{aligned} ReadInWriteOut \quad = & \quad A.sr \rightarrow A.mr \rightarrow A.er?d \\ & \rightarrow B.sw.d \rightarrow B.mw.d \rightarrow B.ew.d \\ & \rightarrow ReadInWriteOut \end{aligned}$$

Unbounded buffer This process accepts values and writes them as fast as it can, storing up values read as a sequence (Figure 4.6).

Figure 4.7 is a state machine for the unbounded buffer in Figure 4.6, and is a clearer representation of the process. The annotations on the transitions are of the form

$$g\&e : a$$

where g is a guard condition; e is the CSP event; and a is an action on variables. An absent guard condition is considered to be true. The interpretation intended is that the event e can only be engaged in if g is true, and when e occurs, the actions a are executed.

$$
\begin{aligned}
\mathit{UnboundedBuffer} &= U_{ss}(\langle\rangle) \\
U_{ss}(\langle d\rangle^\frown s) &= B.sw.d \rightarrow U_{ms}(s,d) \;\Box\; A.sr \rightarrow U_{sm}(s) \\
U_{sm}(\langle d\rangle^\frown s) &= B.sw.d \rightarrow U_{mm}(s,d) \;\Box\; A.mr \rightarrow U_{se}(s) \\
U_{se}(\langle d\rangle^\frown s) &= B.sw.d \rightarrow U_{me}(s,d) \;\Box\; A.er?e \rightarrow U_{ss}(s^\frown\langle e\rangle) \\
U_{ss}(\langle\rangle) &= A.sr \rightarrow U_{sm}(\langle\rangle) \\
U_{sm}(\langle\rangle) &= A.mr \rightarrow U_{se}(\langle\rangle) \\
U_{se}(\langle\rangle) &= A.er?e \rightarrow U_{ss}(\langle e\rangle) \\
U_{ms}(s,d) &= B.mw.d \rightarrow U_{es}(s,d) \;\Box\; A.sr \rightarrow U_{mm}(s,d) \\
U_{mm}(s,d) &= B.mw.d \rightarrow U_{em}(s,d) \;\Box\; A.mr \rightarrow U_{me}(s,d) \\
U_{me}(s,d) &= B.mw.d \rightarrow U_{ee}(s,d) \;\Box\; A.er?e \rightarrow U_{ms}(s^\frown\langle e\rangle, d) \\
U_{es}(s,d) &= B.ew.d \rightarrow U_{ss}(s) \;\Box\; A.sr \rightarrow U_{em}(s,d) \\
U_{em}(s,d) &= B.ew.d \rightarrow U_{sm}(s) \;\Box\; A.mr \rightarrow U_{ee}(s,d) \\
U_{ee}(s,d) &= B.ew.d \rightarrow U_{se}(s) \;\Box\; A.er?e \rightarrow U_{es}(s^\frown\langle e\rangle, d)
\end{aligned}
$$

Figure 4.6: 'Unbounded buffer' composition of routes

Note that the unbounded buffer has an infinite state graph; the parameterization of the definition given above allows us to give a finite graph representing the control flow.

In Chapter 7, we will see what properties, if any, are possessed by the basic IDAs composed together.

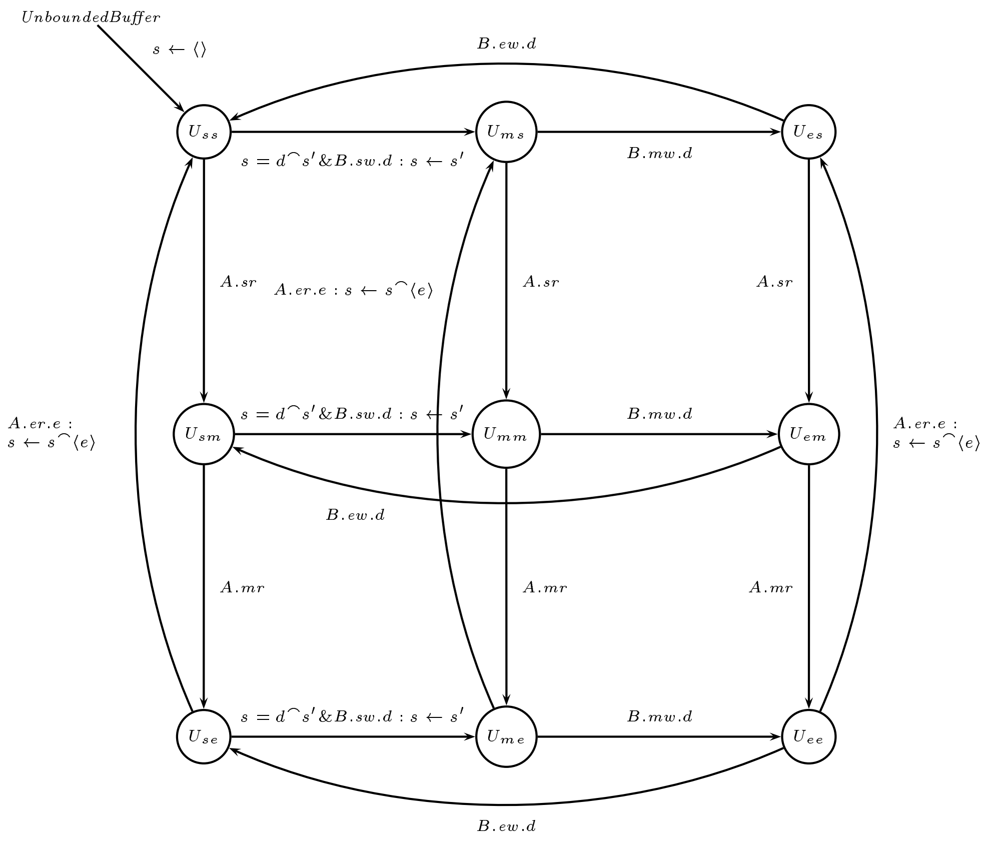


Figure 4.7: State machine for the unbounded buffer

4.8 Multiplexing

Simpson and Paynter have suggested multiplexing operators for IDAs [110]. Instead of composing one IDA on each side, these multiplexing operators have n IDAs on one side, and one on the other (although a generalisation to m on one side and n on the other is possible).

An n-ary merge takes n IDAs, and merges them into one IDA output. The only IDAs that can be merged are channels and signals: pools and constants would continuously feed the same value, and are not generally suitable for this type of merge.

Three possible strategies for collecting input are:

Round robin Each input IDA is polled, and waited for, until it has supplied a value (or additionally, after a timeout, proceed to the next IDA input).

Priority Each IDA is polled in a particular order using timeouts, until one can supply a value, at which point return to the first IDA.

Random An IDA is chosen at random.

A fourth possibility is analogous to the Ada select statement, *e.g.* for n input IDAs, A_i,

$$\begin{aligned} Multiplex \quad = \quad & \Big(|||_{i:\{1,\ldots,N\}} (\mu X \bullet \quad A_i.sr \to A_i.mr \to A_i.er?d \\ & \qquad \to B.sw!d \to B.mw.d \to B.ew.d \\ & \qquad \to X) \Big) \\ & \| \Big(\mu X \bullet B.sw?e \to B.mw.e \to B.ew.e \to X \Big) \end{aligned}$$

The program above collects input from each IDA as soon as it is ready. The value is then offered via the IDA B. The combination of interleaving the input IDA reads with the synchronous parallel composition of the outputs means that only one value is output at a time.

It is easy to devise variants on this theme, *e.g.* and-combinations:

and-combination Wait until every input IDA has supplied a value, then output the values as a single tuple.

(The program $Multiplex$ can be viewed as an or-combination.)

Similarly, the output could be distributed to all output IDAs ('replication') or to one (randomly chosen) IDA ('split').

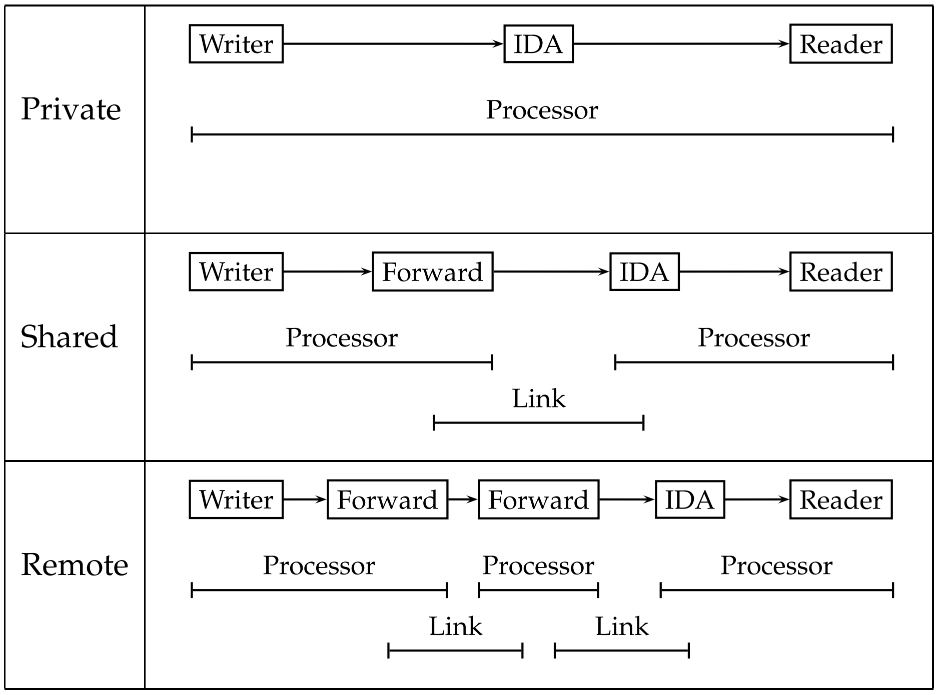


Figure 4.8: Simple forwarding across transputer links

4.9 Forwarding

Section 4.1.2 described the situation where an IDA is located within the transputer network, and therefore requires a *forwarding* process. Figure 4.8 illustrates the three cases that occur. In the shared and remote cases, a process (marked 'Forward') arranges for the interaction to be carried out across the transputer link.

A simple approach is to use the read in-write out process given on Page 99 or the unbounded buffer (Figure 4.6).

A more complex approach involves using a CSP process at each forwarding point, and then using the transputer link information to implement a transputer link mutual exclusion (mutex) scheme. This could operate in the same fashion as the transputer scheduler described in the next chapter (Section 5.6), or could use one of the strategies from the previous section.

4.10 Implementation

The overall proof strategy for a large system would involve defining lemmas about smaller parts of the system to reduce the overall complexity of the problem. Proofs about IDAs are useful for this: ordinarily, the predicate definitions of the IDAs would be used, so that the underlying definitions could be avoided.

This imposes the proof obligation that the implementation of an IDA (say, as a Timed CSP program, using definitions of Lamport's variables) satisfies the specification.

We now define the two-point untimed programs for each of the three types of Lamport's variables with data domain D and initial value d_0. In each case, one reader and one writer is assumed. (See Section 4.2.1 for the description of Lamport's variables.) We are then able to 'implement' the four-slot model.

4.10.1 Lamport's Safe Variable

$$Safe(D, d_0) \quad = \quad LS(D, d_0)$$

The safe variable can be represented as a parameterised state machine with four states. The four states arise from the cross product of reading or not reading, and writing or not writing.

$$\begin{aligned}
LS(D, v) \quad &= \quad sw?d : D \rightarrow LS_W(D, v, d) \\
& \Box\ sr \rightarrow LS_R(D, v, \{v\}) \\[1ex]
LS_W(D, v, d) \quad &= \quad ew.d \rightarrow LS(D, d) \\
& \Box\ sr \rightarrow LS_{WR}(D, v, d, D) \\[1ex]
LS_R(D, v, R) \quad &= \quad sw?d : D \rightarrow LS_{WR}(D, v, d, D) \\
& \Box\ \sqcap x : \{e : R \bullet er.e\} \rightarrow LS(D, v) \\[1ex]
LS_{WR}(D, v, d, R) \quad &= \quad ew.d \rightarrow LS_R(D, d, R) \\
& \Box\ \sqcap x : \{e : R \bullet er.e\} \rightarrow LS_W(D, v, d)
\end{aligned}$$

The components of $Safe$ have up to four parameters associated with them:

D the alphabet;

v the 'value' of the variable, which is either the initial variable, or the value of the last end-write;

d the value currently being written; and

R the set of values that can be returned to a read: either a singleton element (for non-overlapped reads) or all of D (for overlapped reads).

4.10.2 Lamport's Regular Variable

$$\mathit{Regular}(D, d_0) \quad = \quad LR(D, \{d_0\})$$

As for the safe variable, the regular variable has four states.

$$\begin{array}{rcl}
LR(D, v) & = & \quad sw?d : D \rightarrow LR_W(D, v, d) \\
& & \Box \; sr \rightarrow LR_R(D, v, \{v\}) \\[1em]
LR_W(D, v, d) & = & \quad ew.d \rightarrow LR(D, d) \\
& & \Box \; sr \rightarrow LR_{WR}(D, v, d, \{v, d\}) \\[1em]
LR_R(D, v, R) & = & \quad sw?d : D \rightarrow LR_{WR}(D, v, d, R \cup \{d\}) \\
& & \Box \; \sqcap x : \{e : R \bullet er.e\} \rightarrow LR(D, v) \\[1em]
LR_{WR}(D, v, d, R) & = & \quad ew.d \rightarrow LR_R(D, d, R) \\
& & \Box \; \sqcap x : \{e : R \bullet er.e\} \rightarrow LR_W(D, v, d)
\end{array}$$

The components of *Regular* have up to four parameters associated with them:

D the alphabet;

v the 'value' of the variable, which is either the initial variable, or the value of the last end-write;

d the value currently being written; and

R the set of values that can be returned to a read.

4.10.3 Lamport's Atomic Variable

$$
\begin{aligned}
Atomic(D, d_0) &= LA(D, d_0) \\
LA(D, v) &= \quad sw.d \to ew.d \to LA(D, d) \\
&\quad \Box\ sr \to er.v \to LA(D, v)
\end{aligned}
$$

$Atomic$ is the simplest of the three types of variable: we implement it here with a simple mutual exclusion scheme. (This implementation is not the most general process for the description of atomic variables given in Section 4.2.1, Page 81.)

4.10.4 Four-slot Implementation of the Pool

We now take the four-slot implementation of the pool (with alphabet D and initial value d_0), and implement it using regular variables [9].

The implementation consists of the parallel composition of three groups of components:

- the reader and writer;
- the shared data slots; and
- the bit-valued control variables.

$$
\begin{aligned}
Fourslot = \quad & Writer \| Reader \\
& \| data.0.0 : DS \| data.0.1 : DS \| data.1.0 : DS \| data.1.1 : DS \\
& \| reading : Bit \| latest : Bit \| slot.0 : Bit \| slot.1 : Bit
\end{aligned}
$$

where

$$
\begin{aligned}
DS &= Regular(D, d_0) \\
Bit &= Regular(\{0, 1\}, 0)
\end{aligned}
$$

(*i.e.* DS is a regular variable with initial value 0, and data domain D, and Bit is a bit valued regular variable.)

$$
Writer = \quad sw?d
$$

$$
\begin{aligned}
& \to reading.sr \to reading.er?p \\
& \to slot.\bar{p}.sr \to slot.\bar{p}.er?i \\
& \to data.\bar{p}.\bar{i}.sw.d \to data.\bar{p}.\bar{i}.ew.d \\
& \to mw.d \\
& \to slot.\bar{p}.sw.\bar{i} \to slot.\bar{p}.ew.\bar{i} \\
& \to latest.sw.\bar{p} \to latest.ew.\bar{p} \\
& \to ew.d \\
& \to Writer
\end{aligned}
$$

$$
\begin{aligned}
Reader \quad = \quad & \quad sr \\
& \to latest.sr \to latest.er?p \\
& \to reading.sw.p \to reading.ew.p \\
& \to slot.p.sr \to slot.p.er?i \\
& \to mr \\
& \to data.p.i.sr \to data.p.i.er?d \\
& \to er.d \\
& \to Reader
\end{aligned}
$$

($\bar{b}$, where b is a bit value, returns the inverse of b.) In Chapter 7, we attempt to demonstrate that this satisfies the specification of the pool.

4.11 Timed IDAs

4.11.1 Specifications

Section 4.5 described a number of *untimed* specifications for IDAs. The previous section has illustrated how such untimed specifications can be implemented. But interactions with IDAs can take time: how do we extend these definitions?

In the next chapter, we will explicitly model schedulers that could deschedule the activities that interact with the IDAs. In this section, we require that there is no (scheduler) interference with IDA timing.

Then, all that is required in the specifications is a change to the refusal predicates on Page 91. We now make these changes[4]:

$$\begin{array}{rcl}
May_{sr}(D) & = & \quad (\tau \downarrow \alpha R(D) = \langle\rangle) \\
& & \quad \Rightarrow \ sr \ \textbf{live within} \ \hat{t_0} \ \textbf{of} \ 0 \ \textbf{until} \ \{sr\} \\
& \wedge & \quad (foot(\tau \downarrow \alpha R(D)) = (t, er) \\
& & \quad \Rightarrow \ sr \ \textbf{live within} \ \hat{t} \ \textbf{of} \ t \ \textbf{until} \ \{sr\} \\
May_{mr}(D) & = & (foot(\tau \downarrow \alpha R(D)) = (t, sr) \\
& \Rightarrow & mr \ \textbf{live within} \ \hat{t} \ \textbf{of} \ t \ \textbf{until} \ \{mr\} \\
May_{er}(D) & = & (foot(\tau \downarrow \alpha R(D)) = (t, mr) \\
& \Rightarrow & \exists A : \mathbb{P}^{+} ER(D) \bullet A \ \textbf{live within} \ \hat{t} \ \textbf{of} \ t \ \textbf{until} \ A
\end{array}$$

$$\begin{array}{rcl}
May_{sw}(D) & = & \quad (\tau \downarrow \alpha W(D) = \langle\rangle) \\
& & \quad \Rightarrow \ SW(D) \ \textbf{live within} \ \hat{t_0} \ \textbf{of} \ 0 \ \textbf{until} \ SW(D) \\
& \wedge & \quad (foot(\tau \downarrow \alpha W(D)) = (t, ew) \\
& & \quad \Rightarrow \ SW(D) \ \textbf{live within} \ \hat{t} \ \textbf{of} \ t \ \textbf{until} \ SW(D) \\
May_{mw}(D) & = & (foot(\tau \downarrow \alpha W(D)) = (t, sw.d) \\
& \Rightarrow & mw.d \ \textbf{live within} \ \hat{t} \ \textbf{of} \ t \ \textbf{until} \ \{mw.d\} \\
May_{ew}(D) & = & (foot(\tau \downarrow \alpha W(D)) = (t, mw.d) \\
& \Rightarrow & ew.d \ \textbf{live within} \ \hat{t} \ \textbf{of} \ t \ \textbf{until} \ \{ew.d\}
\end{array}$$

These first six predicates are for the six individual IDA events, and have the form 'if the last event precedes this event, then this event must be live no later than a specified delay'.

In each case, $\hat{t}$ is the maximum delay permitted before the event becomes live, except for the very first read and write, when $\hat{t_0}$ ('set-up time') is the maximum permitted delay.

$$\begin{array}{rcl}
MayRead(D) & = & May_{sr}(D) \wedge May_{mr}(D) \wedge May_{er}(D) \\
MayWrite(D) & = & May_{sw}(D) \wedge May_{mw}(D) \wedge May_{ew}(D)
\end{array}$$

These two predicates conjoin each group of three predicates for the reader and writer events. A more complex model could easily be formed by allowing six different values of $\hat{t}$, one for each predicate.

[4]Additional notations used in these predicates are '$\mathbb{P}$' for powerset, and '$\mathbb{P}^{+}$' for powerset excluding the empty set. Refer to Page 40 for further details on the Timed CSP macros.

We also need to modify the conditional refusal predicates from Page 92. These have to allow for c being true at some point in a timed trace, but then being false before the event concerned has occurred — thus withdrawing the offer to engage in that event. When this occurs, the delay $\hat{t}$ is again permitted. (Here, c is considered to be a predicate with the timed trace τ a free variable.)

This possibility of withdrawing the offer complicates this issue. It is useful to define $\mathit{LiveIntervals}$, which returns all half-open intervals after t where c is true. $\mathit{MayBlock}_{mr}$ and $\mathit{MayBlock}_{ew}$ (below) exploit this by saying that if the predecessor event has occurred at time t, then the IDA is prepared to engage in the next event after a delay of no more than $\hat{t}$. The '**of** t_2' term allows for the offer to engage in the event to be withdrawn at time t_2. The final part of the conjunction requires that c was true if the event concerned has just occurred.

So we can define the timed conditional refusal predicates:

$$\begin{aligned}
\mathit{LiveIntervals}(c,\tau) &= \{(t_1,t_2) \in \mathit{Time} \times \mathit{Time} \mid \\
&\qquad\qquad \forall t' \in [t_1,t_2) \bullet c(\tau \uparrow [0,t'))\}
\end{aligned}$$

$$\begin{aligned}
\mathit{MayBlock}_{mr}(D,c) &= \quad (\mathit{foot}(\tau \downarrow \alpha R(D)) = (t, sr)) \\
&\quad \Rightarrow \forall (t_1,t_2) \in \mathit{LiveIntervals}(c,\tau) \bullet \\
&\qquad\qquad mr \;\mathbf{live\ within}\; \hat{t} \;\mathbf{of}\; t_1 \;\mathbf{until}\; \{mr\} \;\mathbf{or}\; t_2 \\
&\quad \wedge (\tau = s^\frown\langle (t, mr)\rangle) \Rightarrow c
\end{aligned}$$

$$\begin{aligned}
\mathit{MayBlock}_{ew}(D,c) &= \quad (\mathit{foot}(\tau \downarrow \alpha W(D)) = (t, mw.d)) \\
&\quad \Rightarrow \forall (t_1,t_2) \in \mathit{LiveIntervals}(c,\tau) \bullet \\
&\qquad\qquad ew.d \;\mathbf{live\ within}\; \hat{t} \;\mathbf{of}\; t_1 \;\mathbf{until}\; \{ew.d\} \;\mathbf{or}\; t_2 \\
&\quad \wedge (\tau = s^\frown\langle (t, ew.d)\rangle) \Rightarrow c
\end{aligned}$$

Much of the complexity in this definition is due to the difference between untimed and timed failures. An untimed failure (tr, rf) means that the events in the set rf were refused *after* the trace tr occurred. The corresponding timed failure $(\tau, \aleph)$ means that *while* the timed trace τ occurred, various events were refused at the times indicated by the timed refusal set $\aleph$. This means that we have to give the constraints for the entire history when considering a timed failure.

4.11.2 Implementations

The implementations are easier to handle: a simple way is to use the delayed prefix

$$a \xrightarrow{t} P = a \rightarrow Wait\ t; P$$

and use this in place of '$\rightarrow$' in the definitions of the Lamport variables. This results in a delay every time a 'real' variable (*i.e.* a variable at a lower level of abstraction) is accessed.

The drawback to these definitions is that they have much more information in them. It is not clear at the outset whether such specifications are amenable to proof. This will be addressed in Chapter 7.

Chapter 5

Language for Activities

5.1 Overview

In the previous two chapters, the details of how a simple server or simple activity is defined have been omitted. This chapter describes these details.

What is a simple activity? It is a ‘processing’ or ‘active’ node of the DORIS network. At an implementation level, the activity may be (for example) an Ada task. However, at the design and distribution levels, we are only concerned with periods when the activity is working (*i.e.* when it requires processor resources) and its interactions with its ports (which are connected to routes when placed in context). This allows a large level of abstraction, while capturing the interactions that we are interested in.

A simple server is a specialised simple activity that interacts with a piece of external hardware. Conceptually, such a piece of hardware is allowed to ‘store’ data in a fashion similar to a route, so servers are allowed windows as well as ports.

There are a number of requirements for a language to describe activities:

- It must be compact — a representation that unnecessarily uses many pages is not useful.
- The assumptions must easily fit into the DORIS concepts described in the previous chapter.
- It should be easy to translate into the semantic domain, *i.e.* Timed CSP.

- Finally, and possibly most importantly, it must be easy to learn, read, and write.

One possibility involves simply embedding Timed CSP programs. This has the drawbacks that it can be confusing for non-specialists, and that in any case, some degree of processing on the program is required to map the ports to routes.

The second approach, which is adopted in this thesis, is to model an activity's important events by using a very simple language that is easily translatable to Timed CSP, while allowing for the interactions with the scheduler and ports to be captured.

```
An_Example = Work(5, 10);
             Write(p1, x);
             Self
```

and

```
Another_Example = ExtChoice(Read(p1, y);
                            Work(10, 30);
                            Self,
                            Read(p2, y);
                            Work(6, 12);
                            Self)
```

are simple examples of this language, which is defined later in this chapter. Both are loops: the first is very simple, and carries out between five and ten time units worth of work, and writes out some value onto port '1'. The second has a choice of ports to read from, and then carries out a different amount of work based on the port read.

The general point here is that the language only has to capture interactions with the outside world, and indicate the times that it needs processing resources (*i.e.* 'Work').

There are two other possible approaches which we will first examine: Paynter *et al.*'s Activity Description Language (ADL) [83], and graphical notations in general.

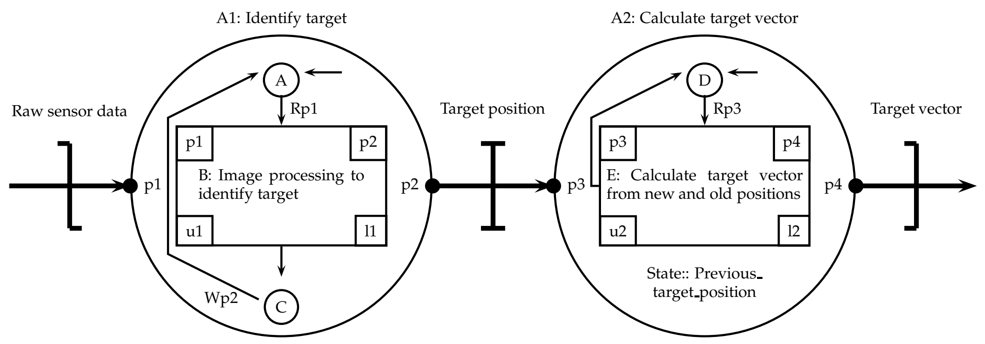

A1: Identify target
A2: Calculate target vector
Raw sensor data
Target position
Target vector
A
D
Rp1
Rp3
p1
p2
p3
p4
B: Image processing to identify target
E: Calculate target vector from new and old positions
u1
l1
u2
l2
Wp2
C
State:: Previous_target_position

Figure 5.1: A target tracker in ADL

5.1.1 Activity Description Language

ADL [83] is intended to fill a similar 'gap' in the DORIS notation as the simple language in this chapter. Its major strength is that it makes use of the concept of 'deadlines' when specifying these activities. 'Static' states are points where the process may be descheduled, and 'dynamic' states have some processing action associated with them. The underlying semantic foundation of the ADL is RTL. ADL is currently under development.

Consider the diagram in Figure 5.1. This example is Figure 2 (Page 13) of Paynter's paper [83]. In 'Identify target', the transition 'Rp1' occurs when an input becomes available on 'p1' from the signal, 'Raw sensor data'. Between 'l1' and 'u1' units of work are carried out, and then the system will wait until there is space to write on 'p2', *i.e.* space to write on the channel 'Target position'.

The major difference from the work presented here is that the ADL specifies in more detail what is meant to happen; the language in the rest of this chapter is an abstraction to model the timing of the system, although it can capture functional properties.

5.1.2 Graphical Notations

A possible alternative involves using Harel's Statecharts [40] (*e.g.* see Figure 5.2, taken from Harel's paper [40]). However, Statecharts themselves were rejected quickly due to a number of problems [6, 10]. (For example, Statecharts allow an infinite amount of work in a finite time, and are often ambiguous.)

Instead, a Statechart-like notation closely based on Timed CSP was considered. This notation had a very close relationship with the structure of the textual grammar. Although the notation was reasonably easy to understand, it would not have been particularly compact for the level of detail required.

Anecdotally, it would appear that graphical notations that represent the imperative-like design of an activity give rise to very large descriptions. It is more efficient to spend a small amount of time learning a more compact notation, and be subsequently more effective.

In particular, whether or not a particular notation is usable or aesthetically pleasing is a subjective matter. However, graphical notations, by virtue of using their layout for imparting information, rather than textual symbols, suffer more from secondary notation than textual notations [88], *i.e.* the layout of the graphical notation may be essential to its meaning, whereas a badly formatted C program can be difficult to read and understand, but it has exactly the same meaning as a 'nicely' formatted version.

It is this secondary notation (*e.g.* layout and typographic cues) that sometimes appeals to users, and causes confusion in interpreting such notations. There is an increasing amount of work on the subject of 'visualisation' [32, 33, 87]. It is largely inconclusive; it is not clear what an appropriate notation (textual or graphical) is in a given context. There are no obvious rules to apply.

Leveson *et al.* have described design criteria for such a language [64]:

- semantically consistent
- unambiguous intuitive meaning
- readability given priority over writeability
- easy to hand-draw
- easy to computer-draw

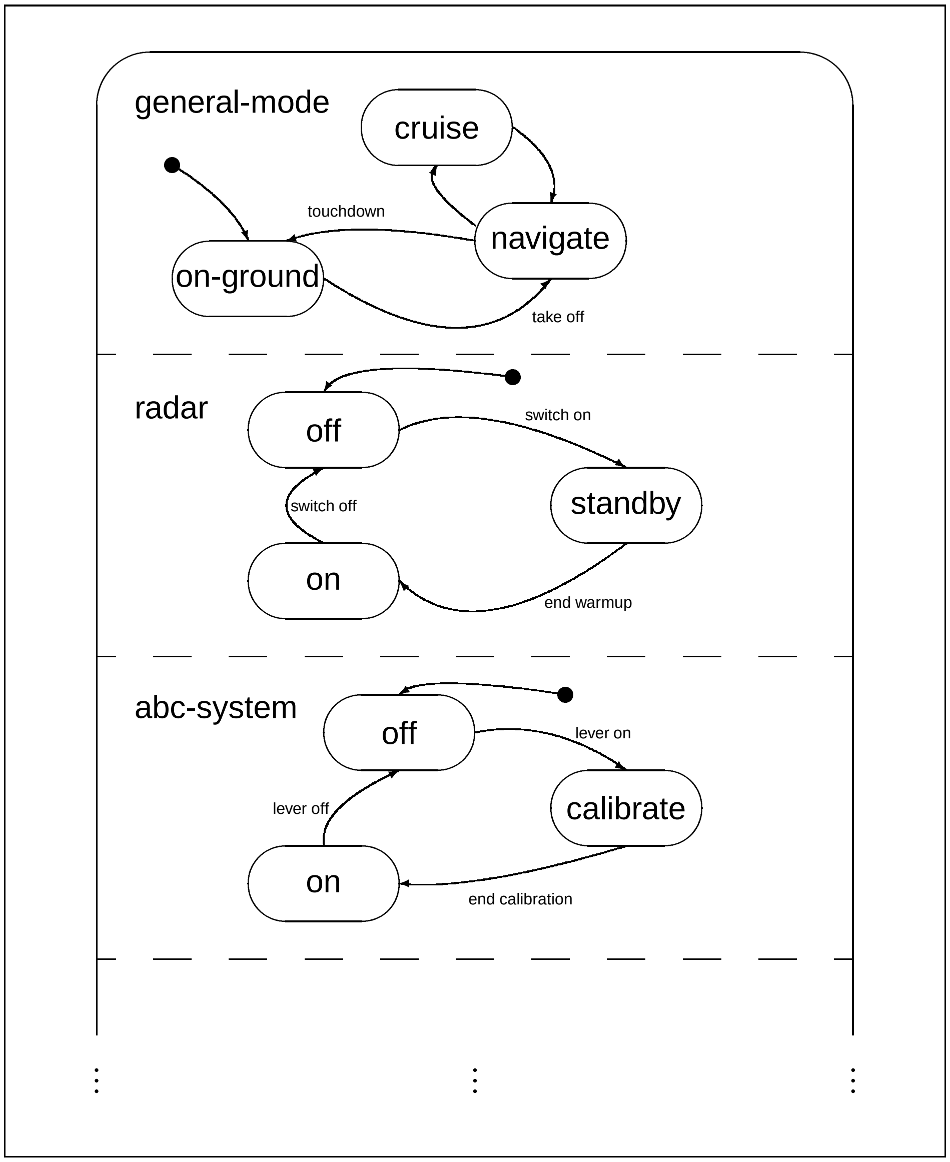


Figure 5.2: Statechart for part of an avionics system

The aim of the work described in this thesis was to produce a semantics for DORIS that was acceptable to engineers. A graphical notation would (on current trends) be appealing to users, but there are no clear benefits to introducing such a notation. Therefore, in the rest of this thesis, we shall remain firmly in the textual domain.

5.2 Alphabets and Events

At this point, it is useful to identify the alphabet of the CSP program for each basic component; *i.e.* the alphabet of the programs returned by $\mathcal{NSS}$ and $\mathcal{NAS}$ (the semantic functions which return the CSP for a simple server or simple activity of given name, respectively). There are two parts: the 'working' or 'processing' alphabet, and the alphabet associated with any IDAs the activity is connected to.

The language consists of a collection of expressions (introduced in the next section) that are translated to Timed CSP programs. One of these expressions is Work(l,u), where $0 \leq l \leq u$. This expression represents an activity undertaking some work that takes an amount of processor time between l and u. Three events are involved with modelling this: sp, mp, and ep (start, middle and end of processing). Using the CSP renaming function, and the FQUN semantic function, $\mathcal{F}$ (defined in the previous chapter), we have three events unique to each instance of a simple server or simple activity that can be used for scheduler modelling.

The significance of the processing events and the times attached to them ($sp.(l,u)$, mp and ep) are that:

- The $sp.(l,u)$ event represents the start of processing, where $l, u : Time; l \leq u$ (although we will sometimes refer to sp unadorned).

- The mp event means that at least l time units of scheduled time have passed since the immediately preceding sp event.

- The ep event represents the end of processing, and should not occur more than u time units of scheduled time after the immediately preceding sp event.

  Since it is impossible to code 'must occur' in CSP (or, indeed, in the real world), the interpretation is more precisely stated as 'must be offered to the environment'. Of course, when these events are hidden (as they ultimately would be), the 'must occur' coincides with 'must be offered to the environment'.

We can then define the 'processing alphabet'

$$\alpha P = \{sp.(l,u) \mid l,u : \mathit{Time}; l \leq u\} \cup \{mp, ep\}$$

We require that these three events follow a sequence, as defined by the following predicates:

$$\begin{aligned}
\mathit{Working}_A &= (tr\#sp - tr\#mp = 1) \wedge (tr\#mp = tr\#ep) \\
\mathit{Working}_B &= (tr\#sp = tr\#mp) \wedge (tr\#mp - tr\#ep = 1) \\
\mathit{NotWorking} &= (tr\#sp = tr\#mp = tr\#ep) \\
\\
\mathit{WorkSequence} &= \mathit{Working}_A \vee \mathit{Working}_B \vee \mathit{NotWorking}
\end{aligned}$$

This predicate is similar to those in the previous chapter for the IDA predicates. (Scheduler modelling issues are discussed in Section 5.6.)

Similarly, (from the previous chapter) three events are associated with reading from a route: sr, mr, and er (start, middle, and end of read). A further three events are associated with writing to a route: sw, mw, and ew (start, middle, and end of write). Again, Rename is used to produce six events unique to the instance of the route, this time using $\mathcal{R}$ (also defined in the previous chapter). Some of these events appear in the alphabet of any simple server or activity that accesses the route: *i.e.* an activity which reads from port p has the events $\mathcal{R}(\mathsf{p}p).sr$, $\mathcal{R}(\mathsf{p}p).mr$, and $\mathcal{R}(\mathsf{p}p).er$ in its alphabet.

As before, four of the events associated with routes carry a value (*i.e.* they are channels): er (the value read); and sw, mw, and ew (the values written). The previous chapter covered routes in more detail.

When writing 'raw' CSP (rather than using the simple language given in this chapter), there is one major consideration: the events that are visible outside the program (*i.e.* not hidden) should be a (not necessarily strict) subset of those given here:

- For simple activities:
  - the processing events: $\mathcal{F}(n).sp.(l,u)$, $\mathcal{F}(n).mp$, and $\mathcal{F}(n).ep$; and
  - events for each route accessed via a port: $\mathcal{R}(p).sr$, $\mathcal{R}(p).mr$, $\mathcal{R}(p).er.v$ (or the writing variant of these events as appropriate).

- For simple servers: the same as for simple activities, with the addition of events for each route accessed via a window.
- For routes:
  - events for each route accessed via a window: $\mathcal{R}(p).sr$, $\mathcal{R}(p).mr$, $\mathcal{R}(p).er.v$ (or the writing events as appropriate).

Additionally, within any particular simple server, activity or route, a particular triplet of events (start, middle, end) should be consecutive. The first event should also be a 'start'. This restriction is so that processing events (sp, mp, and ep) can be refused for some time without affecting the IDA timing specifications.

5.3 Syntax

We now define a language which is intended to abstractly model the activities:

$$
\begin{array}{rcl}
\langle \mathit{Definition} \rangle & ::= & \langle \mathit{Name} \rangle \text{“=”} \langle \mathit{Program} \rangle \\
\langle \mathit{Program} \rangle & ::= & \text{“Stop”} \\
 & | & \text{“Wait(”} \langle \mathit{Duration} \rangle \text{“);”} \langle \mathit{Program}' \rangle \\
 & | & \text{“Work(”} \langle \mathit{Duration} \rangle \text{“,”} \langle \mathit{Duration} \rangle \text{“);”} \langle \mathit{Program}' \rangle \\
 & | & \text{“Read(”} \langle \mathit{Port} \rangle \text{“,”} \langle \mathit{Variable} \rangle \text{“);”} \langle \mathit{Program}' \rangle \\
 & | & \text{“Write(”} \langle \mathit{Port} \rangle \text{“,”} \langle \mathit{Variable} \rangle \text{“);”} \langle \mathit{Program}' \rangle \\
\langle \mathit{Program}' \rangle & ::= & \text{“Self”} \\
 & | & \langle \mathit{Program} \rangle \\
 & | & \text{“ExtChoice(”} \langle \mathit{Program}' \rangle \text{“,”} \langle \mathit{Program}' \rangle \text{“)”} \\
 & | & \text{“IntChoice(”} \langle \mathit{Program}' \rangle \text{“,”} \langle \mathit{Program}' \rangle \text{“)”} \\
\langle \mathit{Duration} \rangle & ::= & \ldots \\
\langle \mathit{Port} \rangle & ::= & \ldots \\
\langle \mathit{Variable} \rangle & ::= & \ldots
\end{array}
$$

Self means 'loop to the beginning of the program'. Stop is the termination statement. Wait means 'wait for at least the specified duration'. Read means 'read from the specified port and place the value in the variable'.

Similarly, Write means 'write the value in the variable to the specified port'. For a simple server, a $\langle Window \rangle$ may be used in place of a $\langle Port \rangle$.

ExtChoice represents external (deterministic) choice between two possible programs, and is decided on the first event offered. IntChoice is the internal (nondeterministic) operator.

The two definitions of $\langle Program \rangle$ and $\langle Program' \rangle$ occur to ensure that the program so defined is time-guarded when using a Timed CSP model.

5.4 Semantics

We can now give a three-part meaning function, $\mathcal{A}$, for this language:

$$\mathcal{A}(P, Q, F) = (\mathcal{A}_P(P, Q, F), \mathcal{A}_I(P, Q, F), \mathcal{A}_W(P, Q, F))$$

$\mathcal{A}_P$ translates the statements into the obvious CSP expressions using the (recursive) structure of the grammar.

$$\begin{aligned}
\mathcal{A}_P(\mathsf{Self}, Q, F) &= \mathcal{A}_P(Q, Q, F) \\
\mathcal{A}_P(\mathsf{Stop}, Q, F) &= Stop \\
\mathcal{A}_P(\mathsf{Wait}(d);R, Q, F) &= Wait(d); \mathcal{A}_P(R, Q, F) \\
\mathcal{A}_P(\mathsf{Work}(l,u);R, Q, F) &= F.sp.(l, u) \to F.mp \to F.ep \\
&\quad \to \mathcal{A}_P(R, Q, F) \\
\mathcal{A}_P(\mathsf{Read}(p,v);R, Q, F) &= \mathcal{R}(p).sr \to \mathcal{R}(p).mr \\
&\quad \to \mathcal{R}(p).er.v \to \mathcal{A}_P(R, Q, F) \\
\mathcal{A}_P(\mathsf{Write}(p,v);R, Q, F) &= \mathcal{R}(p).sw.v \to \mathcal{R}(p).mp.v \\
&\quad \to \mathcal{R}(p).ew.v \to \mathcal{A}_P(R, Q, F) \\
\mathcal{A}_P(\mathsf{ExtChoice}(P_1,P_2), Q, F) &= \mathcal{A}_P(P_1, Q, F) \mathrel{\Box} \mathcal{A}_P(P_2, Q, F) \\
\mathcal{A}_P(\mathsf{IntChoice}(P_1,P_2), Q, F) &= \mathcal{A}_P(P_1, Q, F) \sqcap \mathcal{A}_P(P_2, Q, F)
\end{aligned}$$

$\mathcal{A}_I$ also uses the syntax structure to recursively determine which ports and windows (and thus which routes) are accessed.

$$\begin{aligned}
\mathcal{A}_I(\mathsf{Self}, Q, F) &= \emptyset \\
\mathcal{A}_I(\mathsf{Stop}, Q, F) &= \emptyset \\
\mathcal{A}_I(\mathsf{Wait}(d);R, Q, F) &= \mathcal{A}_I(R, Q, F) \\
\mathcal{A}_I(\mathsf{Work}(l,u);R, Q, F) &= \mathcal{A}_I(R, Q, F)
\end{aligned}$$

$$\begin{array}{rcl}
\mathcal{A}_I(\textsf{Read}(p,v);R,Q,F) & = & \mathcal{R}(p).\alpha R \cup \mathcal{A}_I(R,Q,F) \\
\mathcal{A}_I(\textsf{Write}(p,v);R,Q,F) & = & \mathcal{R}(p).\alpha W \cup \mathcal{A}_I(R,Q,F) \\
\mathcal{A}_I(\textsf{ExtChoice}(P_1,P_2),Q,F) & = & \mathcal{A}_I(P_1,Q,F) \cup \mathcal{A}_I(P_2,Q,F) \\
\mathcal{A}_I(\textsf{IntChoice}(P_1,P_2),Q,F) & = & \mathcal{A}_I(P_1,Q,F) \cup \mathcal{A}_I(P_2,Q,F)
\end{array}$$

$\mathcal{A}_W$ is simply the processing alphabet for the activity:

$$\mathcal{A}_W(P,Q,F) \quad = \quad F.\alpha P$$

We can now give the definition of $\mathcal{NSS}$ and $\mathcal{NAS}$ (first mentioned in Chapter 3). The functions $\mathcal{NSS}$ and $\mathcal{NAS}$ have identical definitions, but are kept separate (until the definition of $\mathcal{A}$) to highlight the distinction between servers and activities.

$$\begin{array}{rcll}
\mathcal{NSS}[\![\, n \,]\!] & = & \mathcal{A}(P,P,\mathcal{F}(n)) & \text{when } n = P \\
\mathcal{NAS}[\![\, n \,]\!] & = & \mathcal{A}(P,P,\mathcal{F}(n)) & \text{when } n = P
\end{array}$$

The '$n = P$' side clause relates to n being the name of a code stub, and P being the program for the code stub.

This is intentionally a very sparse language: it is intended to be as abstract as possible. This is achieved by using simple (and short) translations to Timed CSP (choice) operators (in the case of choice operators), and linear sequences of events for Work, Read and Write.

The only major omission is some form of conditional control structure, *e.g.*

```
if v=20 then P else Q
```

(since the language has sequences and simple eternal loops). However, many programs can be satisfactorily specified using the IntChoice (nondeterministic) operator: if the nondeterministic version satisfies the specification, then a refined version (say, with deterministic tests) will certainly satisfy the specification.

In general, choice can be modelled in several ways. When the value returned by a read is important in deciding the next event, there are several options:

- Give a mathematical expression that can be evaluated without reference to CSP.

- Write some raw CSP, and offer several processes that choose between the values, *e.g.*

  $$(\mathsf{Data.w1}.er.1 \to P_1) \,\square\, (\mathsf{Data.w1}.er.2 \to P_2)$$

  chooses between the returned values 1 and 2.
- Extend the activity language to capture the previous option (or some other alternative).

However, for the purposes of the examples and case studies in this thesis, the language is sufficient to capture the interactions with their environment.

5.5 Example

Recall the example introduced in Figure 3.2 and developed in Section 3.7 (Pages 59 and 71 respectively). The meaning of the system

```
((sy, Controller,
        (sa, Process_A)
        (sa, Process_B)
        (rt, Data, Pool)
        (sp, , Process_A:p1, Data:w1)
        (sp, , Data:w2, Process_B:p1)))
```

was given by the expressions

$$\mathcal{M}[\![\, \mathcal{S} \,]\!] \;=\; \left(\mathcal{C}_P[\![\, \mathcal{SY} \,]\!] \,\|_{\mathcal{C}_W[\![\, \mathcal{SY} \,]\!]} Scheduler\right) \setminus \mathcal{C}_W[\![\, \mathcal{SY} \,]\!]$$

$$\begin{aligned}\mathcal{C}_P[\![\, \mathcal{SY} \,]\!] \;=\; \| \;\{\; & (\mathcal{NAS}_P[\![\, \textsf{Process\_A} \,]\!], \\ & \;\mathcal{NAS}_I[\![\, \textsf{Process\_A} \,]\!]), \\ & (\mathcal{NAS}_P[\![\, \textsf{Process\_B} \,]\!], \\ & \;\mathcal{NAS}_I[\![\, \textsf{Process\_B} \,]\!]), \\ & (\mathcal{NRS}_P[\![\, \textsf{Data}, \textsf{Pool} \,]\!], \\ & \;\mathcal{NRS}_I[\![\, \textsf{Data}, \textsf{Pool} \,]\!]), \\ \} \;& \setminus \mathcal{H}[\![\, \mathcal{SY} \,]\!]\end{aligned}$$

$$\begin{aligned}\mathcal{C}_I[\![\, \mathcal{SY} \,]\!] \;=\; (\quad & \mathcal{NAS}_I[\![\, \textsf{Process\_A} \,]\!] \\ \cup \; & \mathcal{NAS}_I[\![\, \textsf{Process\_B} \,]\!] \\ \cup \; & \mathcal{NRS}_I[\![\, \textsf{Data}, \textsf{Pool} \,]\!] \quad)\end{aligned}$$

$$\setminus \mathcal{H}[\![\mathcal{SY}]\!]$$

$$\begin{aligned} \mathcal{C}_W[\![\mathcal{SY}]\!] &= \mathcal{NAS}_W[\![\text{Process\_A}]\!] \\ &\cup \mathcal{NAS}_W[\![\text{Process\_B}]\!] \\ &\cup \mathcal{NRS}_W[\![\text{Data}, \text{Pool}]\!] \end{aligned}$$

Suppose we have the following as the code stubs for the simple activities:

```
Process_A = Work(5, 10);
            Write(p1, x);
            Self
```

```
Process_B = Read(p1, y);
            Work(10, 30);
            Self
```

Then the value of $\mathcal{NAS}[\![\text{Process\_A}]\!]$ is determined thus:

$$\mathcal{NAS}[\![\text{Process\_A}]\!] = \mathcal{A}(P, P, F)$$

where

$$\begin{aligned} P &= \text{Work(5, 10);} \\ &\quad \text{Write(p1, x);} \\ &\quad \text{Self} \\ F &= \mathcal{F}(\text{Process\_A}) \end{aligned}$$

This gives

$$\begin{aligned} \mathcal{NAS}_P[\![\text{Process\_A}]\!] &= F.sp.(5, 10) \to F.mp \to F.ep \\ &\to \mathcal{R}(\text{p1}).sw.x \to \mathcal{R}(\text{p1}).mw.x \to \mathcal{R}(\text{p1}).ew.x \\ &\to \mathcal{NAS}_P[\![\text{Process\_A}]\!] \end{aligned}$$

Next, we need to determine which IDA p1 is connected to. Now, the term

```
(sp, , Process_A:p1, Data:w1)
```

means that

$$\mathcal{R}(\mathsf{p1}) = \mathsf{Data}$$

and in this instance, we set

$$F = \mathsf{Controller.Process\_A}$$

since this is a unique name. Then, the semantic function $\mathcal{A}_P$ on Page 119 gives us that

$$\begin{aligned}\mathcal{NAS}_P[\![\ \mathsf{Process\_A}\ ]\!] \ &= \ \mu X \bullet \mathsf{Controller.Process\_A}.sp.(5, 10) \\ &\quad \rightarrow \mathsf{Controller.Process\_A}.mp \\ &\quad \rightarrow \mathsf{Controller.Process\_A}.ep \\ &\quad \rightarrow \mathsf{Data}.sw.x \rightarrow \mathsf{Data}.mw.x \\ &\quad \rightarrow \mathsf{Data}.ew.x \rightarrow X\end{aligned}$$

(We write the program in terms of the CSP recursion operator, $\mu X \bullet F(X)$ rather than attempt to repeatedly expand Self.)

The other components of $\mathcal{NAS}[\![\ \mathsf{Process\_A}\ ]\!]$ are calculated similarly:

$$\mathcal{NAS}_I[\![\ \mathsf{Process\_A}\ ]\!] \ = \ \mathsf{Data}.\alpha W$$

$$\mathcal{NAS}_W[\![\ \mathsf{Process\_A}\ ]\!] \ = \ \mathsf{Controller.Process\_A}.\alpha P$$

The $\mathcal{NAS}_W$ expression means 'rename the processing alphabet, αP, using the name Controller.Process_A' (see Pages 98 and 117).

Similarly,

$$\begin{aligned}\mathcal{NAS}_P[\![\ \mathsf{Process\_B}\ ]\!] \ &= \ \mu X \bullet \mathsf{Data}.sr \rightarrow \mathsf{Data}.mr \rightarrow \mathsf{Data}.er.y \\ &\quad \rightarrow \mathsf{Controller.Process\_B}.sp.(10, 30) \\ &\quad \rightarrow \mathsf{Controller.Process\_B}.mp \\ &\quad \rightarrow \mathsf{Controller.Process\_B}.ep \rightarrow X\end{aligned}$$

$$\mathcal{NAS}_I[\![\ \mathsf{Process\_B}\ ]\!] \ = \ \mathsf{Data}.\alpha R$$

$$\mathcal{NAS}_W[\![\ \mathsf{Process\_B}\ ]\!] \ = \ \mathsf{Controller.Process\_B}.\alpha P$$

Now, Section 4.6 (Page 98) provides the information required to define Data fully:

$$\mathcal{NRS}_P[\![\ \mathsf{Data}, \mathsf{Pool}\ ]\!] \ = \ Data\_Proc$$

$$\mathcal{NRS}_I[\![\ \mathsf{Data}, \mathsf{Pool}\ ]\!] \;=\; \mathsf{Data}.\alpha IDA$$

$$\mathcal{NRS}_W[\![\ \mathsf{Data}, \mathsf{Pool}\ ]\!] \;=\; \emptyset$$

where $Data\_Proc$ has alphabet $\mathsf{Data}.\alpha IDA$ and behaves as a $Pool$.

5.6 Support for Modelling a Scheduler

5.6.1 Why Model A Scheduler?

Why would we want to model a scheduler in this framework? Timing is an important part of the models, even though it is not a very visible part of the definitions. If certain timing requirements have been specified, and the scheduler can interfere with the satisfaction of these requirements, then to reason about the system, we must have some means of modelling a scheduler — within this framework.

Arguably, the scheduler's sole purpose is to ensure that the processes complete all their work within a deadline. We will introduce deadlines as a test to be proven towards the end of this section. First, we will identify how our framework supports the modelling of a scheduler so that we can test that these deadlines are supported.

5.6.2 Abstraction

Abstractly, the DORIS systems are being modelled as a number of CSP programs composed using the parallel operator, with carefully chosen alphabets. Each of these CSP programs represents either

- a simple server,
- a simple activity, or
- a route.

From the description of CSP in Section 2.4, recall that CSP assumes that there are no restrictions such as scheduling limits.

In Section 3.6.3, we referred to a process called $Scheduler$. In Section 5.2, a number of events were identified as being important in the modelling of DORIS systems: we now use the events in $\mathcal{C}_W[\![\ \mathcal{SY}\ ]\!]$ to model a scheduler.

This will be in the form of predicates specifying the behaviour of such a scheduler: then the process $\mathit{Scheduler}$ is any process satisfying those predicates.

By construction, the events in $\mathcal{C}_W[\![\ \mathcal{SY}\ ]\!]$ are based on renamings of the processing alphabet, αP.

5.6.3 Initial Approach

In general, we can model the action of a scheduler by placing a scheduler process in parallel with the CSP meaning of the system. This scheduler process can be easily specified with timed failure predicates. In particular, the use of trace (safety) and refusal (liveness) predicates allows statements like these examples:

$$\begin{aligned} \mathit{WorkSafe}(X) \ = & \quad \tau \downarrow X.\alpha P = s^\frown\langle(t_1, X.sp.(l,u)), (t_2, X.mp)\rangle \\ & \Rightarrow \ t_2 - t_1 \geq l \end{aligned}$$

(A safety property: the middle processing event for X must not occur before l time units have passed since the relevant start processing event.)

$$\begin{aligned} \mathit{WorkLive}_{ep}(X) \ = & \quad \tau \downarrow X.\alpha P = s^\frown\langle(t_1, X.sp.(l,u)), (t_2, X.mp)\rangle \\ & \Rightarrow \ X.ep \notin \aleph \uparrow [\max\{t_1 + u, t_2\}, \infty) \end{aligned}$$

(The end-processing event for X must be available either (a) no later than after u time units have passed since the relevant start-processing event, or (b) immediately after the middle-processing event, whichever is the later.)

By defining the scheduler in terms of predicates of this type, arbitrary scheduling policies can be specified. (Note that the liveness predicate concerns only the ep event. Generally, each of the processing events should have an associated liveness predicate, as is the case in the next section.)

This model does not explicitly take into account processes requiring more time when two or more are scheduled together: a more complex definition incorporating a notion of 'time descheduled' can cope with this case.

5.6.4 Descheduling

In this section, it is important to note that we are not attempting to construct a scheduler that meets the constraints (l, u) for a particular triple of processing events. Instead, we are attempting to simulate what happens with a scheduler which is enforcing certain policies.

For a given activity X, suppose that two functions, $\mathit{Working}(X)(s)$ and $\mathit{Desched}(X)(s)$, are defined thus:

$\mathit{Working}(X)(s)$ returns true if an $X.sp$ event has occurred in the trace s without the corresponding $X.ep$:

$$\mathit{Working}(X)(s) = (s\#X.sp > s\#X.ep)$$

$\mathit{Desched}(X)(s)$ is partial on the domain of traces where $\mathit{Working}(X)$ is true, *i.e.* it is only defined for those traces where $\mathit{Working}(X)$ is true. $\mathit{Desched}(X)(s)$ returns the amount of time that the activity X has been descheduled since the most recent $X.sp$ event. This, of course, depends on each individual scheduler strategy.

We now redefine the predicates in the previous section, this time including three predicates for liveness, one for each event:

$$\begin{aligned} \mathit{WorkSafe}(X) \;=\; & \quad \tau \downarrow X.\alpha P = s^\frown\langle(t_1, X.sp.(l,u)), (t_2, X.mp)\rangle \\ & \Rightarrow\; t_2 - t_1 \geq l + \mathit{Desched}(X)(\tau) \end{aligned}$$

(The middle processing event for X must not occur before l time units, plus the time descheduled so far, have passed since the relevant start-processing event.)

$$\begin{aligned} \mathit{WorkLive}_{sp}(X) \;=\; & \quad \mathit{foot}(\langle(0, X.ep)\rangle^\frown\tau \downarrow X.\alpha P) = (t_1, X.ep) \\ & \Rightarrow\; X.sp \;\mathbf{live\ from}\; t_1 \;\mathbf{until}\; \{X.sp\} \end{aligned}$$

(The start-processing event must be available if X is not working.)

$$\begin{aligned} \mathit{WorkLive}_{mp}(X) \;=\; & \quad \mathit{foot}(\tau \downarrow X.\alpha P) = (t_1, X.sp.(l,u)) \\ & \Rightarrow\; X.mp \notin \sigma(\aleph \uparrow [t_1 + u + \mathit{Desched}(X)(\tau), \infty)) \end{aligned}$$

(The middle-processing event for X must be available no later than after $u + \mathit{Desched}(X)$ time units have passed since the relevant start-processing event.)

$$\begin{array}{rcl} \mathit{WorkLive}_{ep}(X) & = & \tau \downarrow X.\alpha P = s ^\frown \langle (t_1, X.sp.(l,u)), (t_2, X.mp) \rangle \\ & & \Rightarrow\ X.ep \notin \sigma(\aleph \uparrow [\max\{t_1 + u + \mathit{Desched}(X)(\tau), t_2\}, \infty)) \end{array}$$

(The end-processing event for X must be available either (a) no later than after $u + \mathit{Desched}(X)$ time units have passed since the relevant start-processing event, or (b) immediately after the middle-processing event, whichever is the later.)

5.6.5 Example

Suppose we take the previous example (Section 5.5, Page 121), and decide that any event may occur at any time, except that when Process_B is working, Process_A is not allowed to do anything.

We define the $\mathit{Desched}$ function for Process_A, which measures the amount of time that Process_A has been descheduled due to Process_B working since it last started:

$$\mathit{Desched}(\mathsf{Process\_A})(s) = \sum_{s' = x ^\frown \langle (t_1, e_1), (t_2, e_2) \rangle ^\frown y} (t_2 - t_1) \cdot \delta(f(x ^\frown \langle (t_1, e_1) \rangle))$$

where

$$\begin{array}{rl} & s = z ^\frown s' \\ \wedge & s' = \langle (t_0, \mathsf{Process\_A}.sp) \rangle ^\frown z' \\ \wedge & \mathsf{Process\_A}.sp \notin \sigma z' \end{array}$$

and

$$\begin{array}{rcl} f(w) & = & \mathit{last}(w \downarrow \mathsf{Process\_B}.\alpha P) = \mathsf{Process\_B}.sp \\ & \vee & \\ & & \mathit{last}(w \downarrow \mathsf{Process\_B}.\alpha P) = \mathsf{Process\_B}.mp \end{array}$$

δ is the Krönecker delta, and is defined as

$$\delta(e) = \begin{cases} 1 & \text{if } e \text{ is true} \\ 0 & \text{otherwise} \end{cases}$$

$Desched$(Process_A) considers each pair of events since (and including) the most recent Process_A.sp. The full trace leading up to the first of each of the pair of events is checked to see if Process_B is working (using the function $f(w)$). If $f(w)$ is true, the difference in times of the pair of events is added to the descheduled time total.

$Desched$(Process_A) is simply a way of counting the amount of time that both Process_A and Process_B were 'working' at the same time.

This function is partial on the domain of traces where $Working$(Process_A) is true (*i.e.* it is only defined for traces where $Working$(Process_A) is true). This is not a problem, since we only need to calculate it when between sp and ep for Process_A.

Finally, the timed trace and refusal predicates for Process_A result by inserting the definition of $Desched$(Process_A) into the four predicates on Pages 126 and 127: $WorkSafe$, $WorkLive_{sp}$, $WorkLive_{mp}$, and $WorkLive_{ep}$.

5.6.6 Deadlines

The predicates described above can add substantially to the burden of proof in a system by adding much more detail. Why would we want to do this?

Suppose that a system must generate a response to an input within a particular *deadline*. We can use the predicates above to test whether or not the system can guarantee the deadline. This sort of requirement is common in embedded systems.

Alternatively, an additional parameter could be added to the Work keyword, say d, a deadline. This is distinct from the parameters l and u already introduced, which indicate how much time the work will need to be carried out on an otherwise unloaded system.

This would impose a proof obligation on the final combination of scheduler and system to prove that the work would always be completed within that deadline, no matter what other work occurs. This could then be used as part of rely/guarantee proofs, *e.g.* process A guarantees to finish work in 15 seconds, process B guarantees to finish work in 10 seconds, so the sequential combination of the processes can be guaranteed to finish within 25 seconds.

None of this is meant to be a replacement for scheduler theory. Instead, it is meant to simulate the behaviour of a scheduler on the timing responses of the system being modelled within the framework of the rest of the thesis.

5.6.7 Untimed Scheduler Predicates

Timing is not required to model every problem. Sometimes, we will need some untimed failures predicates to ensure that the servers and activities can make progress. These predicates are similar to the IDA untimed refusal predicates (Page 91).

$$\begin{aligned} WorkLive_{sp}(X) &= \quad (tr \downarrow X.\alpha P = \langle\rangle \vee last(tr \downarrow X.\alpha P) = X.ep) \\ &\quad \Rightarrow X.sp \notin rf \\ WorkLive_{mp}(X) &= (last(tr \downarrow X.\alpha P) = X.sp) \Rightarrow mp \notin rf \\ WorkLive_{ep}(X) &= (last(tr \downarrow X.\alpha P) = X.mp) \Rightarrow ep \notin rf \end{aligned}$$

When a particular condition is met (*i.e.* the predecessor event has occurred), the next event must not be refused.

Part III

Mechanical Implementation

Chapter 6

Mechanical Support

6.1 Overview

In previous chapters, this thesis has given a semantics to the DORIS notation. For this semantics to be industrially useful, the theory must be exploitable to produce results. Typically, this means that the solution must scale up, so the theory requires tool support.

In this chapter, we discuss the possible approaches to mechanizing our work so far, and describe why we adopted the tools and techniques reported in the rest of the chapter.

The issue of mechanical support in this thesis can be viewed as two parts:

- how to turn a textual description of a DORIS system into Timed CSP; and

- how to analyse the resulting Timed CSP.

6.2 Rationale

To a certain degree, the choice of tools is influenced by the semantic domain that we have chosen, and, similarly, our choice of semantic domain is influenced by the tools available for particular semantic domains (see Section 3.1.2, Page 54).

In this section, we will start from the assumption that we are working in the Timed CSP domain, and outline some of the tools that are available. We then indicate why we have chosen this approach.

6.2.1 Related Work

With the exception of FDR, most approaches to mechanical support for CSP are based on theorem provers.

Bryans, Dutertre, and Schneider have embedded and used a model of untimed traces CSP in PVS to verify security protocols [11, 26, 27]. Camilleri has produced a mechanization of three untimed semantic models in HOL [13], and Thayer has produced an embedding in IMPS using monoids [118]. Tej and Wolff [117] have embedded CSP in Isabelle/HOL, and discovered a problem with the previously accepted semantics of failures-divergence CSP.

Of the work so far on embedding CSP in a theorem prover, only Thayer has addressed timing, by proposing a monoid model of Timed CSP. This work has not been developed any further than a suggestion. The other embeddings have been used only for very tightly defined application domains (*e.g.* security protocols) or for very small problems.

The PVS models referred to above are very similar in style to the Isabelle/HOL models: they both consider the observations of the process as the primary objects of interest. Camilleri took a different approach in that the algebraic structure of CSP was the primary focus.

6.2.2 Choice of PVS and FDR

As the work reported in this thesis developed, it became clear that using both a theorem prover and a state space exploration tool would provide complementary approaches. The dual approach of theorem prover and model checker allows the use of sophisticated mathematical concepts (*e.g.* induction over infinite domains) with the theorem prover, and brute force checks of finite systems with the model checker. (Later in the thesis, we report a result from FDR that can be fed back into PVS — see Section 7.4, Page 169.)

The choice of FDR was easy: there is a lot of experience in applying FDR to CSP problems. Indeed, FDR was explicitly developed for the purpose.

The choice of theorem prover, and how to model CSP, was slightly harder. Most approaches are based on modelling the observations of a process; only Camilleri differed here. In this thesis, the theorem proving approach models observations: this approach has had some success, and it seems sensible to develop it further.

On the grounds that more work had been carried out in PVS, we chose PVS, although HOL would have probably been at least as adequate.

6.2.3 Other Tools

Other tools were considered for this work.

The NASA guidebooks [75, 76] include a comprehensive list of formal methods tools (Appendix B of Volume 1). Generally, these formal methods tools are either state space exploration tools (a graph is constructed for the system concerned, which is then subjected to constraint and reachability tests) or theorem provers (which more closely follow rigorous proofs, but often at a far greater level of detail). Some state space tools also include simulators, which can follow a particular execution of the system being modelled.

In particular, several tools warrant further comment:

- Concurrency Workbench (CWB) [120] analyzes CCS [74] systems using model checking. It can also handle a discrete time variant of Temporal CCS (using 'idle' transitions where time passes). In style, it is very similar to FDR (except for FDR's X Windows interface).

  However, there are sufficiently many minor semantic differences between CSP and CCS that no benefit would be gained by using CWB.

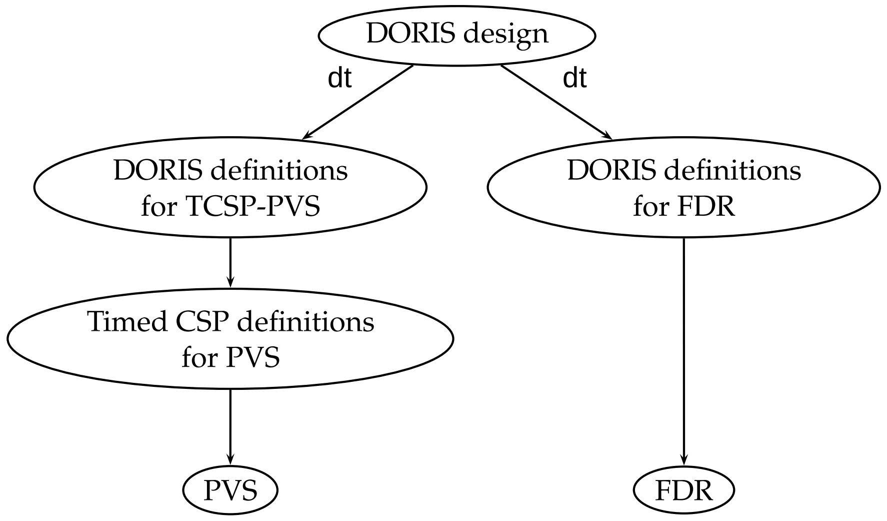
- UPPAAL [7] can perform safety and bounded liveness checks on real-time automata. It is a model checker, and claims to deal with the state-space explosion by reducing the verification problems to linear constraint solutions.

  At this time, it is a much less stable tool than either FDR or PVS: we are aiming at providing an 'industrial-strength' approach. In the future, it may provide a further complementary view of problems.

- Isabelle [124] and HOL [123] are two theorem provers from Cambridge. Isabelle is a generic theorem prover where logics are defined by specifying their syntax and inference rules, and HOL is an interactive prover for higher-order logic based on ML.

6.3 Summary of Mechanical Support

A bespoke tool, named 'dt', handles the reading and checking of DORIS descriptions. It then produces two groups of output, one for a PVS implementation of Timed CSP, and another for FDR. This is illustrated below.


DORIS design
dt
dt
DORIS definitions for TCSP-PVS
DORIS definitions for FDR
Timed CSP definitions for PVS
PVS
FDR

Our complementary tools have advantages and disadvantages:

PVS
- requires an embedding of Timed CSP;
- requires the definition of DORIS in PVS-CSP; and
- is a theorem prover's assistant.

FDR
- requires the definition of DORIS in FDR;
- the predicate specifications must be converted into programs; and
- is a state exploration system.

We will discuss these points more fully in the rest of this chapter.

6.4 PVS

PVS is a mechanical theorem prover [15, 79, 80, 96, 105]. Abstractly, it takes sequents of the form

$$A_1 \wedge \ldots \wedge A_M \vdash C_1 \vee \ldots \vee C_N$$

where A_i are antecedents, and C_j are consequents. By the application of a sequence of commands (a *proof script*, either interactive, or in a 'batched' mode), the aim is to transform the sequent until at least one consequent is 'obviously' true.

Commands include 'LEMMA' which introduces a new antecedent for the lemma concerned, and 'GRIND', a strategy which tries a series of other commands which commonly result in a successful proof.

PVS is supplied with a library of definitions and lemmas concerning basic mathematical objects, including sets and sequences. These definitions are grouped into *theories*. PVS users can create other theories: indeed, theories are the basic building block in the theorem prover.

6.4.1 Timed CSP

The embedding reported in this thesis draws much inspiration from the work by Bryans, Dutertre, and Schneider referenced above.

Four semantic models of Timed CSP are constructed within PVS:

- untimed traces (UT);
- untimed failures (UF);
- timed traces (TT); and
- timed failures (TF).

Each semantic model has a group of theories constructed for it, which draws on another two theories containing basic definitions about observations and events.

The PVS theory hierarchy for these theories is included in Figure 6.1 (Page 138). At the bottom of this hierachy are some of the predefined theories from the PVS distribution. These provide facilities like type extension and restriction, maps (filters), and set, sequence and list manipulation.

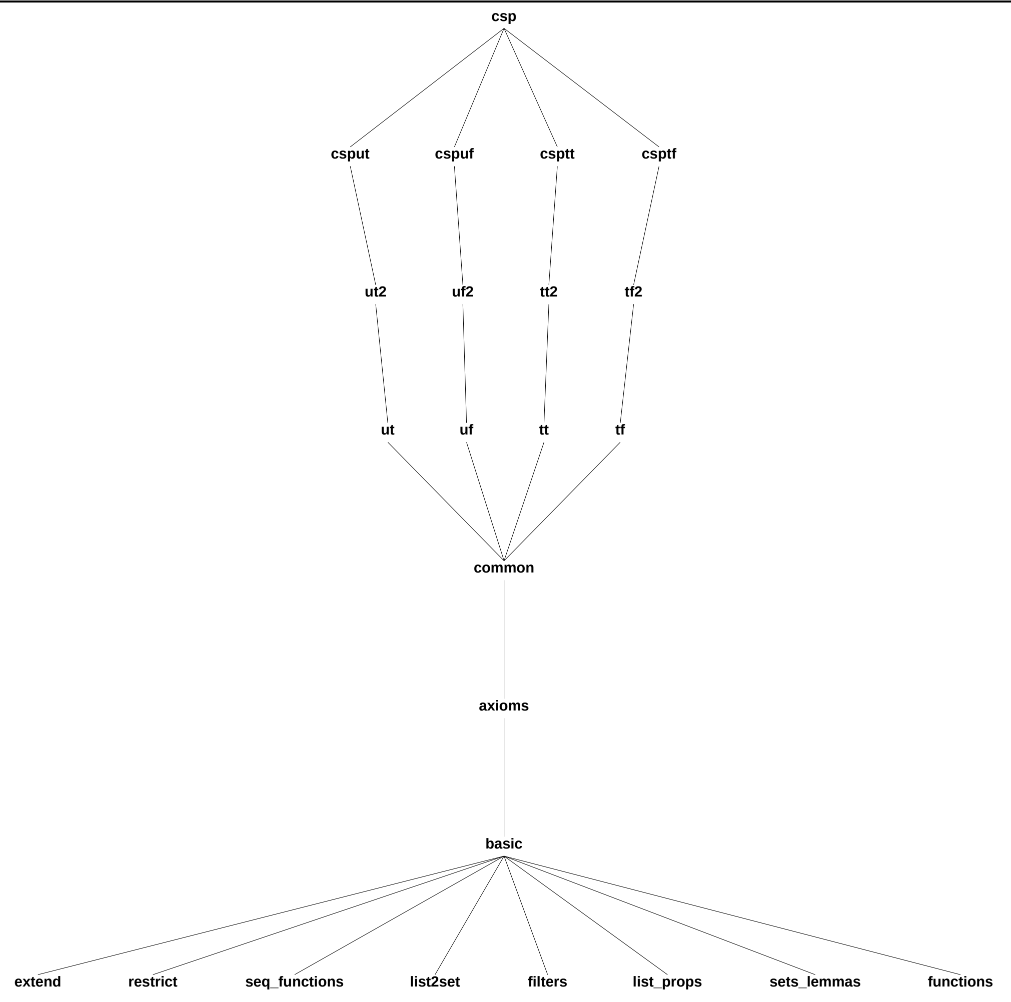


Figure 6.1: PVS theory hierarchy for the TCSP semantic models

These are used ('imported') in the next theory up, called basic. This defines the basic concepts in the Timed CSP theory: the observations; what an event is; what timed events and refusals are. The next theory is axioms, which reasons about some of the constructs in basic. The theory common ties these two together so that they can be more easily imported into the per-semantic model theories.

Each semantic model provides exactly the same operators. This is so that the semantic model for a particular problem can be changed by altering a single definition (which indicates the current semantic model). These operators are defined in the four theories ut, uf, tt, and tf. The operators provided are listed in Appendix C.2 (Page 232).

The next layer of theories (with '2' suffixed) provide definitions of satisfaction for each theory, and some algebraic laws. The remaining theories provide a consistent interface and bundle the CSP theories together.

We will touch upon some elements of the untimed traces theory when we discuss PVS type-correctness conditions in Section 6.4.4 (Page 144). In the next section (Section 6.4.2), we describe how these Timed CSP theories were constructed.

(We will outline the time and effort required to construct both the Timed CSP and DORIS theories in Section 6.4.5, Page 152.)

6.4.2 Timed CSP Theory Construction

In the previous section, we have identified the four semantic models of Timed CSP for which we have constructed PVS theories in this thesis. These are:

- untimed traces (UT);
- untimed failures (UF);
- timed traces (TT); and
- timed failures (TF).

Similarly, in Appendix C.2 (Page 232), we have identified the Timed CSP operators available in each of these theories.

The theories themselves were constructed by building the UT theory first; this is the simplest theory here, and was used to resolve initial issues concerning the usability of the interface described in the appendix.

The second theory constructed was the TF theory, the most complex. From this, we can construct the two remaining theories with relatively little effort.

Therefore, in this section, we use examples from the UT and TF theories.

Events and Alphabets

In Section 2.4.1 (Page 34), we introduced the notion of events. The PVS theories are parametrised by a set of events, which includes a distinguished event `success` (representing $\checkmark$).

However, we do not use an alphabetized model of processes. This is really a matter of taste: some entities are easier to describe in an alphabetized model; some are harder. A similar comment applies to proofs about such entities.

The choice made in this instance is that the interface is clearer in the non-alphabetized model.

Definition of `Process`

When we refer to `Process`, we really use one of four definitions, depending on the semantic model. A general feature, however, is that a process is defined as a set of observations satisfying some properties:

$$\texttt{Process} = \{\texttt{P : setof[Obs]} \mid \texttt{prop1(P)} \wedge \ldots \wedge \texttt{propN(P)}\}$$

Each `prop`n constrains the possible observations.

The observations for each model are described in Sections C.1.2 and C.1.5 (Pages 230 and 231 respectively). The table below summarises the properties on sets of observations for the UT and TF models:

UT	TF
ProcessUT_Base	ProcessTF_Base
ProcessUT_PrefixClosed	ProcessTF_InfOrder
ProcessUT_SuccessClosed	ProcessTF_SuccessClosed
	ProcessTF_TimeForward
	ProcessTF_Complete

Each 'property' is a PVS predicate taking a set of observations as an argument.

The `Base` property is that the 'bottom' observation is a member of every process. In the UT model, this is the empty trace; in the TF model, this is the pair consisting an empty timed trace and an empty timed refusal set.

The `PrefixClosed` property requires that if a trace is an observation of a process, then all prefixes of that trace are observations of that process.

`SuccessClosed` requires that success, if it appears, is the last event in a trace.

`InfOrder` uses the information order operator ($\preccurlyeq$) from Schneider's text [102]:

$$(\tau', \aleph') \preccurlyeq (\tau, \aleph) \Leftrightarrow \exists \tau'' \bullet \tau = \tau' \frown \tau'' \wedge \aleph' \subseteq \aleph \uparrow [0, begin(\tau''))$$

In the TF model, if $(\tau, \aleph)$ is an observation of the process, then any $(\tau', \aleph')$ such that $(\tau', \aleph') \preccurlyeq (\tau, \aleph)$ is also an observation of that process.

`TimeForward` applies to timed traces: this ensures that time goes forward as the trace extends.

`Complete` is a more complicated statement intended to ensure that there is enough timed refusal information in the process for consistency: all events are either available or refusable. Like other parts of the TF model, this is drawn from Schneider's text [102]. This breaks down into a statement referring to two clauses of a conjunction (`ProcessTF_Complete1` and `ProcessTF_Complete2`).

Example

PVS allows a natural translation from 'written' mathematics, *e.g.* `InfOrder`, as described above, is written in PVS thus:

```
ProcessTF_InfOrder(P : ObsSetTF) : bool
  = FORALL (f : (P), fp : ObsTF) :
      InfOrder(f, fp) IMPLIES P(fp)
```

`P` is a set of observations in the TF model (`ObsSetTF`) representing a process. The predicate contains a universal quantification over observations of `P` (denoted `f`) and all possible observations (denoted `fp`). `P(fp)` means '`fp` is a member of `P`', and`InfOrder` is defined thus:

```
InfOrder(fp, f : ObsTF) : bool
  = EXISTS (spp : TimedTrace) :
      ( TimedTrace(f) = append(TimedTrace(fp), spp)
       AND
        subset?(TimedRefusal(fp),
                DuringBefore(TimedRefusal(f),
                             tBegin(spp)     ))   )
```

in the basic theory.

When all the definitions referred to above have been completed, the definitions of ProcessUT and ProcessTF can be written:

```
ProcessUT : TYPE
  = { P : ObsSetUT | ProcessUT_Base(P)
                   AND ProcessUT_PrefixClosed(P)
                   AND ProcessUT_SuccessClosed(P) }

ProcessTF : TYPE
  = { P : ObsSetTF | ProcessTF_Base(P)
                   AND ProcessTF_SuccessClosed(P)
                   AND ProcessTF_TimeForward(P)
                   AND ProcessTF_InfOrder(P)
                   AND ProcessTF_Complete(P) }
```

Process Definitions

At this point, we have a definition of Process. We can then define the base processes[1] in terms of that definition.

The simplest example uses Stop in the UT model:

```
StopDefnUT : ObsSetUT = { t : ObsUT | t = null }
```

This simply defines a set of UT observations consisting of only the empty trace ('null'). Next, we define three lemmas, one for each property of the definition of ProcessUT:

```
StopUT_Base : LEMMA
  ProcessUT_Base(StopDefnUT)
```

[1]The 'base' processes, as distinct from 'derived' processes, are those appearing in Appendix C.2.1 (Page 232).

```
StopUT_PrefixClosed : LEMMA
  ProcessUT_PrefixClosed(StopDefnUT)

StopUT_SuccessClosed : LEMMA
  ProcessUT_SuccessClosed(StopDefnUT)
```

This illustrates a standard approach adopted in the PVS model: regular naming. This then allows strategies to be written to exploit this naming structure to automate some proof elements. Finally, we define `Stop`:

```
Stop : ProcessUT = StopDefnUT
```

An alternative example concerns the external choice operator in the TF model. This follows the same structure as the example above: first we define the process in terms of a set of observations. In this example, the operator takes two operands (`P` and `Q`).

```
EChoiceDefnTF(P, Q : ProcessTF) : ObsSetTF
  = { b : ObsTF |
         (P(b) OR Q(b))
       AND
         intersection(P, Q)(null,
            DuringBefore(TimedRefusal(b),
                        tBegin(TimedTrace(b))))  }
```

i.e. an observation `b` in `EChoiceDefnTF(P, Q)` is either an observation of `P`, or an observation of `Q`, or until the choice is resolved, an observation of both. We then identify the lemmas matching the process property definitions:

```
EChoiceTF_Base : LEMMA
      FORALL (P, Q : ProcessTF) :
         ProcessTF_Base(EChoiceDefnTF(P, Q))
```

(We have omitted the remaining lemmas.) Finally, the operator `EChoice` is defined:

```
EChoice(P, Q : ProcessTF) : ProcessTF
      = EChoiceDefnTF(P, Q)
```

From these examples, we can see that there are many subsidiary definitions used to support the rich structure of Timed CSP.

Proofs of Process Definitions

In Section 6.4.4, we describe the PVS notion of a type-correctness condition. The result of this is that we have to prove that every process definition is in fact a process. For example,

```
  Stop : ProcessUT = StopDefnUT
```

generates the TCC that

```
Stop_TCC1: OBLIGATION
      ProcessUT_Base(StopDefnUT)
          AND ProcessUT_PrefixClosed(StopDefnUT)
            AND ProcessUT_SuccessClosed(StopDefnUT);
```

(This proof is illustrated in Section 6.4.4.)

Note that we do not have to prove the TCCs to use the definitions; however, a proof relying on the definition is not complete until we have done so. This is useful when experimenting with alternative definitions.

6.4.3 DORIS

We now need to construct a PVS theory for DORIS on top of the CSP theories.

The PVS theory hierarchy for the DORIS theories is included in Figure 6.2 (Page 145). The first theory to note is dorisdefs, which defines the events for a DORIS system: *i.e.* the IDA and work events. Also defined are server events: these are like the IDA events, but cover the case where the path connects to a simple server, not a route.

dorisidaspecs, dorisproc, and dorisservers define the IDA specifications (*i.e.* the constraints on IDA events for particular routes), the constraints on work events, and the constraints on server events respectively.

The two theories dorisidaimpl and dorisprocimpl include implementation details and lemmas about the specifications. Finally, doris bundles all the DORIS definitions together.

6.4.4 TCC Proofs

We have overlooked one important feature of PVS: it is strongly-typed, but the type system is undecidable (due to the inclusion of general predicate

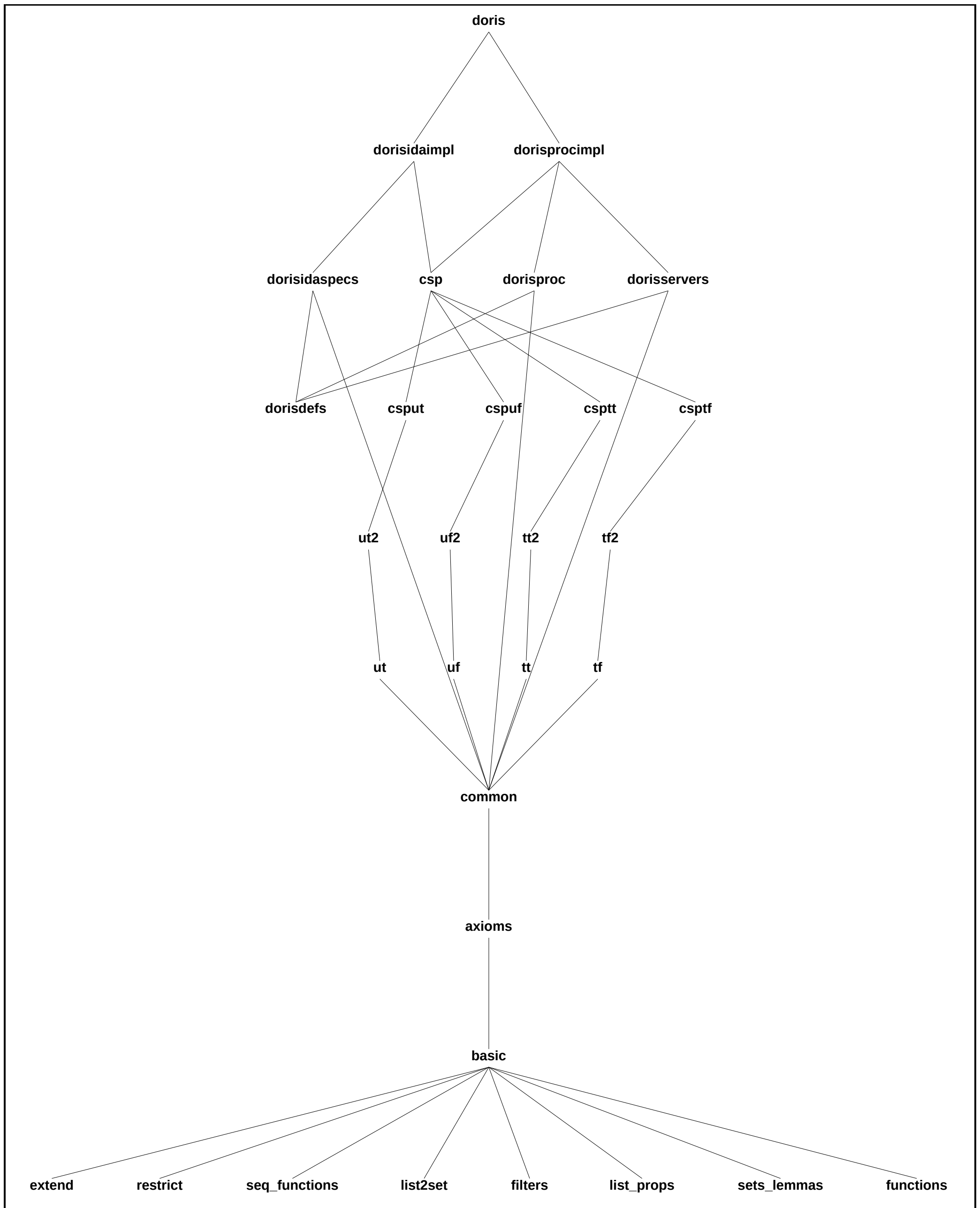


Figure 6.2: PVS theory hierarchy for DORIS

subtypes). This means that to show type-correctness, the prover will generate *type-correctness conditions* (TCCs, or proof obligations). *Termination* TCCs are generated when recursive definitions are encountered: typically these are required to ensure that the recursive definition is total (as PVS does not admit partial functions).

In many cases, TCCs can be automatically discharged by PVS. The more complex the definition, the less likely this is. Most of the TCCs from the DORIS theories concern either recursion or proving that a particular named event is not the termination event, both of which are easily discharged. The TCCs that cause most problems at first are those concerned with defining CSP processes.

$Stop$ in the UT model

The simplest example is to prove that (within the UT model) the process $Stop$ satisfies the constraints placed upon processes.

Processes in the UT model can be defined as any set of observations such that

1. the 'bottom' observation, $\langle\rangle$, is in the set;
2. the set is 'prefix closed' (all prefixes of a trace in the set are also in the set); and
3. the set is 'success closed' (if success appears in the trace, then it appears exactly once, and is the last event in the trace).

Thus there are three parts to be proved in each case.

Since the process $Stop$ is defined as being the set of observations containing the singleton element $\langle\rangle$, the three parts of the TCC are trivially proved.

Least Fixed Point operator in the UT Model

A more complex construct, the least fixed point (LFP) operator, can be used to illustrate PVS's proof capabilities. The following PVS definitions implement this operator.

IterateUT carries out n unwindings of the process map F.

```
IterateUT(F : PMapUT, n : nat) : RECURSIVE PMapUT =
    (LAMBDA (X : ProcessUT) :
        IF n = 0 THEN
            X
        ELSE
            F(IterateUT(F, n-1)(X))
        ENDIF)
    MEASURE n
```

(PMapUT is the type from ProcessUT to ProcessUT.)

LFPDefnUT uses IterateUT to define the least fixed point of this function.

```
LFPDefnUT(F : PMapUT) : ObsSetUT
    = { b : ObsUT |
        FORALL (n : nat) : IterateUT(F, n)(Chaos(Sigma))(b) }
```

The definition says that a trace, b, is a member of the set of observations LFPDefnUT if, for all values of n, b is a member of the set of observations that results from n unwindings of IterateUF applied to the most nondeterministic process, Chaos(Sigma). (Sigma is the set of all events 'in the universe'.)

Finally, LFP is defined as LFPDefnUT.

```
LFP(F : PMapUT) : ProcessUT = LFPDefnUT(F)
```

(This step-wise definition allows us to introduce lemmas about the definition more easily.)

The proof that LFPUT is indeed a process is split into three parts. (Figures 6.3–6.5, on Pages 149–151, are the PVS proof trees illustrating each proof.) The sketch proof for each part is as follows:

1. (Base element) Proof is by induction. The base case (zero unwindings) is trivial, because $Chaos$ is a process. The inductive case then relies on the fact that if the base element is a member of the process

represented by the jth unwinding, then it is a member of the $j + 1$st unwinding.

2. (Prefix closure) The proof is again by induction. The second split on each branch is due to more TCCs being created in the course of the proof.

3. (Success closure) This proof is not by induction. It works by showing that the LFP process is success closed because the zero'th unwinding is the $Chaos$ process which is itself success closed.

The proof trees show the overall structure of the *formal* proof, but obscure the detail needed for the overall *rigorous* proof to be clear. Indeed, many of the steps in the tree are simple manipulations, *e.g.* substituting one expression for an equivalent one.

It is important to note that the rigorous proofs are simple proofs, yet the formal proof is quite involved: complex rigorous proofs are very complex formal proofs.

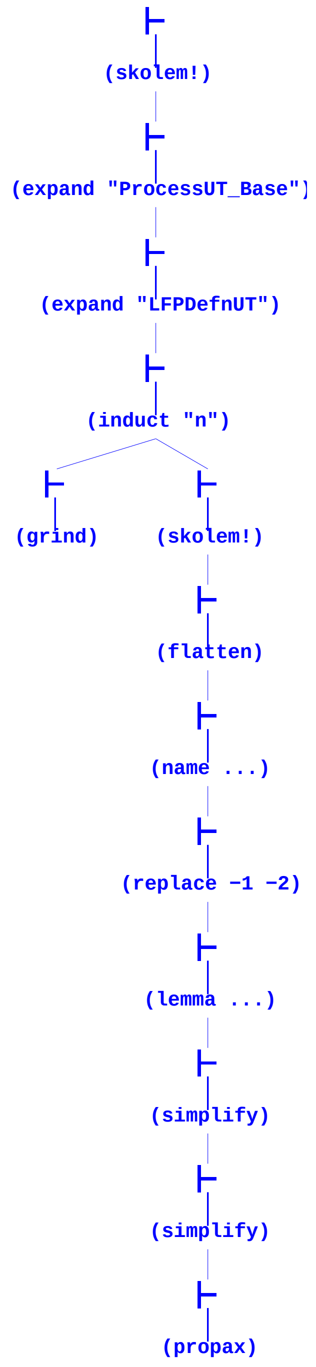


Figure 6.3: Base element proof tree for LFPUT

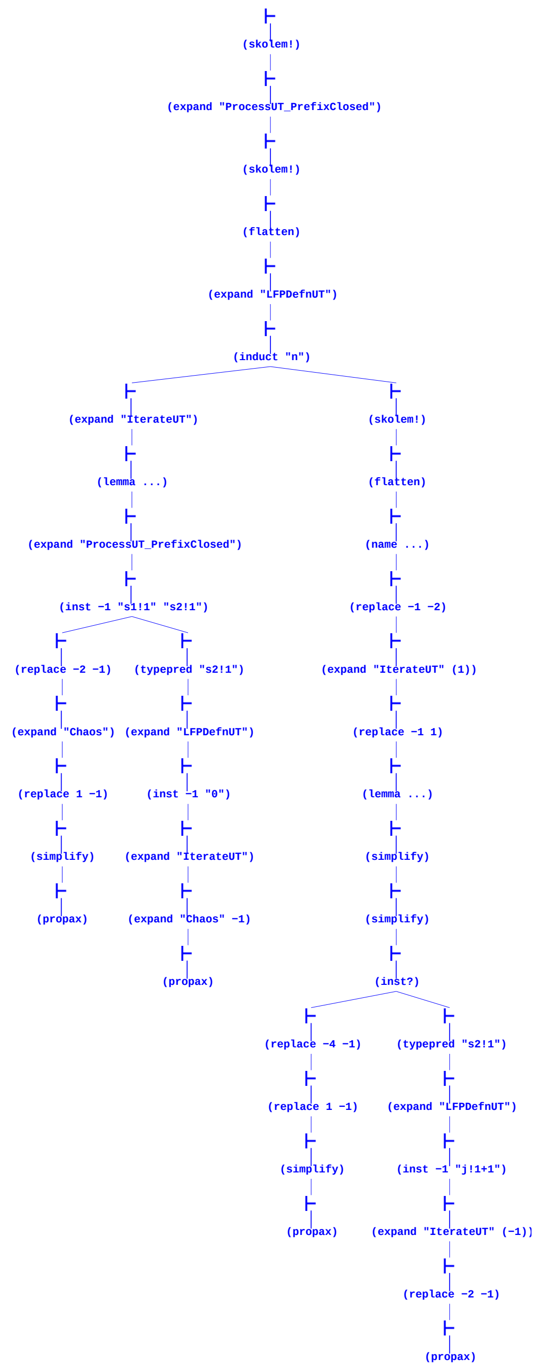


Figure 6.4: Prefix closure proof tree for LFPUT

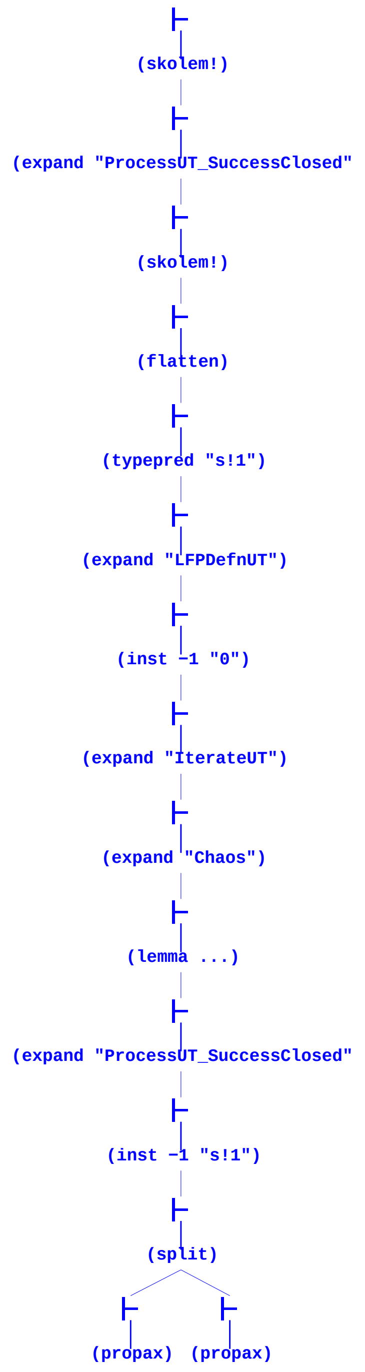


Figure 6.5: Success closure proof tree for LFPUT

6.4.5 Time and Effort Required

The Timed CSP theories are not insignificant in size. The following table gives (an incomplete) list of the theories to illustrate this.

Theory	Size (in lines of code)
basic	421
axioms	328
ut	435
uf	617
tt	520
tf	798
ut2	206
tf2	214
csppmr	200

The last entry, csppmr, does not appear in the hierarchy (in Figure 6.1, Page 138), because it is directly imported by a 'client' theory which requires parameterised mutual recursion in CSP. (We will find a use of this theory on Page 165.)

For comparison, the DORIS theories have the following sizes:

Theory	Size (in lines of code)
doris	127
dorisdefs	59
dorisidaimpl	210
dorisidaspecs	510
dorisprocimpl	21
dorisproc	52
dorisservers	198

This is relatively meaningless: it provides a very rough measure of how much work was involved. A better measure concerns the time needed to construct these theories.

The Timed CSP theories took around two months to define, using Schneider and Roscoe's texts [95, 102] as the primary reference. Similarly, the DORIS theories took three weeks.[2]

[2]In fact, they took longer — but much development of the underlying theory took place during this time.

Once defined, these do not need defining again. Also, the proofs, once discharged, do not need performing again, unless something they depend upon changes (*e.g.* an unproven lemma is found to be false).

As this thesis aims to demonstrate whether this is a tractable method for industry, we have not carried out all the proofs required. Due to the size of the Timed CSP and DORIS theories, this is not a trivial task. We estimate that at least one man-year (maybe several years) of time is required to rework all the theories and ensure that all proof obligations are discharged.

Fortunately, we can accept many of the lemmas 'on trust'. They have been rigorously proven in the past (although we must be careful of even these proofs). However, we must be aware of their strictly unproven status (in the PVS sense).

6.5 FDR

FDR

- is a model checker: it requires little direction from the user (but is susceptible to state space explosion);
- it already uses a widely accepted variant of CSP syntax [98]; and
- has a (relatively) friendly X Windows interface for debugging processes;

but

- FDR does not model real (*i.e.* non-sequential) time;
- it only reports problems and counter-examples — it cannot give an output to demonstrate why a refinement holds;
- it has a flat naming structure: it does not have a notion of 'scoping'; and
- all CSP processes must be represented as programs: predicates can not be used.

It can be a very useful and effective tool.[3]

[3]Thanks are due to Bryan Scattergood of Formal Systems Europe Limited, who offered much assistance with the mechanical aspects of FDR.

6.5.1 Time

Although FDR does not handle *real* time, it can model the passage of discrete intervals of time by using an event that we will call $tock$, in common with other treatments of this problem [95].[4]

The complication here is that $tock$ must be represented in every process where time is allowed to pass. This significantly complicates the definitions of processes.

At its simplest, an untimed process P continues regardless of the passage of time: $P|||(\mu X \bullet tock \to X)$. If the passage of time can alter the properties of the process, then the program must be altered to allow for time to pass 'inside' the original program.

The PVS implementation of DORIS can handle this by simply using the delayed prefix operator (derived from $Wait$ and $\to$), and a timed semantic model. The FDR implementation has to count $tock$s in the definition of $Wait$: note that once the wait delay has passed, more time is permitted to pass.

6.5.2 DORIS

In the same way that PVS requires a definition of DORIS theories, we define a separate file that is included in the dt-generated file for FDR . This contains all the basic definitions that we use, *i.e.* definitions of IDAs, servers, and processing events.

6.5.3 Program Definitions of IDAs

Earlier, we noted that we must give specifications in the form of programs. The IDAs were defined in the form of predicates in Section 4.5 (Page 89).

Figures D.1–D.4 (Pages 238–241 of Appendix D) give the state machines (and thus the programs) for the four main groups of IDA. There are three points to note here:

- The constant IDA state machine (Figure D.1) allows the writer to write, but without affecting the reader. The definition given on Page 92 permits refusal of writer events.

[4]Here, we take 'real time' to mean that the domain representing time is the set of real numbers, $\mathbb{R}$, as opposed to 'discrete time', represented by the integers, $\mathbb{Z}$.

- The pool IDA state machine does not make it clear that the value returned by a read is a nondeterministic (internal) choice. When written as a CSP program, this can be made explicit.
- Timing is not explicitly represented on the state machines. The predicate specifications indicate that a maximal delay $\hat{t}$ is permitted on each transition ($\hat{t}_0$ for the first start-read and start-write).

Generally, programs such as these are not the most general program that could satisfy a particular predicate specification; however, this is generally not a problem in practice. (Typically, the programs that are defined are 'sufficiently general'.) It is important to be aware of the problem so that counter-examples can be checked more thoroughly.

In Section 7.1 (Page 159), we give a specific example relating to this issue.

6.5.4 Time and Effort Required

There is one FDR include file for each dt format: these are several hundred lines of code each. They took slightly less time to initially develop than their PVS counterparts: around two weeks (the previous remark about ongoing development increasing this time applies here). However, debugging the FDR definitions proved to be time-consuming, taking another week.

6.6 Bespoke Tool for Translating DORIS to Timed CSP

'dt' has been mentioned several times before in this thesis. It is a bespoke program written by the author in Ada 95 (using the Ada Core Technologies GNAT compiler [122]) as part of the work for this thesis.

Inputs to dt are flat text files in a particular format. They can define either DORIS designs, or supply the code stubs for those designs. Internal consistency checks are then carried out based on the constraints in Section 3.5.

Output takes several forms:

Plain text A format intended for human analysis;

PVS Four versions for PVS:

- untimed traces (UT) model of the design layer (*i.e.* no scheduling information);
- UT model of the distribution layer (*i.e.* with scheduling information);
- timed failures (TF) model of the design layer; and
- TF model of the distribution layer.

FDR Four versions for FDR:

- untimed model of the design layer;
- untimed model of the distribution layer;
- timed model of the design layer; and
- timed model of the distribution layer.

The four versions for both PVS and FDR are intended for different 'views' of similar applications. We may only be interested in an untimed safety model; there is no need to include all the extra detail in the timed failures model. Similarly, depending on which part of the software development cycle we are up to, we may or may not need to incorporate the scheduler framework.

The output routine has been written with the intention of producing other output formats as needed. This allows for use of other tools that support a similar view of the world, *e.g.* the concurrency workbench or UPPAAL. These output routines have been complicated by the slight differences between the 'blackboard' semantics implemented in PVS, and the FDR semantics.[5]

MADGE is BAe's bespoke CASE tool for handling MASCOT-3 and DORIS designs. dt has been developed to show that mechanical generation from a design file can produce the inputs to other tools. Ideally, MADGE would generate code in a form ideal for input to a modified version of dt. This would allow software engineers to produce their design

[5]In particular, the parallel operators are used slightly differently, but to achieve the same effect. This has the useful side-effect of allowing slightly easier proof strategies in PVS. In the PVS version, we use the hybrid parallel operator; in the FDR version, we use the network parallel operator (and include the work events, $\mathcal{C}_W$, in the alphabet interface).

in MADGE, and produce either Ada code (as they can at the moment), or Timed CSP for analysis of the design.

Further details of dt are given in Section B.3 (Page 228).

Chapter 7

IDA Analysis

The purpose of this chapter is to both test the mechanical support for CSP, and explore the properties of the DORIS IDAs. This is not just exploration for its own sake: if we know more about how the IDAs behave, then we can use those results as part of larger results about systems.

A number of proofs and reports of FDR checks are given here. When PVS is used, a summary of the proof is given. This chapter also serves to demonstrate just how much work is required to formalize even a simple rigorous proof.

7.1 Predicate and Program Specifications: The Pool

When there are two specifications of the same object, say S_D for the predicate specification, and S_G for the program specification, we should prove that S_D is refined by S_G:

$$\forall f : \mathcal{O}(S_G) \bullet S_D(f)$$

i.e. every observation f admitted by the program S_G satisfies the predicate S_D.

The converse is not necessary: what we need to show is that S_G is the most general program that satisfies S_D. This is not trivial, and for the pool, a solution has not been found.

This causes a complication if we use S_G in place of S_D, and find that a proposed implementation, say I, does not satisfy S_D. We must check that each counter-example (for S_G) is also false for S_D.

We now illustrate this using the pool IDA as an example.

7.1.1 Untimed Traces Model

We want to show that $Pool_D$ is satisfied by $Pool_G$:

$$Pool_G \textbf{ sat } Pool_D$$

i.e.

$$\forall s : Traces(Pool_G) \bullet Pool_D(s)$$

(where $Pool_D$ is the predicate specification on Page 93, and $Pool_G$ is the program specification on Page 239).

If we can show that this is true, then we can (subject to the caveats about processes that are not the most general possible process) replace $Pool_D$ with $Pool_G$ for the rest of our work.

Approach

We can adopt a two-stage approach: first, we break the predicate definition $Pool_D$ into its conjunctive parts. Then we define a program, the 'conjunctive program', for each conjunctive part, and show that this new program refines the conjunctive part.

Then we use the facts[1] that

$$(\forall i \bullet P_i \textbf{ sat } S_i) \Rightarrow (\|_i P_i) \textbf{ sat } \bigwedge_i S_i$$

and

$$(I_2 \sqsupseteq I_1 \wedge I_1 \textbf{ sat } S) \Rightarrow I_2 \textbf{ sat } S$$

to justify using FDR to check that the program $Pool_G$ satisfies the parallel combination of conjunctive programs, and infer that $Pool_G$ refines $Pool_D$. This reduces the amount of mechanical work that we need to do: instead of attempting a single monolithic proof (I_2 **sat** S), we use FDR to show that $I_2 \sqsupseteq I_1$, and PVS to show that I_1 **sat** S.

More generally, this is a very useful technique for breaking down large CSP problems into smaller ones. However, it requires a considerable amount of effort in designing processes that can be broken down in this fashion.

[1]These facts are not proved in this thesis; they are asserted in many CSP texts. Note that the first fact is only valid in the traces models.

Recall from the definition of Pool on Page 93 that

$$\begin{aligned} \mathit{Pool}(D, d_0) \quad = \quad & \mathit{Basic}(D) \\ & \wedge\ \mathit{MayRead}(D) \\ & \wedge\ \mathit{MayWrite}(D) \\ & \wedge\ \mathit{PoolValue}(D, d_0) \end{aligned}$$

Because this is the untimed *traces* model, we can disregard the two conjunctive terms $\mathit{MayRead}(D)$ and $\mathit{MayWrite}(D)$. (In any of the other semantic models used in this thesis, we would have to include both of these terms in the following argument.) Moreover, Basic splits into three terms:

$$\begin{aligned} \mathit{Basic}(D) \quad = \quad & \mathit{ReaderSequence}(D) \\ & \wedge\ \mathit{WriterSequence}(D) \\ & \wedge\ \mathit{ConsistentWrite}(D) \end{aligned}$$

We choose to split Pool_D into three specification-program pairs:

- $\mathit{ReaderSequence}$;
- $\mathit{WriterSequence} \wedge \mathit{ConsistentWrite}$; and
- $\mathit{PoolValue}$.

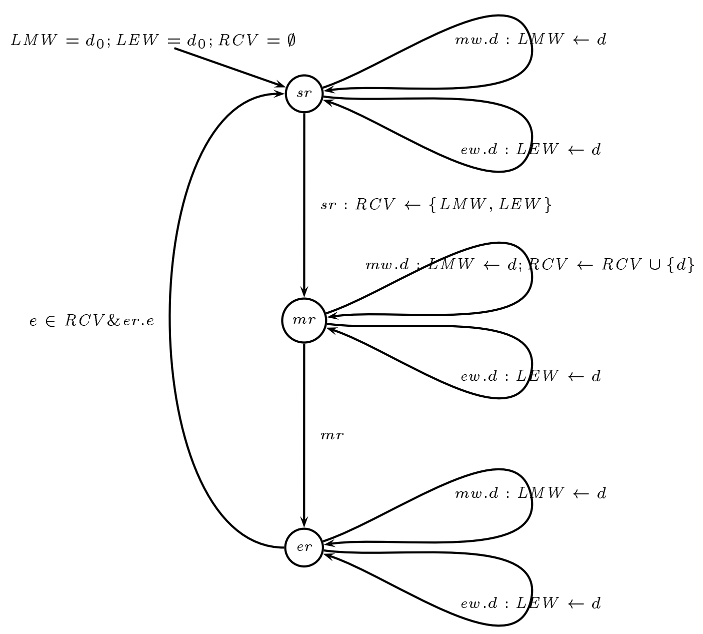
The first two programs are easy to define:

$$\mu X \bullet sr \rightarrow mr \rightarrow er?d \rightarrow X$$

for $\mathit{ReaderSequence}$, and

$$\mu X \bullet sw?d \rightarrow mw.d \rightarrow ew.d \rightarrow X$$

for $\mathit{WriterSequence} \wedge \mathit{ConsistentWrite}$. The program for $\mathit{PoolValue}$ (Page 93) can be described in state machine form (the name on each node indicates the next reader event that is expected):


$LMW = d_0; LEW = d_0; RCV = \emptyset$
$mw.d : LMW \leftarrow d$
$ew.d : LEW \leftarrow d$
$sr : RCV \leftarrow \{LMW, LEW\}$
$mw.d : LMW \leftarrow d; RCV \leftarrow RCV \cup \{d\}$
$e \in RCV \& er.e$
$ew.d : LEW \leftarrow d$
mr
$mw.d : LMW \leftarrow d$
$ew.d : LEW \leftarrow d$
sr
mr
er

The three variables are:

LMW the value of the last middle write;

LEW the value of the last end write; and

RCV the candidate values to be returned for a read.

Note that we do not say anything about the start-write event (this is important when determining the event sets for parallel composition[2]).

This program is arguably clearer than the predicate definition to describe the value that should be returned by a pool, except that the value returned by er is nondeterministic in the specification. That is, an implementation must be prepared to return at least one value, but does not have to be prepared to return all of them. (The same remark applies to the program on Page 239.)

[2]Parallel composition can be viewed analogously to conjunction: it constrains the possible observations of a process. Since we do not care about the start-write event, nor the order of the writer events for this particular program, we avoid even mentioning it in the parallel composition interface.

Proof for $\mathit{ReaderSequence}$

This proof is a trivial instance of the following general proof rule[3]:

$$\begin{array}{ll} & \mu X \bullet a : A \to b : B \to c : C \to X \\ \textbf{sat} & \\ & \quad tr\#A = tr\#B = tr\#C \\ & \vee\ tr\#A - tr\#B = 1 \wedge tr\#B = tr\#C \\ & \vee\ tr\#A = tr\#B \wedge tr\#B - tr\#C = 1 \end{array}$$

where

$$A \cap B = \emptyset \wedge A \cap C = \emptyset \wedge B \cap C = \emptyset$$

Here, we see one of the complications of moving from a two-point to a three-point model: the corresponding proof rule for two-point models is much clearer and shorter. We have traded a more descriptive model for more complicated proof obligations.

Carrying out this 'trivial' instantiation in PVS proves to be a little harder; we have to arrange that the types of the operators match correctly. For example, the overloading we use of the 'counting' operator (#) for both single events, and sets of events complicates the PVS implementation. The only remarkable point is just how much time it takes to prove this very simple statement.

Proof for $\mathit{WriterSequence}$ **and** $\mathit{ConsistentWrite}$

This requires two parts, which we combine using the rule that

$$\begin{array}{rl} & P\ \textbf{sat}\ S_1 \wedge P\ \textbf{sat}\ S_2 \\ \Rightarrow & P\ \textbf{sat}\ (S_1 \wedge S_2) \end{array}$$

for S_1 and S_2 predicate specifications, and P a program.

The first predicate of these two predicates is $\mathit{WriterSequence}$: the proof of this is identical to the proof for $\mathit{ReaderSequence}$ above.

The second predicate concerns the $\mathit{ConsistentWrite}$ predicate. First, we can replace the term $tr \downarrow \alpha W(D)$ in $\mathit{ConsistentWrite}$ on Page 90 with tr because the alphabet of the program we are using here is exactly the alphabet

[3]This rule is very nearly a tautology: however, it is a necessary step in a formal proof.

αW. (This can be done formally — we elect not to in this exposition for brevity.) This leaves us with showing that

$$\begin{array}{l} \mu X \bullet sw?d \rightarrow mw.d \rightarrow ew.d \rightarrow X \\ \textbf{sat} \\ \quad tr = s^\frown\langle sw.d_1, mw.d_2\rangle \Rightarrow d_1 = d_2 \\ \quad \wedge\ tr = s^\frown\langle mw.d_2, ew.d_3\rangle \Rightarrow d_2 = d_3 \end{array}$$

Note that the specification itself splits into two conjunctive terms. Although this is 'obvious' (a dangerous word in a proof), we need two further lemmas, to prove this formally:

$$\forall tr : Q \ \bullet\ last(tr) = mw.d \Rightarrow tr = s^\frown\langle sw.d, mw.d\rangle$$

$$\forall tr : Q \ \bullet\ last(tr) = ew.d \Rightarrow tr = s^\frown\langle mw.d, ew.d\rangle$$

where

$$Q = \mu X \bullet sw?d \rightarrow mw.d \rightarrow ew.d \rightarrow X$$

PVS aside, we can show that both lemmas are true by inspection of $Traces(Q)$ (the set of all traces in Q):

$$\begin{array}{l} \{\ \langle sw.d_1\rangle, \\ \quad \langle sw.d_1, mw.d_1\rangle, \\ \quad \langle sw.d_1, mw.d_1, ew.d_1\rangle, \\ \quad \langle sw.d_1, mw.d_1, ew.d_1, sw.d_2\rangle, \\ \quad \langle sw.d_1, mw.d_1, ew.d_1, sw.d_2, mw.d_2\rangle, \\ \quad \langle sw.d_1, mw.d_1, ew.d_1, sw.d_2, mw.d_2, ew.d_2\rangle, \\ \quad \langle sw.d_1, mw.d_1, ew.d_1, sw.d_2, mw.d_2, ew.d_2, sw.d_3\rangle, \ldots \mid d_1, d_2, d_3, \ldots \in D\} \end{array}$$

However, we cannot easily replicate this style of proof in PVS.

Instead, we use the substitution of **sat** in induction over traces (described on Page 171):

$$\begin{array}{l} \quad (\langle\rangle \in Traces(Q) \Rightarrow S(\langle\rangle)) \\ \quad \wedge \\ \quad \forall s, a \bullet \left(\begin{array}{ll} & (s \in Traces(Q) \Rightarrow S(s)) \\ \Rightarrow & ((s^\frown\langle a\rangle) \in Traces(Q) \Rightarrow S(s^\frown\langle a\rangle)) \end{array} \right) \\ \Rightarrow \\ \quad \forall s \bullet (s \in Traces(Q) \Rightarrow S(s)) \end{array}$$

and set

$$S(s) = (last(s) = mw.d \Rightarrow s = s'^\frown\langle sw.d, mw.d\rangle)$$

Now, we just need to prove two statements:

1.
$$\langle\rangle \in Traces(Q) \Rightarrow S(\langle\rangle)$$

2.
$$\forall s, a \bullet \left(\begin{array}{ll} & (s \in Traces(Q) \Rightarrow S(s)) \\ \Rightarrow & ((s^\frown\langle a\rangle) \in Traces(Q) \Rightarrow S(s^\frown\langle a\rangle)) \end{array} \right)$$

This can be proved by working through the individual cases that occur (*i.e.* classify the different sequences and events), but is very tedious to implement formally.

Again, the type system in PVS causes us difficulty in implementing this proof, but a mechanized proof is possible (although very messy). The second lemma follows the style of proof for the first.

Proof for $PoolValue$

This is the first proof we encounter that is not based on simple loops (although the previous proofs implicitly used general choice for carrying data values). However, it is very similar to the proof above in terms of overall strategy, because we use our trace induction scheme.

We now explain the proof more informally. For the event $er.d$ to have occurred, the event sr must have occurred previously in the trace. At this point, the set RCV is set to the values of the last mw and ew. This corresponds to $PoolValuesSR$. Any mw events occurring between the sr and mr are added into the set RCV; this corresponds to $PoolValuesSMR$. Thus the two sets RCV and $PoolValues$ are identical.

The difficulty is in proving this at a level that we can formalize in PVS. One of the reasons that this is harder than the previous proof is the existence of these three variables in the program definition: RCV, LMW, and LEW. This requires a particularly unpleasant PVS definition, as it incorporates both mutual recursion (as a least fixed point operator) and parameterization (a function from the parameters onto processes). The least fixed point operator has parameterized processes as both domain and range.

Further, we have to aid PVS by giving it further lemmas about the specific program. These say, 'this particular form of trace is admitted by this program', or 'this particular form of trace is not admitted by this program'.

This can be mechanized, but requires multiple lemmas (of the form described in the previous paragraph). This substantially limits the current usability of this overall method (*i.e.* for proof of even slightly complicated expressions). From this we conclude that formal proofs cannot generally be manipulated interactively; we need more automation to make this method usable.

7.1.2 Timed Failures Model

We aim to provide a timed semantics for DORIS in this thesis, so we must examine the timed properties as well as the untimed properties described in the previous section.

Fortunately, we do not need to repeat the conceptual work of the proofs above. They can be repeated in the timed model, although the extra (timing) information inherent in this (timed) model causes implementation difficulties with PVS: the repeated proofs have to manage the timed trace-timed refusal pairs.

What remains is to prove the $MayRead$ and $MayWrite$ terms. Earlier, we simply stated that

> "Timing is not explicitly represented on the state machines. The predicate specifications indicate that a maximal delay $\hat{t}$ is permitted on each transition ($\hat{t}_0$ for the first start-read and start-write)."

The proofs for $MayRead$ and $MayWrite$ in the case of the pool do not add anything interesting to the discussion: the program specification for the timed case merely has to nondeterministically delay for a time between 0 and $\hat{t}$. Our approach simply adds another two conjunctive specification-program pairs: one for each of $MayRead$ and $MayWrite$.

The more interesting cases relate to the channel and signal, with the conditional refusal predicates. However, they turn out to be managed in a similar way to the proof for $PoolValue$ (*i.e.* we define two conjunctive specification-program pairs, and add the programs into the FDR model).

Although the proofs are generally tedious and difficult to implement, they illustrate one useful fact (introduced on Page 160): we can refine the conjunction of predicates with the parallel composition of programs, and exploit the many conjunctions in the specification of the IDAs. Note that this approach is only valid in the traces models; the corresponding lemma when refusals are involved is more complicated.

7.1.3 Time and Effort

The proofs described in this section have required more work than the PVS and FDR model constructions together. An estimate of the time taken is around one year of work: moreover, this only covers several of the more interesting proofs, and even then, we have often omitted the fine details (*e.g.* proving that the trace denoted 'null' is the equal to the trace '(: :)').

Part of this time can be considered 'learning' time. Automated proof techniques are difficult. For the amount of time and effort invested, it is hard to derive any useful results. We will further discuss the benefits and difficulties later in the thesis.

7.2 FDR and Time

At this point, it becomes very clear that both rigorous and PVS proofs are interesting, but extremely time-consuming. The proofs themselves are not intellectually hard: indeed, some are very easy. It is mechanizing the proofs that is time-consuming.

As an alternative, we can use FDR. This raises an important issue: what happens to the deeper issues relating to time when we move from the real-time model of PVS, to the discrete time model using $tock$ in FDR?

We require that the granularity of $tock$ is sufficient to represent the timed traces and refusal sets. This means that we have to transform times until they fit the positive integers.

This is (for most traces) possible, but refusal sets are dense in time. For this, the granularity of $tock$ has to be sufficient to represent the refusal tokens in a refusal set (*i.e.* the step function from time to untimed refusal sets).

The way that the processes have been constructed means that we can be sure that time passes in a sensible way. Although the timed model in FDR is not the timed model used in PVS, it is a good enough approximation, provided that no processes violate the conditions described above, *e.g.* a Zeno process.

We are now in a position to use PVS when a particularly interesting or critical assertion is to be tested, and to use FDR for exploring more substantial problems.

7.3 Four-slot Implementation of the Pool

In Section 4.10, we presented a model of the four-slot implementation of the pool IDA, which itself was defined in Section 4.5. The obligation upon us is to prove that this implementation does indeed satisfy the specification.

Earlier in this chapter, in Section 7.1, we showed that the program specification of the pool refined the predicate specification. We can therefore use the program as our specification in FDR.

Note that if any counter-examples are found, we would have to check these against the predicate specification, or show that the predicate specification refined the program specification before we could assume that the implementation was faulty.

However, we did not need to refer to the predicate specification, because:

> *Under the untimed traces and untimed failures models, the four-slot implementation refined the pool specification.*

As we noted on Page 153, FDR does not give supporting evidence of such statements.

We can expand on this further by varying the type of Lamport variables involved. When Safe variables are used, the four-slot mechanism fails, which is not very surprising. The more reasonable regular variables satisfy the model, as do the atomic variables. Further, the mechanism still works when the Bit is implemented as a $\mathit{Regular}$ variable, and DS is implemented as a Safe variable.

Timed models (in so far as they can be analysed — it is extremely difficult to obtain results due to the large state spaces) will obviously vary according to the timing parameters for each component. The models only confirm this, but no details can be derived from FDR output.

7.4 Composition of IDAs

Having developed the programs for the four main types of IDA, it was unknown what behaviour the composed IDAs would possess using (in this case) the read in-write out composition process described in Section 4.7 (Page 99). This section describes the results of testing the following assertion for every combination of A, B, and C:

$$A \circ B \sqsubseteq C$$

where each is an IDA. Here, '$\circ$' denotes composition, hiding, then renaming the composed process to match the interface of C. Figure 7.1 gives the results for the untimed IDAs, and notes the semantic models that the result was found for.

Why is this useful to know? Although it would not result in faster implementations of the IDAs, it is possible to imagine circumstances when two composed IDAs could be replaced by a single IDA (assuming that an appropriate composition operator was used).

When considering the IDAs more abstractly, we conjecture that they might possibly be a group under composition-and-refinement. (This is partially for mathematical curiosity. If we knew that IDA A could be decomposed into IDAs B and C, we would not have to implement A in the proof system, although this would obviously not be sensible in a real implementation of a design.) Figure 7.1 is derived from results from FDR, and shows that this is not the case. Although every pair of IDAs was refined by an IDA, in most cases, there were many IDAs matching the refinement.

More interesting is that most were only refinements under the traces model. The failures resulted from refusals to engage in an event quickly enough.

For a more general application, this result can be re-run in FDR with different composition operators to produce lemmas for PVS (which we then claim are proved by FDR). By choosing an appropriate composition operator (for the particular problem), parts of a network can be compressed.

IDA Type			
A	B	C	Semantic Model
Channel	Channel	Channel	Traces only
Channel	Constant	Constant	Traces only
Channel	Pool	Channel	Traces only
Channel	Pool	Constant	Traces only
Channel	Pool	Pool	Traces and failures
Channel	Pool	Signal	Traces only
Channel	Signal	Channel	Traces only
Channel	Signal	Signal	Traces and failures
Constant	Channel	Constant	Traces and failures
Constant	Constant	Constant	Traces and failures
Constant	Pool	Constant	Traces only
Constant	Pool	Constant	Traces only
Constant	Signal	Constant	Traces only
Pool	Channel	Channel	Traces only
Pool	Channel	Constant	Traces only
Pool	Channel	Pool	Traces and failures
Pool	Channel	Signal	Traces only
Pool	Constant	Constant	Traces only
Pool	Pool	Channel	Traces only
Pool	Pool	Constant	Traces only
Pool	Pool	Pool	Traces only
Pool	Pool	Signal	Traces only
Pool	Signal	Channel	Traces only
Pool	Signal	Constant	Traces only
Pool	Signal	Pool	Traces only
Pool	Signal	Signal	Traces only
Signal	Channel	Channel	Traces only
Signal	Channel	Signal	Traces and failures
Signal	Constant	Constant	Traces only
Signal	Pool	Channel	Traces only
Signal	Pool	Constant	Traces only
Signal	Pool	Pool	Traces and failures
Signal	Pool	Signal	Traces only
Signal	Signal	Channel	Traces only
Signal	Signal	Signal	Traces and failures

Figure 7.1: Results of composition of untimed IDAs

7.5 PVS Lemmas

Even without the case studies in the next chapter, there is a very wide range of proofs that we can choose as a test subject for the theory. Indeed, the work that this thesis is based on examines only a few of these lemmas.

The possible lemmas can be categorized thus:

- TCCs generated for parts of the Timed CSP theory;
- TCCs generated for parts of the DORIS-on-Timed CSP theory;
- 'Trivial' lemmas, *e.g.* to convert from one style of notation to another within the theorem prover;
- Other 'obvious' lemmas (*e.g.* 'a trace is either empty, or consists of a trace with a singleton trace appended');
- General proof strategies (*e.g.* the trace_induction lemma described below); and
- Lemmas about IDAs.

To make the overall problem more manageable, a number of the more interesting lemmas have been chosen. The following lemma is generally more useful than the IDA lemmas, and is used in several of the previous proofs.

Induction Over Traces

The mathematical statement of this lemma is

$$\begin{array}{lll} & & P(\langle\rangle) \\ & \wedge & \\ & & \forall s, a \bullet P(s) \Rightarrow P(s ^\frown \langle a \rangle) \\ \Rightarrow & & \\ & \forall s \bullet P(s) & \end{array}$$

where P is a predicate over traces; s is a trace; and a is an event. If we wish to prove that Q **sat** S, then the expansion of **sat** gives us

$$\begin{array}{ll} & \forall tr : \mathit{Traces}(Q) \bullet S(tr) \\ \equiv & \forall s \bullet s \in \mathit{Traces}(Q) \Rightarrow S(s) \end{array}$$

Setting

$$P(s) = (s \in Q \Rightarrow S(s))$$

allows us to use our induction lemma:

$$\begin{array}{l} \quad (\langle\rangle \in \mathit{Traces}(Q) \Rightarrow S(\langle\rangle)) \\ \wedge \\ \quad \forall s, a \bullet \left(\begin{array}{ll} & (s \in \mathit{Traces}(Q) \Rightarrow S(s)) \\ \Rightarrow & ((s ^\frown \langle a \rangle) \in \mathit{Traces}(Q) \Rightarrow S(s ^\frown \langle a \rangle)) \end{array} \right) \\ \Rightarrow \\ \quad \forall s \bullet (s \in \mathit{Traces}(Q) \Rightarrow S(s)) \end{array}$$

For this to be part of a valid proof, we must prove that the induction lemma is true. We can perform this proof in PVS, as illustrated by the proof tree in Figure 7.2. There are three noteworthy steps in this proof:

- In this proof, the 'measure induction' scheme (measure-induct in PVS) replaces a consequent of the form

  $$\forall s \bullet P(s)$$

  with

  $$\forall s_1 \bullet (\forall s_2 \bullet \mathit{length}(s_2) < \mathit{length}(s_1) \Rightarrow (P(s_2) \Rightarrow P(s_1))$$

- The lemma cons_foot states that any trace s is empty, or can be written $s = s' ^\frown \langle a \rangle$ where s' is a trace, and a is an event. This introduces a disjunctive antecedent, which can be split into two proof branches.

- Lemma length_foot is related to the previous lemma: it says that

  $$\mathit{length}(s' ^\frown \langle a \rangle) = \mathit{length}(s') + 1$$

The rest of the proof steps are concerned with substituting variables, instantiation, and simplifying expressions.

Of course, PVS then imposes an obligation to prove that the two lemmas cons_foot and length_foot are true. Curiously, these much simpler statements are quite difficult to prove within PVS.

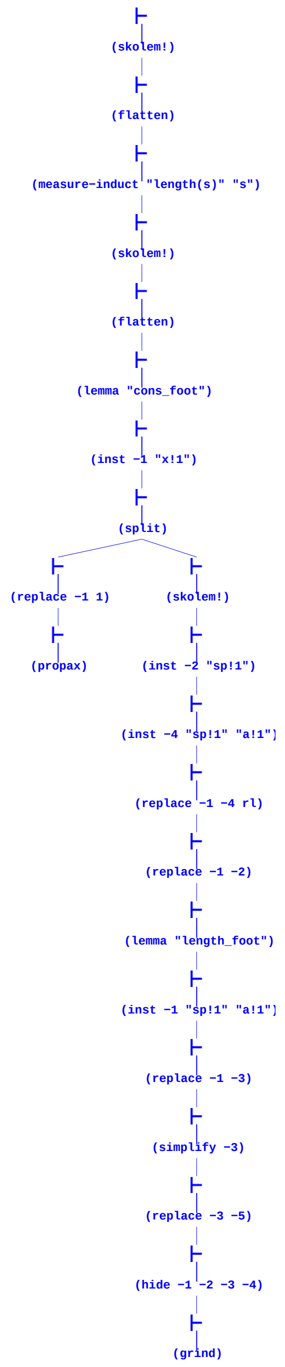


Figure 7.2: Proof tree for induction over traces

7.6 Comments on Results

The most striking thing about these results is the sheer difficulty encountered in attempts to formalize these proofs. These are small objects: they are relatively easy to understand, and to provide with rigorous (or at least sketch) proofs. Indeed, the programs for the different IDAs are relatively easy to informally validate with respect to the predicate specifications. Implementing parts of these proofs as formal PVS proofs has taken the year referred to in Section 7.1.3.

There are two broad (overlapping) classes of problem: those that are unbounded, and those concerning programs that have relatively simple state machine representations. The unbounded problems are not handled well by FDR: it is difficult to construct a 'phrasing' of the program that does not exhibit a state-space explosion. These problems are more easily solved by PVS.

Conversely, it is difficult to manipulate programs within the PVS model. Given a program, it is difficult construct an argument from the structure of the program to a predicate expression. This could be a problem of the construction of the CSP embedding used in this thesis. The embedding is intended to be a very general model: previously published work has used highly constrained models (*e.g.* Dutertre *et al.*'s work on security protocols [26, 27]).

Alternatively, we could argue that there is in fact a great deal of information in our abstract model. This would imply that we have to handle this information regardless of the formalism. Thus we should expect to find it difficult to simplify the problems.

A further alternative is that the rich language of CSP requires a very rich library of lemmas about constructs, and that only a small number have been constructed as needed. It would be helpful if sufficient libraries could be constructed, but it is not clear what general statements are needed. Some useful statements concern the usual commutativity, associativity, and identity laws for the primitive CSP processes. As indicated in Section 7.4, we can derive certain types of results about the IDAs from PVS. But the overall proofs strategies require a higher-level of structure which is difficult to apply to more than a single proof.

It is more likely that both the general nature of the construction, and the inherent complexity of the formal proofs are the source of our difficulties.

Chapter 8

Case Studies

8.1 Introduction to Case Studies

The overall aim of this work is to produce an industrially usable semantics for the DORIS notation. Although the meaning of 'usable' can often be disputed (it depends on the overall context), case studies can provide useful information on the method or notation in question. In this instance, we intend to produce a semantics that as well as being mathematically elegant, is suitable for machine checking or proof.

Two case studies have been chosen: one small and one large. The small case study is based on the example in Paynter *et al.*'s paper [83], and is presented in Section 8.2. The second is a system of significant size, based on a British Aerospace product (Section 8.5, Page 185).

8.2 Small Case Study

This first case study is intended to show whether the syntax and semantics defined in Chapters 3–5 fit together sensibly, and are amenable to machine work on a small scale.

Recall Figure 8.1 (this is the same as Figure 5.1 on Page 113). This example is Figure 2 (Page 13) of Paynter's paper [83].

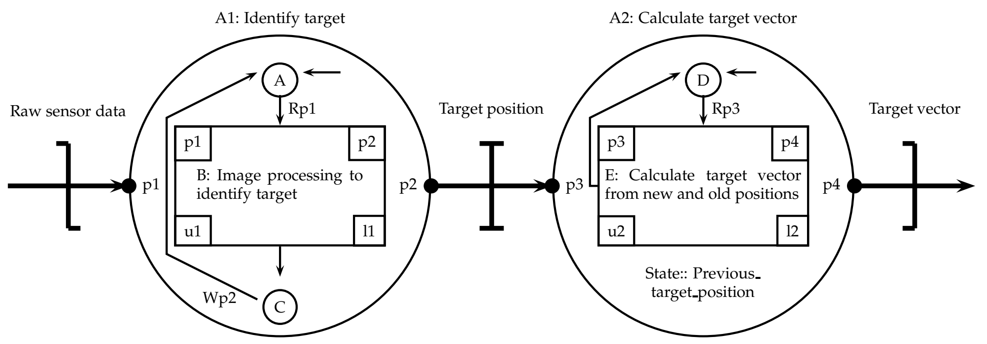

A1: Identify target
A2: Calculate target vector
Raw sensor data
Target position
Target vector
A
Rp1
p1
p2
B: Image processing to identify target
u1
l1
Wp2
C
D
Rp3
p3
p4
E: Calculate target vector from new and old positions
u2
l2
State:: Previous_target_position

Figure 8.1: A target tracker in ADL

8.2.1 Translation of the Design

Figure 8.2 gives the design in the DORIS notation. This design contains two activities and three IDAs. These are connected with six simple paths, and the whole object is viewed as a complex server. (It is not a system alone; and it cannot be a complex activity, since it possesses windows.) As in Paynter's paper, this translation is performed on an *ad hoc* basis.

We are now in a position to give a statement in terms of the syntax defined in Chapter 3 (Page 58). This is presented in Figure 8.3. Note that for IDAs, we are setting the window for data input (*i.e.* write events) to be `w1`, and the window for data output (read events) to be `w2`. Two code stubs are required: these are given in Figures 8.4 and 8.5.

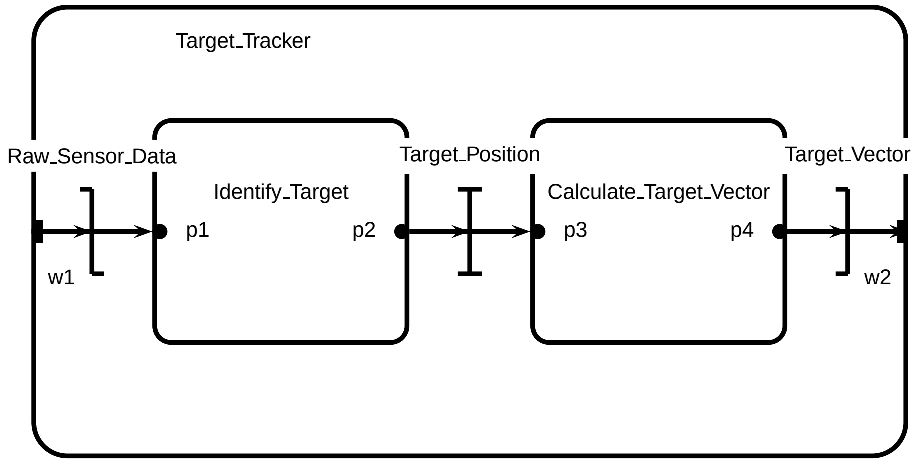


Figure 8.2: DORIS version of the target tracker

```
(cs,Target_Tracker,
    (sa, Identify_Target),
    (sa, Calculate_Target_Vector),
    (rt, Raw_Sensor_Data, Signal),
    (rt, Target_Position, Channel),
    (rt, Target_Vector, Pool),
    (sp, , w1, Raw_Sensor_Data:w1),
    (sp, , Raw_Sensor_Data:w2, Identify_Target:p1),
    (sp, , Identify_Target:p2, Target_Position:w1),
    (sp, , Target_Position:w2, Calculate_Target_Vector:p3),
    (sp, , Calculate_Target_Vector:p4, Target_Vector:w1),
    (sp, , Target_Vector:w2, w2))
```

Figure 8.3: Textual statement of small case study

```
Identify_Target  =  Read(p1, x);
                    Work(l1, u1);
                    Write(p2, x);
                    Self
```

Figure 8.4: Code stub for Identify_Target

```
Calculate_Target_Vector  =  Read(p3, y);
                            Work(l2, u2);
                            Write(p4, y);
                            Self
```

Figure 8.5: Code stub for Calculate_Target_Vector

8.2.2 Required Properties

Paynter *et al.* identify two properties that this system should exhibit:

Timeliness-deadline (also known as 'liveness') is the requirement that input to Raw_Sensor_Data should generate output to the pool Target_Vector within some deadline; and

Safety is the requirement that no spurious outputs are generated.

Note that in the first condition, we have identified the left-hand side of the pool (Target_Vector) as the terminal point for timing, since the pool will continue to serve old data on its output (to Target_Tracker.w2) if it doesn't receive newer data.

We need to identify the CSP statements for these two requirements. The first is

$$\begin{aligned} & \tau \downarrow \{\text{Target\_Vector}.sw\}_i = (t_2, \text{Target\_Vector}.sw) \\ \Rightarrow \quad & \tau \downarrow \{\text{Target\_Tracker.w1}.sw\}_i = (t_1, \text{Target\_Tracker.w1}.sw) \\ & \wedge\, t_2 \leq t_1 + D \end{aligned}$$

(where s_i means the ith element of the sequence s). This can be interpreted as

> "If the ith write to Raw_Sensor_Data starts at time t, then the ith write to Target_Vector must start within D time units."

This statement asks us to check that a deadline is satisfied across multiple components.

Similarly, the second statement is

$$tr\#\{\text{Target\_Vector}.sw\} \leq tr\#\{\text{Target\_Tracker.w1}.sw\}$$

and can be interpreted as

> "The number of start writes to Target_Vector must be no greater than the number of start writes to Raw_Sensor_Data."

There are two points to note here: firstly, we have assumed the natural mapping of DORIS names to CSP events; secondly, that the routes are 'reasonably fast', *i.e.* the IDAs will not impose additional delays, except for when there is no space in the IDA. (We will expand on this later.)

A single file captures the information above for input to dt. This file is given in Figure 8.6. Note that we have to give the work time bounds — the numbers here are completely arbitrary.

```
cs:Target_Tracker:0
sa:Identify_Target:H
sa:Calculate_Target_Vector:H
rt:Raw_Sensor_Data:Signal
rt:Target_Position:Channel
rt:Target_Vector:Pool
sp::w1::Raw_Sensor_Data:w1
sp::Raw_Sensor_Data:w2:Identify_Target:p1
sp::Identify_Target:p2:Target_Position:w1
sp::Target_Position:w2:Calculate_Target_Vector:p3
sp::Calculate_Target_Vector:p4:Target_Vector:w1
sp::Target_Vector:w2:w2:
::


defIdentify_Target
Read(p1, default); Work(5, 10);
Write(p2, default); Self
enddef


defCalculate_Target_Vector
Read(p3, default); Work(4, 11);
Write(p4, default); Self
enddef
```

Figure 8.6: dt input file for the small case study

8.3 Small Case Study: Safety Condition Analysis

8.3.1 Sketch Proof

This proof is very simple, and relies on the mathematical fact that

$$\begin{aligned} & a \geq b \wedge b \geq c \\ \Rightarrow \quad & a \geq c \end{aligned}$$

(*i.e.* $\geq$ is transitive). We choose to count the start-write and end-read events of each IDA in Figure 8.2 (except for the end-reads of Target_Vector, because it is a pool).

We define several variables:

$$\begin{aligned} c_1 &= tr\#\mathsf{Raw\_Sensor\_Data}.SW \\ c_2 &= tr\#\mathsf{Raw\_Sensor\_Data}.ER \\ c_3 &= tr\#\mathsf{Target\_Position}.SW \\ c_4 &= tr\#\mathsf{Target\_Position}.ER \\ c_5 &= tr\#\mathsf{Target\_Vector}.SW \end{aligned}$$

We can see that the following property holds:

> For any signal or channel IDA, the number of start-writes is greater than or equal to the number of end-reads.

A sketch proof for this statement follows from the construction of the safety predicate on $SignalValue$ and $ChannelValue$ respectively (Pages 95 and 97). This safety predicate, in conjunction with $ReaderSequence$ means that the reader cannot read unless there has been a write. (In the case of a signal, some values are also discarded unread.)

Therefore

$$\begin{aligned} & c_1 \geq c_2 \\ \wedge \quad & c_3 \geq c_4 \end{aligned}$$

By examination of the two programs Identify_Target and Calculate_Target_Vector, we can see that

$$\begin{aligned} & c_2 \geq c_3 \\ \wedge \quad & c_4 \geq c_5 \end{aligned}$$

Thus $c_1 \geq c_5$, which is the statement we intended to prove. Note that at no stage in this proof do we care about the value that is transmitted — we simply want to ensure that no spurious outputs appear.

8.3.2 PVS Proof

The PVS proof follows the structure of the sketch very closely. The only difficulties are encountered in the two lemmas, the first concerning the signal and channel, and the second relating to the two programs' inputs and outputs.

8.3.3 FDR Check

There is some difficulty in using FDR for this problem: the specification program is inherently unbounded. Instead, we introduce a specification that has an upper bound, and generates an alarm event when that bound is reached. (This illustrates a general method of addressing these types of FDR checks. Note that this can generate 'false witnesses' where the alarm bound is reached, but the unbounded program would refine the unbounded specification.)

In this case, the program for the small case study successfully refines our specification program in FDR.

8.4 Small Case Study: Liveness Condition Analysis

8.4.1 Sketch Proof

For the purpose of this section, we simplify our model slightly, and choose to allow IDAs to work 'very fast'. This does not make the problem easier, but smaller to illustrate.

Our strategy chains together significant events, in a similar fashion to the safety property. Instead of counting the events, we calculate the maximum time between the pairs. The total of these times gives the maximum time.

Thus, if Identify_Target and Calculate_Target_Vector use up to d_1 and d_2 time units respectively, then the total is $d_1 + d_2$. By inspection of the programs, we can see that the worst case is $d_1 =$ u1 and $d_2 =$ u2. (The simplification given in the first paragraph effectively sets the maximum time spent in each IDA to be zero: generally, we would end up with three further terms to add to the total.)

In this type of proof, we must carefully check that nothing can block: if it could, the maximum time is unbounded. There is indeed a problem: a process could write to w1 (Raw_Sensor_Data) twice very quickly. If Identify_Target has not finished processing a previous input (*i.e.* less than l1 time units have passed) then the first input will be overwritten by the second, thus falsifying our liveness condition. An example trace is in Figure 8.7. The response for v_2 will never be generated, as it has been overwritten in the signal IDA.

Therefore, we have to impose a constraint on a user of this system (the process that writes to Raw_Sensor_Data). We suppose that there is a minimum delay, g, between two consecutive writes to Raw_Sensor_Data.

We can see that g must be strictly greater than u1 to ensure that a read can be completed before the value is overwritten. Alternatively, the signal could take sufficiently long to process a write, thus refusing the write events 'too soon'.

This means that the system must keep up with the environment, or have some means of refusing to cooperate when the previous input hasn't been handled.

Although this is a long (sketch) proof to show this point, it is a relatively verbose proof. However, it demonstrates that the IDAs possess properties that can have serious implications in a system: the overwriting in the signal.

8.4.2 PVS Proof

Again, the only difficulty in this proof is extracting a desired property from a CSP program. In this instance, we need to go from the program which reads, works, then writes, to a statement of particular executions: merely enumerating all possible observations is not useful, since we cannot subsequently use them.

The minimum delay condition is easily built into the lemma that we

$\langle$ $(0, \textsf{Raw\_Sensor\_Data}.sw.v_1)$,
$(0, \textsf{Raw\_Sensor\_Data}.mw.v_1)$,
$(0, \textsf{Raw\_Sensor\_Data}.ew.v_1)$,
$(0, \textsf{Raw\_Sensor\_Data}.sr)$,
$(0, \textsf{Raw\_Sensor\_Data}.mr)$,
$(0, \textsf{Raw\_Sensor\_Data}.er.v_1)$,
$(0, \textsf{Identify\_Target}.sp.\textsf{l1}.\textsf{u1})$,
$(0, \textsf{Raw\_Sensor\_Data}.sw.v_2)$,
$(0, \textsf{Raw\_Sensor\_Data}.mw.v_2)$,
$(0, \textsf{Raw\_Sensor\_Data}.ew.v_2)$,
$(0, \textsf{Raw\_Sensor\_Data}.sw.v_3)$,
At this point, the input of v_2 has been overwritten:
it will never generate an output.
$(0, \textsf{Raw\_Sensor\_Data}.mw.v_3)$,
$(0, \textsf{Raw\_Sensor\_Data}.ew.v_3)$,
$(t_1, \textsf{Identify\_Target}.mp)$,
$(t_2, \textsf{Identify\_Target}.ep)$,
$(t_2, \textsf{Raw\_Sensor\_Data}.sr)$,
$(t_2, \textsf{Raw\_Sensor\_Data}.mr)$,
$(t_2, \textsf{Raw\_Sensor\_Data}.er.v_3)$ $\rangle$

where $\textsf{l1} \leq t_1 \leq t_2 \leq \textsf{u1}$

Figure 8.7: Counter-example for small case study (liveness condition)

use for this problem. For each event transition (since this is a chain of events without any choices), we determine the maximum delay. Subject to the condition described, this proof is successful.

8.4.3 FDR Check

We use a similar process to the previous FDR check: engage in an 'alarm' event if the process takes too long. Under any failures model, the refinement will fail if too much time passes, since alarm will be refused in the implementation.

Although the model and method for testing the model both appear to be sound, the construction of the model causes FDR to run rapidly out of resources (it appears to be unbounded). The assertion (under FDR) cannot be tested.

8.5 Large Case Study

The system that we are modelling is the Electronics and Power Unit (EPU) for a flight control system produced by British Aerospace. The design details are from several original documents:

- 45 pages of DORIS notation; and
- several booklets describing the general make-up of the system, and brief descriptions of the code stubs.

8.5.1 Summary of the Design

It consists of a number of concurrent activities distributed over a transputer network which includes six processors, which communicate on 8 direct multiplexed links. The breakdown by DORIS type is given in Figure 8.8. Section B.2 (Page 225) contains a series of extracts from the actual main data file; this file is over one thousand lines long — calculating the CSP for this case study by hand would have been both tedious, and difficult to perform correctly. Instead, dt was used to generate the CSP code.

Complex Components		Basic Components	
sy	1	tp	6
cs	8	td	8
ca	13	ti	7
dv	36	cs	8
		ca	13
		ss	18
		sa	39
		dv	36
		rt	144
		cp	88
		sp	656
Total	58	Total	1023

Figure 8.8: Breakdown of large case study by DORIS type

8.5.2 Code Stubs

The major problem with this model appears with the definition of the code stubs. The documentation available to the author had relatively little description of the code stubs.

In general, for every entity requiring a code stub, a simple loop was constructed, possibly with choices, which read and wrote to (most if not all) of the connected ports. Although this is not exactly the design intended, this does exercise the theory appropriately.

57 such code stub definitions were required.

8.5.3 Machine Assistance

The prototype tool, dt, described in Appendix B (Page 223) was used to construct the CSP for the PVS model. This overcame the problem of a hand-translation: sheer size. This means that (when appropriate) the forwarding processes for the IDAs can be automatically generated, and that all the ports can be assigned their correct windows.

8.5.4 Required Property

Having constructed the model, we now require a predicate to test the model with. We chose the deadlock freedom test, *i.e.* for all traces, there is an event that extends the trace.

$$\forall s \in traces(A) \bullet (\exists e \bullet s^\frown\langle e\rangle \in traces(A))$$

where $traces(A)$ is the set of all traces of the model.

8.6 Large Case Study Analysis

8.6.1 FDR

We chose to use the inbuilt FDR tests for deadlock and livelock. The *untimed* model was too large for either test; instead, each of the six main subsystems were tested: these were deadlock and livelock free. Note that we cannot infer that the timed system is deadlock or livelock free from these results.

(The model pushed the (big) machine FDR was running on to the limit of resources in terms of virtual memory, to such a degree that other processes using resources would sometimes cause a check to fail, even when FDR compression features were used.)

8.6.2 PVS

No proof for deadlock freedom in the large case study was attempted. There is a very simple reason for this: a rigorous proof is not obvious, even when the design is examined closely. Without brute-force mechanical assistance, it is difficult to understand the structure of the system in terms of deadlock. Without a candidate rigorous proof, a mechanical proof would have been impossible to perform.

There is the question of why we could not determine a proof. We would like to generate lemmas that are based on transactions, like those in the small case study: if a component takes an input at one time, we might except a particular output after some specified delay later.

However, the system consists of many paths which traverse the entire hierarchy. This results in a system that cannot be split into functionally separate parts to produce transaction lemmas about. The inability to reduce the problem to small lemmas is the difficulty here.

On Page 202, we describe a metric which followed from this observation.

8.7 Comments on Results

The first comment is that some results about both small objects (IDAs — in the previous chapter) and larger systems could be obtained. A mixture of mechanical methods (model checking and proof) were used, illustrating that this theory is applicable.

Some rigorous proofs with PVS about small properties (but not necessarily small systems) are relatively easy. The difficulty, as noted previously, is in completing the mechanized proof in sufficient detail. FDR allows systems that are difficult to reason about to be checked for relatively simple properties without user intervention.

However, regardless of the result, a PVS analysis usually increased the understanding of how a particular system worked. FDR simply says, 'success' or 'failure: here are some witnesses'. The latter results are useful for debugging, but the former does not add to the understanding of a process.

There are several areas in which improvement would aid the implementation of this work:

- A large library of properties about CSP would remove some of the difficulty with PVS proofs.
- An improved prover interface for PVS would allow time that was previously used 'fighting' the prover to be used instead for constructing proofs.
- Continuing improvements to FDR, in particular, to allow a cleaner model of time (using $tock$, as described by Roscoe [95]), and to allow larger systems to be analysed. In common with other model checkers, the 'state of the art' is continually improving.
- An integration of theorem provers and model checkers: theorem provers would benefit from automated examination (under the direction of heuristics); model checkers simply cannot cope with the

large systems — the incorporation of results from a theorem prover might allow larger compressions of systems than current methods allow. Thus the integration would help both types of mechanical support.

Part IV

Discussion and Conclusions

Chapter 9

Discussion

In this chapter, we address each specific part of the work in this thesis. The individual steps from the DORIS design through to tool support are considered. We summarise and evaluate these individual steps, and then discuss the implications for hierarchical timed notations generally and future work.

9.1 DORIS Semantics

Chapters 3, 4, and 5 present a denotational semantics for DORIS in Timed CSP.

A number of issues arose:

- Why use Timed CSP for the semantic domain?
- Why formulate the overall syntax and semantics in that form?
- How to model concurrent shared variables (the IDAs).
- How to model the active parts of the system.
- How to manage the interaction of time and functionality.

Each of these issues can be addressed briefly in the following sections.

9.1.1 Why Timed CSP?

Timed CSP is a well-understood process algebra. It has a wide range of operators that match the application domain, *e.g.* network parallel and hiding. The consistent set of semantic models allows us to choose a semantic model appropriate to the property we are testing.

There are alternative process algebras, *e.g.* CCS and Circal [72, 73]. We had to choose one: Timed CSP appears to be the most well-developed theory, especially in terms of timing, although any of the other well-known process algebras would almost certainly have been at least as adequate for our purpose.

9.1.2 Overall syntax and semantics

The semantics presented in Chapter 3 are compositional in the sense that any complex component can be calculated alone via the three-part semantic meaning function $\mathcal{C}$ (Page 69). These complex components can then be combined into other components, by resolving the currently unresolved paths ($\mathcal{C}_I$).

This means that we can construct individual components in both a top-down and bottom-up approach. Top-down system design is supported by defining 'black box' specifications of the lower components; bottom-up construction is supported by building layers of components and combining them as complex components. Re-use by (re)naming is supported in the syntax.

This gives us a natural view of the system design process: it reflects how systems are designed, rather than imposing a strict process.

Later in this chapter, we discuss the point that the denotational semantics effectively 'translates' the design into Timed CSP. Certainly, the denotational approach hinders some aspects of proof; an axiomatic method would have been more suitable. However, we would have lost the more intuitive approach offered by the denotational semantics.

The denotational semantics is effective in the sense that it is intuitive. It directly supports (through recursion of the semantic functions) the compositional approach. We noted on Page 46 that an adequate understanding of a language is attained when it possesses several different, but consistent, types of semantics. The denotational semantics presented here is clear and unambiguous, and offers a good starting point for complementary semantics to be designed.

9.1.3 IDAs

The study of concurrent shared variables is interesting, and has important applications to any situation where data corruption or timing interference are harmful. Lamport's variables (Section 4.2.1, Page 81) provide a more abstract view of this issue, and because of their more primitive nature can be used to 'implement' Simpson's IDAs (Section 4.2.2, Page 83).

The four main (families of) IDAs possess two classes of orthogonal properties: can they block the reader, and can they block the writer? This leads to modelling decisions about the policy to be adopted when data can be lost. In the process of making these decisions, we encountered difficulties when only two events were used to indicate an interaction with an IDA, and found that using three events provides a sensible approach.

The specifications of the IDAs then provide an abstraction away from possible implementations: we can reason about a system without regard to the implementation because we have a proof obligation that can be discharged independently, *i.e.* prove that a particular implementation of an IDA refines the specification.

Chapter 7 gave a program specification of the pool IDA; compared to the rather convoluted predicate specification (from Chapter 4), we might be inclined to say that all specifications should be given as programs. However, when contrasted with the constant (can the writer write, or is it refused?) we see that this isn't a simple decision: the predicate specifications can describe some subtle behaviours more easily.

In this sense, the definition of the IDAs is unsatisfactory: the predicate definitions require some effort to understand them; the program definitions are 'large' state machines (nine states each with variables). We can contrast these with Simpson's RTL definitions [111]: we then see that our definitions are not significantly more complex. From this, we conclude that the properties we are trying to model are inherently complex.

The attempts to prove that the four-slot mechanism was indeed a pool illustrated how independent (*i.e.* orthogonal) specifications could be used to break up a definition into manageable parts. For predicates, this reveals itself as conjunction; programs see this as parallel composition. The two can then be mixed: a program that satisfies a particular predicate can be placed in parallel with another program representing other properties.

From these points, we can see that combinations of predicate and program specifications can give a clearer view of a desired specification. By choosing orthogonal properties, the proof burdens can be eased. Future IDA designs should be constructed with this in mind.

It is interesting to note that we can make up plausible properties for IDAs quite easily: choose how many readers and writers are involved, and whether they can be blocked; determine the timing policies (we could have chosen very esoteric timing conditions); determine what value is read on the basis of previous reads; and then compose the chosen properties together. There is much that can be profitably investigated here.

Much of the motivation for designing the IDAs is to 'decouple' the activities connected to either side of a particular IDA. This then results in the distinction between which end of a path is driving the communication. This distinction is denoted by resolved paths having a port at one end (the driving end), and a window at the other end (the driven end).

Of course, ports and windows also have a concept of being inbound or outbound, *i.e.* the direction of the information flow.

The implication for the Timed CSP model is that the passive end can refuse to perform events, while the active end attempts to perform them as fast as possible.

General Access Interfaces

In Section 3.2.2 (Page 57), we mentioned the MASCOT-3 concept of 'general access interfaces'. These can be modelled in this semantics by using a device containing several routes.

To do this, the generalisation of a library of routes with well-known behaviours could be extended to include a library of devices with well-known behaviours.

In the process of developing the semantics and case study models, we can make the following (simple, but important) observation: Ports and windows in DORIS designs are given very sparse names, *e.g.* `p7`, `w3`, or `p1:a` (see the syntax in Appendix A.2, Page 216). In common with higher-level programming languages, replacing the numbers and complex channel letters with more descriptive names would result in a more readily understandable design.

9.1.4 Activities

The model of an activity in this thesis is intended to be a model of the imperative program for that activity. The model is an abstraction of the behaviour of the activity: it only has to capture the interactions with its environment (typically, the IDAs) by considering which ports are read and written, and how long calculations may take. The internal behaviour is not relevant to the overall system.

The functional properties of an activity, when they are of concern, can be modelled by mathematical expressions, *e.g.* if the value a is read, then $f(a)$ will be written out. How f is calculated is irrelevant here — how long it takes is relevant; how it does it, *i.e.* the internal details, is not.

Our model of activities captures exactly the details that we need: we do not need to worry about implementation details, yet we capture the local properties that affect the overall properties of the system.

Complex Servers and Activities

The DORIS semantics given in this thesis does not distinguish between complex servers and complex activities, except for the restriction concerning the appearance of windows. Devices, however, further restrict the appearance of active basic components.

A future semantics could eliminate the complex servers and complex activities, and replace them with a single 'complex active' component which may contain any basic component internally, and may possess both ports and windows. This would be helpful by removing an unnecessary complication in distinguishing between complex servers and activities.

Transputer Information

At a late stage during the development of dt, it became clear that all the transputer and transputer link information could sensibly be separated from the rest of the design information.

This should be stored, along with the details of the transputer allocation for various components, in a related, but alternative file. This would allow for a single design to be mapped to multiple architectures, which then enables easier comparison of the effect of different layout strategies on the performance of the system.

9.1.5 Timing

If we were not worried about the effects that the passage of time might have on the system, then we could simply use the untimed traces and untimed failures models of CSP. However, there are occasions when we need to explicitly model the passage of real-time.

The semantics capture the passage of time in a consistent fashion, where the semantic model of Timed CSP used can be matched to the properties considered.

For the types of properties considered, the model presented in these semantics is at an appropriate abstract level (*i.e.* it captures the interactions that we believe to be relevant, regardless of the semantic model), without too much, nor too little mathematical detail.

9.2 Mechanical Translation of DORIS to Timed CSP

The tool dt can be viewed as evidence that the syntax can be mechanically translated to the semantic domain. (The PVS and FDR models show that the semantics is mechanically implementable.) The experience of writing the tool provided much feedback for the semantics: trying to write the tool and discovering that the general case of some construct is difficult to implement (say, because some feature of the notation has been forgotten) provides useful suggestions to incorporate into revisions of the semantics. Indeed, both the semantics and dt have been through many revisions.

It is interesting to note which parts of dt required the most effort, and the most code: the resolution of paths has proven to be quite difficult to describe, both mathematically and as an imperative program. This is worth remarking since when a user is presented with forty pages of DORIS diagrams, the path resolution for a single interface is trivial: the user can simply trace where the lines go.

The most code of any distinct section of dt is involved with writing the output. This is not surprising, since (at this time) nine distinct output formats are supported, and these require that type-dependent code is generated for virtually every part of the context tree in a depth-first traversal (see Section B.3, Page 228).

9.3 Mechanical Support

The mathematical theory is now reasonably mature, and fits together in a regular, intuitive fashion. Chapter 6 started with the assertion that we require mechanical support for reasoning about systems modelled with these semantics.

The solution presented in Chapter 6 contained three components: an embedding of Timed CSP in PVS, a general model of DORIS systems in Timed CSP/PVS, and a general model of DORIS in FDR.

The main advantage of this approach is its modularity: if another mechanical implementation of Timed CSP is chosen, then the output routines of dt can be extended to generate the appropriate input files. If another notation is given a Timed CSP semantics, then the PVS model of Timed CSP can be employed.

9.3.1 Timed CSP, DORIS, and PVS

The PVS model of Timed CSP consists of four semantic models (see Figure 6.1, Page 138). Two of the semantic models are timed; the other two are untimed. One of each pair is a traces-only semantics; the second has failures as its observations.

With the assistance of dt, the DORIS model is easily implemented on top of Timed CSP/PVS.

In 1997, Tej and Wolff remarked that their very basic model of failures-divergence CSP[1] had taken about one man year just for the theory (not including five months for an earlier attempt) [117].

It is sobering to consider that many months of work have been invested in the PVS model described in this thesis, and that it is quite difficult to use. At times, the type system seems extremely strong, and coercing type conversions results in failure to determine that two equivalent expressions are indeed equal.

Chapters 6, 7, and 8 have illustrated that very simple rigorous or sketch proofs often have quite substantial formal proofs. With considerable effort, formal proofs can be constructed to represent the structure of the sketch proof. Indeed, no formal proof was successfully carried out without a sketch proof having been successfully completed before. Conversely,

[1]Another untimed model of CSP: see Page 38.

many faults with sketch proofs were detected as a result of the far greater rigour involved with the formal proof.

We will comment on the problems of scaling up this work in Section 9.4.2, and possible future approaches in Section 9.5.

9.3.2 DORIS, Timed CSP, and FDR

Again, the use of dt considerably aided the production of the case study models. However, FDR has a number of defects, the most severe of which is the fragility of the system to errors in scripts. A fault in a script is often not detected until late in the FDR cycle of load-compile-test, and even then, the fault often resulted in one component crashing without a useful diagnostic message. Debugging broken processes is easier with FDR than with PVS, but is still very difficult.

Fortunately, FDR is an otherwise excellent model checker, and bounded systems are coped with relatively quickly and effectively. Unfortunately, most of our systems were extremely large.

Modelling time could reasonably be expected to cause problems. It is not real-time that causes problems, but the unbounded nature of even the type nat. Indeed, one can argue that the systems modelled are digital in nature, and that the domain representing time should not be anything other than nat!

At the time of writing, development versions of FDR contain a lazy evaluator: this may assist with scaling up by not attempting to expand unbounded systems totally and immediately (which results in a failure of the model checker).

9.4 Remarks

9.4.1 Correctness of Designs

In the previous sections, we have described the use of PVS and FDR to formally analyse DORIS designs. It is important to note that we do not know for sure that our models are correct — nor can we ever be sure.

In the construction of the CSP theory in PVS, the author has a distinct idea of what is intended: PVS may interpret this differently. The only way to check is to introduce lemmas, and see that they are proved as expected, but this only increases assurance. We cannot take this as proof that the theory is ‘correct’. Indeed, we could easily introduce an axiom that is false.

Similarly, we have used CSP processes as specifications, yet we cannot be sure that these processes correctly model our intended systems. (This is an instance of the more general problem of formalising informal requirements.) Again, we can interrogate the tools, *e.g.* by asking whether a particular trace is in the set of observations of that process, but this is not totally convincing.

Finally, how do we know that the tools are operating correctly? They have certainly not been formally constructed: both PVS and FDR are considerable software engineering efforts, and can not easily be formally proved. Another tool could be used to check that the proofs generated by PVS are correct, although we then need to trust that tool.

In practice, we can take a more optimistic view: long use can allow a user to develop a trust in a tool; many results that are consistent allow a greater degree of confidence in the tool. This confidence will never be total, but will often be adequate for the problem concerned.

9.4.2 Industrial Scale

The PVS and FDR approaches to proving concepts about the IDAs and case studies are complementary: PVS handles inductive proofs about unbounded systems far better than FDR (which by its nature cannot handle unbounded systems). However, FDR does not require any interactive direction from the user at all, whereas PVS requires a lot of input.

It should be noted that the amount of detail required to carry out formal proofs is overwhelming. But this is only when attempting to formalize already existing proofs. In this case, much can be learnt about the nature of the statement being proved.

When no obvious proof can be determined, as is the case for the large case study, attempts to construct a formal proof are entirely doomed. It is this that leads us to the following conclusion:

> Retro-fitting proofs to systems that were designed without any intention of performing formal proofs is extremely difficult, and is likely to be uneconomical even when it is technically feasible for a particular system.

An 'ideal' of high-integrity system development starts by taking a very abstract, black-box specification. This should then be refined, and at each step, the proof obligations carefully documented. This would encourage an isolated style of component development, where the smallest possible interface is employed at each level. (Indeed, Jones has made similar remarks about the interference between concurrent components [55].) This can then be exploited by breaking monolithic proofs into a larger number of smaller lemmas.

This is in contrast to the large case study, where out of 324 resolved paths, 148 were resolved at the top-most parallel statement, and many of the rest were resolved at the next level down. This is despite the policy of the semantics to hide such details as low down the system as possible.

Further, refining a system from its initial specification would allow the work in this thesis to produce small, trusted components, rather than making futile attempts to reason about large systems. The difficulty here would be integrating this into current methods — this would require considerable changes to the notations and development methods.

9.4.3 PCI Metric

The previous section leads us to suggest a metric, which we call the 'proof-complexity/interface metric'. This assigns a higher value of 'badness' to systems which resolve paths higher than necessary.

For each resolved path, count the number of hierarchies traversed. Apply a weighting function (*e.g.* quadratic) to this height. The metric is then the mean of the resulting weighted values for each resolved path. The aim is to minimise the metric.

Obviously, a system will never be 'perfect' by this measure, but a real system never could be: a system design is a trade-off between many factors, including maintainability, cost, ease of design, understanding, *etc.* Here, we have to balance the various 'ideals' involved in these factors.

The metric proposed above aims to reduce the difficulties encountered with proofs about poorly structured designs.

This PCI metric is similar to metrics proposed for object-oriented designs [54], and it has been suggested that this PCI metric could have more general applicability to other design notations[2].

[2] Suggested by Richard Paige in private correspondence.

9.4.4 Alternative Approaches

DORIS is an example of a hierarchical timed design notation. Are there any lessons that can be learnt for such notations in general? Could a different approach to some parts of this work have resulted in greater success?

Several points can be immediately stated:

1. Graphical notations will always have a more severe problem of secondary notation than textual notations, *i.e.* by virtue of using the layout for imparting information, that layout is open to more misinterpretation than a simple text description (see Page 114 for the first remarks on this subject). However, note that the determination of route resolution (Appendix A.2, Page 216) is much clearer graphically than in text form, although this does not help us determine a simpler algorithm.

2. Hierarchical notations can be given intuitive, formal semantics if they are constructed in an encapsulated fashion, *i.e.* the individual components can be constructed and 'encapsulated' with a black box interface. In this thesis, the interface consists of those events indicating free IDA interactions, and processing events.

3. Timing does not necessarily have to be real-time: the natural numbers can be used, where the unit value corresponds to the smallest grain value needed in the application.

The greatest difficulties encountered concerned managing the detail in proofs: we claim that this is not specific to the choice of formalisms used in this thesis. Any formal analysis at this level of abstraction (*i.e.* modelling interactions of IDAs and processing) will require some reasoning about the interactions at that level.

We could change the level of abstraction, but we then miss out on the details which we need: the abstraction we chose matched the properties we wanted to reason about. An alternative abstraction would not help with this particular problem.

Alternatively, we could try a different formalism, say RTL [53], but we would still have difficulties because of the details of the model. Fowler's thesis [30] includes a description of using RTL within PVS (much as we have used Timed CSP within PVS), and still encountered many difficulties: these involved proof steps that were too small, and tactics that were too uncontrollable to produce meaningful results. It is interesting to contrast the relative sizes of the supporting theories: Fowler's RTL description takes approximately forty lines of PVS; the CSP-PVS theories described here (excluding the DORIS theories) consist of around 4000 lines of PVS (including all four semantic models) — but the general problems encountered were broadly the same. (The difference in size is accounted for as follows: CSP has many more primitive operators than RTL; we have also implemented multiple semantic models, whereas RTL only has one. The underlying representation of CSP as observations also requires more description than the corresponding RTL constructs.)

A further option would be to throw away modelling DORIS in terms of another formalism, and model it in terms of the basic entities provided by PVS: in his thesis, Stringer-Calvert suggests this for the subject of compiler verification [115]; this remark applies generally. This would be sensible in the short term, as it would overcome the problems of reasoning about structures relating to the formalism, but in the long term, we would have no formal base to reuse. A general goal of this type of work should be to provide carefully structured machine models of common formalisms so that they can be exploited further.

The conclusion is that if the models are at this level of abstraction (two or three point models describing reads, writes, and 'working'), then both rigorous and formal proofs will be difficult. An approach to manage this was based on using the abstract specifications of components in place of the components: this makes some difference to the magnitude of the problem. It also imposes the problem of determining the specification for a component when it hasn't been the subject of a refinement.

9.5 Future Work

A number of topics for future work can be highlighted:

9.5.1 General Formal Methods Tool Support

The greatest hindrance to industrial application of this work is the poor tool support. If we claim that we have modelled our systems at the correct level of abstraction, and that we have designed our semantics carefully to take advantage of the features provided by our semantic domain (*e.g.* Timed CSP's parallel operator), then we would make no significant changes to the mathematical part of the thesis, except to change the type of semantics. However, even changing to an alternative semantics would still have to manage the individual interactions that we model. Thus we conclude that an essential part of improving the applicability of this (type of) work is to improve formal methods tool support.

This is not a problem specific to this thesis: it is a problem widely reported in the formal methods literature, although it is often over-reported [8, 36].

All those concerned with formal methods tools, including SRI (responsible for PVS) and FSEL (FDR), are working on improving their tools to cope with larger models faster. It is reasonable to expect that with improved understanding of constructing the Timed CSP theories, and improvements to PVS and FDR (or other tools) that larger scale application will become possible.

9.5.2 Improved Proof Interfaces

The very nature of the systems examined means that the proofs being performed are large and detailed. A CADiℤ-like interface would be useful – this is a context sensitive hyperlinked interface [121]. More generally, the human-computer interface of the theorem provers (and model checkers) require much attention [71]. The integration of provers and model checkers suggested at the end of Chapter 8 would need better interfaces for the user to sensibly direct such a system.

9.5.3 More Investigation of IDAs and General Lemmas

A detailed library of IDAs and associated properties is needed. This thesis has only scratched the surface of possible work in this sense. Similarly, general lemmas about all the constructs in this thesis are needed.

The problem with this is that the underlying definitions have to be very carefully designed, and then frozen — because once work starts building on top of these definitions, changes become extremely expensive in terms of time and effort.

Given the difficulty encountered with large-scale formal analysis, most benefit to the software engineering process would probably be derived from exploring more general properties of the IDAs.

On Page 196, we commented that other plausible properties could be suggested and composed together to define new IDAs. Although the analysis of large systems was extremely difficult, this size of analysis is more profitable, and is likely to produce more benefits in the short-term.

9.5.4 Code Generation and Animation

An alternative to formal model checking or theorem proving could be to generate executable code, and either test it, or animate it (like UPPAAL [7]). Animation can be very useful for debugging and understanding a design — although animation only examines a small number of possible execution paths. At a specification level, animation can act as 'testing', *i.e.* 'Does the proposed specification do what it's meant to do?'.

Chapter 10

Conclusions

The thesis presented is that hierarchical design notations can be given a timed semantics, and that this semantics can be exploited. This exploitation using mechanical tools is limited by both the tools, and the underlying abstraction of the semantics, but useful and interesting results can be derived.

10.1 The Problem

The problem addressed in this thesis concerns the design of complex, embedded control systems that have safety and liveness requirements that can include time. In particular, many of these systems are critical: their failure can lead to loss of life, economic harm, or environmental damage. Examples of such systems can be found in the aerospace, automotive, nuclear, and defence industries.

Because of the potential consequences, and the large expense of many of these systems, they have to be designed so that they work 'first-time' when deployed. Strategies to ensure this success include structured analysis, testing, peer review of code, high-level languages, and formal methods [52].

Hierarchical design notations are increasingly used because they allow a system design to be decomposed into smaller and smaller components. DORIS is an example of such a design notation.

DORIS, as initially defined, does not have a formal meaning: we cannot reason about such designs. This thesis proposes an engineering solution for this problem.

10.2 The Solution

This thesis

- defines a syntax for DORIS;
- defines a denotational semantics using Timed CSP as the semantic domain;
- explores the issues involved with concurrent, shared variables;
- describes a PVS embedding of four semantic models of Timed CSP; and
- describes a prototype tool, dt, that checks designs according to a set of constraints, then produces output for Timed CSP/PVS, and the FDR model checker.

On page 24, we noted that Timed CSP can be viewed analogously to machine code. An additional view of these results is that as well as producing a formal timed semantics for DORIS, the DORIS syntax now provides a higher-level language for Timed CSP.

The problems with the solution are involved with the tool support. For example, how do we know that dt produces correct code? It certainly has not been developed formally.

As discussed in the previous chapter, we can never be sure about our programs or models — but further use will develop confidence, provided the results are consistent. Ultimately, software engineers could use the 'Holy Grail' [115] of formal development, which includes formally developed compilers, kernels, and tools. Even then, the hardware might still fail!

10.3 Final Remarks

A pragmatic approach to the use of formal methods in industry should involve a careful assessment of the expected benefits and costs of using a particular formal method.

In some cases, just formally specifying the system in question will bring significant benefits. The rigour involved in producing a formal specification can clarify many design issues that would only appear later in the development process [36]. Other times, several different formal notations used simultaneously can give multiple and useful views of the problem.

Even if a formal method is used on a project from initial specification, through refinement and complete proof, it can only increase assurance of correctness. It is simply not possible to have total assurance — all formal methods are based on models of systems. Thus hardware failure can invalidate the effort of formally producing a product. (Indeed, a failure at any part of the 'chain' of formal steps will invalidate the overall assurance.)

We must also consider which users are intended to use these methods. Different individuals handle different parts of the software life-cycle: where should formal methods fit in? The experience of theorem proving in this thesis suggests that non-specialists are not going to be able to prove assertions about their designs: but they could hand off their designs (which could be automatically generated, as suggested in Section 6.6, Page 155) to a small number of specialists.

Risk assessment could also play a part in choosing small parts of the system which need very high standards of assurance: the remaining parts could then be developed to 'normal' standards.

This thesis has produced a formal semantics for an industrial-strength design notation. It has illustrated the potential strengths —and current weaknesses— of the formal tool support for this semantics. Ultimately, the models described in this thesis can strengthen any DORIS-designed system, but more work is needed before this could be a routine activity in developing industrial-scale systems.

Part V

Appendices, Glossaries, and Bibliography

Appendix A

Semantics for DORIS

A.1 Static Semantics for DORIS

This appendix contains the material for Section 3.5 (Page 66).

Let

$$\begin{aligned} C \;=\; & \{\mathcal{SY}\} \\ & \cup \{\mathcal{CS}_i \mid i \in \{1, \ldots, N_S\}\} \\ & \cup \{\mathcal{CA}_i \mid i \in \{1, \ldots, N_A\}\} \\ & \cup \{\mathcal{DV}_i \mid i \in \{1, \ldots, N_D\}\} \end{aligned}$$

i.e. C is the set of all complex components, in the following rules:

1. All names of complex components are unique:

   $$\forall c, d \in C \bullet c_2 = d_2 \Rightarrow c = d$$

   (From the syntax, the second part of every complex component tuple is the name of that component.)

2. Within a particular complex component, every name referred to is either empty, or unique:

   $$\begin{aligned} & \forall c \in C \bullet \\ & \quad \forall i, j \geq 3 \bullet \\ & \quad\quad c_{i,2} = \langle empty\_string \rangle \end{aligned}$$

$$\begin{aligned}
&\qquad\qquad \vee\ c_{j,2} = \langle \mathit{empty\_string} \rangle \\
&\qquad\qquad \vee\ c_{i,2} \neq c_{j,2}
\end{aligned}$$

$$\begin{aligned}
&\forall c \in C \bullet \\
&\quad \forall i \geq 3 \bullet \\
&\qquad c_{i,2} = \langle \mathit{empty\_string} \rangle \Rightarrow (c_{i,1} = \mathsf{cp} \vee c_{i,1} = \mathsf{sp})
\end{aligned}$$

This ensures that hierarchical names are unique for everything except paths. (Paths are the only constructs that do not need to be referred to by name in the semantics.)

The $i, j \geq 3$ term (and terms like it) refers to the basic components within the complex component c: recall that c is a tuple, where the first element is the tag (Page 61), and the second element is its name. The remaining elements are the basic components contained in c.

Thus we can match the tag of basic component c_i by comparing it with $c_{i,1}$.

3. When working at the distributed level,[1] transputer links only refer to names of transputers; they also refer to distinct transputers within a particular link:

$$\begin{aligned}
&\forall i \geq 3 \bullet \\
&\quad \mathcal{SY}_i = (\mathsf{td}, n, s, d) \Rightarrow \\
&\qquad (\exists j, k \geq 3; j \neq k \bullet \mathcal{SY}_j = (\mathsf{tp}, s) \wedge \mathcal{SY}_k = (\mathsf{tp}, d))
\end{aligned}$$

$$\begin{aligned}
&\forall i \geq 3 \bullet \\
&\quad \mathcal{SY}_i = (\mathsf{ti}, n, s, d, v) \Rightarrow \\
&\qquad (\exists j, k, l \geq 3; j \neq k; j \neq l; k \neq l \bullet \\
&\qquad\quad \mathcal{SY}_j = (\mathsf{tp}, s) \wedge \mathcal{SY}_k = (\mathsf{tp}, d) \wedge \mathcal{SY}_l = (\mathsf{tp}, v))
\end{aligned}$$

Since the transputer and T-link components can only appear within the system component, we only need to check for consistency within the system component.

[1]When working at the design level, transputers (processors) do not appear in the language.

4. If a complex server is referred to inside a complex component, then there must be a complex server defined of that name:

$$\begin{array}{l} \forall c \in C \bullet \\ \quad \forall i \geq 3 \bullet \\ \qquad c_i = (\mathsf{cs}, t) \Rightarrow (\exists j \in \{1, \ldots, N_S\} \bullet t = \mathcal{CS}_{j,2}) \\ \qquad \vee\ c_i = (\mathsf{cs}, n!t) \Rightarrow (\exists j \in \{1, \ldots, N_S\} \bullet t = \mathcal{CS}_{j,2}) \end{array}$$

The first part of the disjunction refers to the case where there is only one instance of that complex server in c; the second part uses the $\langle Instance \rangle$"!"$\langle Target \rangle$ notation to allow multiple instances of a particular $\langle Target \rangle$.

5. Similarly, complex activities must exist to be referred to:

$$\begin{array}{l} \forall c \in C \bullet \\ \quad \forall i \geq 3 \bullet \\ \qquad c_i = (\mathsf{ca}, t) \Rightarrow (\exists j \in \{1, \ldots, N_A\} \bullet t = \mathcal{CA}_{j,2}) \\ \qquad \vee\ c_i = (\mathsf{ca}, n!t) \Rightarrow (\exists j \in \{1, \ldots, N_A\} \bullet t = \mathcal{CA}_{j,2}) \end{array}$$

6. Similarly, devices too must exist to be referred to:

$$\begin{array}{l} \forall c \in C \bullet \\ \quad \forall i \geq 3 \bullet \\ \qquad c_i = (\mathsf{dv}, t) \Rightarrow (\exists j \in \{1, \ldots, N_D\} \bullet t = \mathcal{DV}_{j,2}) \\ \qquad \vee\ c_i = (\mathsf{dv}, n!t) \Rightarrow (\exists j \in \{1, \ldots, N_D\} \bullet t = \mathcal{DV}_{j,2}) \end{array}$$

7. Cycles of complex components are not allowed:

$$\begin{array}{l} \neg \exists c_1, \ldots, c_n \in C \bullet \\ \quad \exists i_1, \ldots, i_n \geq 3 \bullet \\ \qquad \forall j \in \{1, \ldots, n\} \bullet \\ \qquad\quad c_{j,i_j,1} \in \{\mathsf{cs}, \mathsf{ca}, \mathsf{dv}\} \\ \qquad\quad \wedge\ c_{j+1 mod n,1} = c_{j,i_j,1} \\ \qquad\quad \wedge\ \quad c_{j,i_j,2} = t \wedge t = c_{j+1 mod n,2} \\ \qquad\qquad \vee\ c_{j,i_j,2} = n!t \wedge t = c_{j+1 mod n,2} \end{array}$$

Essentially, this says that there does not exist a sequence of complex components such that they form a cycle (by matching the tag and the name).

8. Timed CSP 'code stubs'[2] should be available for any simple server or simple activity:

$$\begin{aligned}
&\forall c \in C \bullet \\
&\quad \forall i \geq 3 \bullet \\
&\qquad\quad c_i = (\mathsf{ss}, t) \Rightarrow t \in \mathit{Code\_Stubs\_Required} \\
&\qquad \vee\ c_i = (\mathsf{ss}, n!t) \Rightarrow t \in \mathit{Code\_Stubs\_Required}
\end{aligned}$$

$$\begin{aligned}
&\forall c \in C \bullet \\
&\quad \forall i \geq 3 \bullet \\
&\qquad\quad c_i = (\mathsf{sa}, t) \Rightarrow t \in \mathit{Code\_Stubs\_Required} \\
&\qquad \vee\ c_i = (\mathsf{sa}, n!t) \Rightarrow t \in \mathit{Code\_Stubs\_Required}
\end{aligned}$$

9. Definitions are also required for every type of route mentioned:

$$\begin{aligned}
&\forall c \in C \bullet \\
&\quad \forall i \geq 3 \bullet \\
&\qquad c_i = (\mathsf{rt}, t, r) \Rightarrow r \in \mathit{Routes\_Required}
\end{aligned}$$

The constraints on paths are described in Section 3.8 (Page 73).

A.2 Path Resolution Algorithm

This appendix provides supplementary information for Section 3.8.1 (Page 73).

We can give a syntax for the interfaces on either end of a path:

$$\begin{aligned}
\langle \mathit{Local\_Interface\_Spec} \rangle &::= \langle \mathit{Name} \rangle \text{“:”} \langle \mathit{Interface\_Spec} \rangle \\
\langle \mathit{Parent\_Interface\_Spec} \rangle &::= \langle \mathit{Interface\_Spec} \rangle \\
\langle \mathit{Complex\_Interface\_Spec} \rangle &::= \langle \mathit{Interface\_Spec} \rangle \text{“:”} \langle \mathit{Complex\_Channel} \rangle \\
\langle \mathit{Interface\_Spec} \rangle &::= (\text{“p”} \mid \text{“w”}) \\
&\qquad \langle \mathit{positive\_number} \rangle \\
\langle \mathit{Complex\_Channel} \rangle &::= \langle \mathit{lower\_case\_alpha} \rangle
\end{aligned}$$

[2]'Code stubs' are Timed CSP expressions that represent the component concerned.

$\langle Path\_Source \rangle$ and $\langle Path\_Destination \rangle$ are then defined as specified by the following categorisation:

Simple paths can be split into several distinct categories:

Simple-local Both the source and destination of the path are of the form 't:i' ($\langle Local\_Interface\_Spec \rangle$), where t is the name of a local component with tag cs, ca, ss, sa, dv, or rt. One end should be a port; the other should be a window.

(In the syntax above, 'p' denotes a port: these can only appear on components with tag cs, ca, ss, or sa. 'w' denotes a window, which can only appear on component tags cs, ss, dv, and rt.)

Simple-parent One end of the path has syntax $\langle Local\_Interface\_Spec \rangle$ and the other end has syntax $\langle Parent\_Interface\_Spec \rangle$. Both ends should be of the same interface type; *i.e.* both ports or both windows.

This effectively 'renames' a port or window; a reference to the parent interface results in a reference of the local interface.

Simple-complex Again, one end of the path has syntax $\langle Local\_Interface\_Spec \rangle$, but the other has syntax $\langle Complex\_Interface\_Spec \rangle$. Both ends should have the same interface type.

The $\langle Complex\_Channel \rangle$ identifies which part of a complex (multiplexed) path the local interface should be identified with.

Complex paths are also split into several categories:

Complex-local Both the source and destination of the path have syntax $\langle Local\_Interface\_Spec \rangle$, and the interface types should differ (*i.e.* one end should be a port; the other should be a window). The same constraints on component types as for simple-local paths apply.

Complex-parent One end of the path has syntax $\langle Local\_Interface\_Spec \rangle$; the other has syntax $\langle Parent\_Interface\_Spec \rangle$. Both ends should have the same interface type.

No other format of path sources and destinations is permitted in this semantics.

The following algorithm effectively 'grows' the interfaces up from the leaves of the hierarchy until they are resolved.

When partial systems are being described, there is the possibility that some ports and windows will not be resolved, because they are already at the 'parent' of the top-most complex component. In this case, they are 'free' (not hidden) events, *i.e.* $\mathcal{C}_I$ is non-empty.

The general algorithm for path-interface resolution is as follows:

1. Identify all the *simple interfaces*, *i.e.* all the $\langle Local_Interface_Spec \rangle$s applying to simple servers, simple activities, and routes.

2. Repeat the following steps until nothing more can be done.

3. Examine each complex component for simple-local paths: these connect two simple interfaces, and provide the final resolution (*e.g.* the window of a route to the port of an activity).

4. Apply every simple-parent path by looking for a local interface that matches the path: these result in the creation of a *simple renaming* for the local interface. The renamed path takes the name of the parent (and the new $\langle Interface\_Spec \rangle$).

5. Apply every simple-complex path by looking for a local interface that matches the path: these result in the creation of a *simple-complex interface*. The interface takes the name of the parent, but has an $\langle Interface\_Spec \rangle$ and a $\langle Complex\_Channel \rangle$. The path that connects to the parent end of the current path must be a complex path (this is now a multiplexed path — this is a mechanism for saying 'lots of paths go from this component to this component').

6. Apply every complex-parent path by looking for a local interface that matches the path: like the simple-parent path, these result in the creation of a *complex renaming* for the local (complex) interface.

7. Apply every complex-local path to local interfaces: these resolve the complex interfaces. These can link many simple ports and windows; the final resolution is determined by the $\langle Complex\_Channel \rangle$ values, where like matches like.

Figures A.1 and A.2 can be used to illustrate this algorithm. (This is the same example as in Section 3.8.1.) There are eight paths (including at least one of each of the five categories described above):

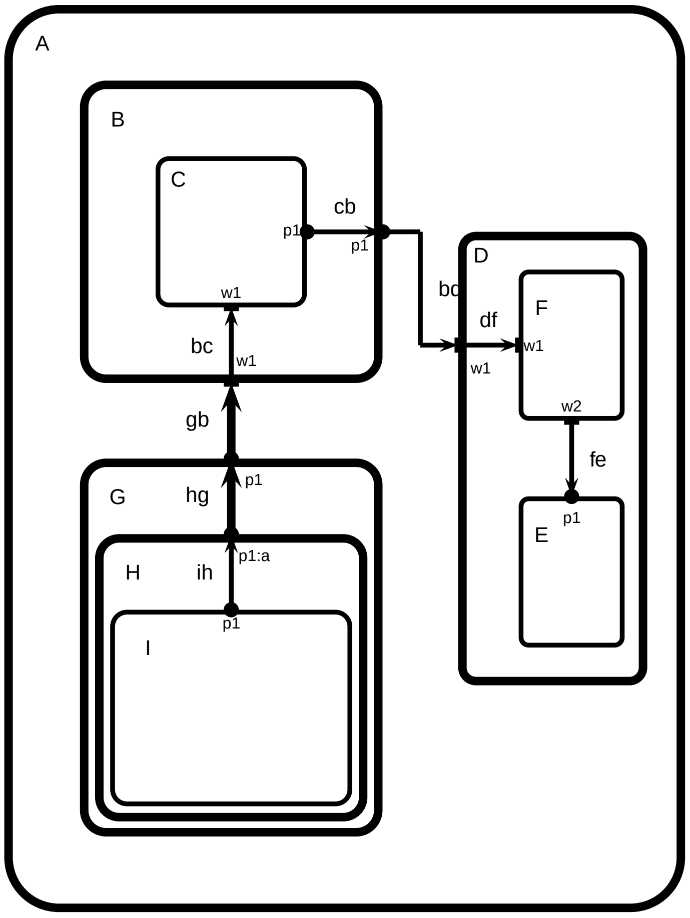
bc	simple-complex	ih	simple-parent
cb	simple-parent	bd	simple-local
gb	complex-local	df	simple-parent
hg	complex-parent	fe	simple-local

The simple interfaces defined by these paths (including the full hierarchical names) are:

A.B.C:w1	A.B.C:p1	A.G.H.I:p1
A.D.F:w1	A.D.F:w2	A.D.E:p1

These are resolved thus:

A.D.F:w2	$\rightarrow$	A.D.E:p1	because fe links them directly
A.B.C:p1	$\rightarrow$	A.D.F:w1	via cb, bd, and df
A.G.H.I:p1	$\rightarrow$	A.B.C:w1	via ih, hg, gb, and bc on complex channel a



Described on Page 218

Textual version in Figure A.2 (Page 221)

Figure A.1: Illustration of path resolution algorithm

```
(     (sy, A,
            (cs, B),
            (cs, D),
            (ca, G),
            (cp, gb, G:p1, B:w1),
            (sp, bd, B:p1, D:w1))
      (cs, B,
            (ss, C),
            (sp, bc, w1:a, C:w1),
            (sp, cb, C:p1, p1))
      (cs, D,
            (sa, E),
            (rt, F, Channel),
            (sp, df, w1, F:w1),
            (sp, fe, F:w2, E:p1))
      (ca, G,
            (ca, H),
            (cp, hg, H:p1, p1))
      (ca, H,
            (sa, I),
            (sp, ih, I:p1, p1:a))     )
```

Code stubs expected are:	simple server C
	simple activity E
	simple activity I
	route type Channel

Described on Page 218

Graphical version in Figure A.1 (Page 220)

Figure A.2: Textual statement of path resolution example

Appendix B

Prototype DORIS Tool

B.1 ASCII Syntax for DORIS Systems

This syntax closely follows the definition given in Section 3.3 (Page 58), and uses the simplified BNF syntax given on Page 62. One or more literal newlines are represented by ⟨newline⟩.

A file containing the definition of a DORIS system consists of a series of complex construct definitions:

$$\begin{aligned} \langle \mathit{System} \rangle \quad ::= \quad & (\langle \mathit{System\_Design}' \rangle \mid \langle \mathit{System\_Distributed}' \rangle) \\ & \{\langle \mathit{Complex\_Server}' \rangle \\ & \ \mid \langle \mathit{Complex\_Activity}' \rangle \\ & \ \mid \langle \mathit{Device}' \rangle\} \end{aligned}$$

Each of $\langle \mathit{System\_Design}' \rangle$, $\langle \mathit{System\_Distribution}' \rangle$, $\langle \mathit{Complex\_Server}' \rangle$, $\langle \mathit{Complex\_Activity}' \rangle$, and $\langle \mathit{Device}' \rangle$ is defined by a name, and a list of other constructs contained within it. $\langle \mathit{Ref} \rangle$ is a reference note: in the large case study (Section 8.5), these are the page numbers of the original network diagram of the construct concerned.

$$\begin{aligned} \langle \mathit{System\_Design}' \rangle \quad ::= \quad & \text{“sy:”}\langle \mathit{Name} \rangle\text{“:”}\langle \mathit{Ref} \rangle\langle \text{newline} \rangle \\ & \{\langle \mathit{Component} \rangle\}^{+}\text{“::”}\langle \text{newline} \rangle \\ \langle \mathit{System\_Distribution}' \rangle \quad ::= \quad & \text{“sy:”}\langle \mathit{Name} \rangle\text{“:”}\langle \mathit{Ref} \rangle\langle \text{newline} \rangle \\ & \{\langle \mathit{Transputer} \rangle \mid \langle \mathit{Direct\_T\_Link} \rangle \\ & \mid \langle \mathit{Indirect\_T\_Link} \rangle \mid \langle \mathit{Component} \rangle\}^{+} \end{aligned}$$

$$
\begin{array}{rcl}
 & & \text{“::”}\langle\text{newline}\rangle \\
\langle \mathit{Complex\_Server'} \rangle & ::= & \text{“cs:”}\langle \mathit{Name} \rangle\text{“:”}\langle \mathit{Ref} \rangle\langle\text{newline}\rangle \\
 & & \{\langle \mathit{Component} \rangle\}^{+} \\
 & & \text{“::”}\langle\text{newline}\rangle \\
\langle \mathit{Complex\_Activity'} \rangle & ::= & \text{“cc:”}\langle \mathit{Name} \rangle\text{“:”}\langle \mathit{Ref} \rangle\langle\text{newline}\rangle \\
 & & \{\langle \mathit{Component} \rangle\}^{+} \\
 & & \text{“::”}\langle\text{newline}\rangle \\
\langle \mathit{Device'} \rangle & ::= & \text{“dv:”}\langle \mathit{Name} \rangle\text{“:”}\langle \mathit{Ref} \rangle\langle\text{newline}\rangle \\
 & & \{\langle \mathit{Passive} \rangle\}^{+} \\
 & & \text{“::”}\langle\text{newline}\rangle \\
\langle \mathit{Component} \rangle & ::= & \langle \mathit{Active} \rangle \mid \langle \mathit{Passive} \rangle \\
\langle \mathit{Active} \rangle & ::= & \langle \mathit{Complex\_Server} \rangle \\
 & & \mid \langle \mathit{Complex\_Activity} \rangle \\
 & & \mid \langle \mathit{Simple\_Server} \rangle \\
 & & \mid \langle \mathit{Simple\_Activity} \rangle \\
\langle \mathit{Passive} \rangle & ::= & \langle \mathit{Device} \rangle \\
 & & \mid \langle \mathit{Route} \rangle \\
 & & \mid \langle \mathit{Complex\_Path} \rangle \\
 & & \mid \langle \mathit{Simple\_Path} \rangle
\end{array}
$$

The individual 'simple' constructs are defined as a single line each:

$$
\begin{array}{rcl}
\langle \mathit{Transputer} \rangle & ::= & \text{“tp:”}\langle \mathit{Name} \rangle\langle\text{newline}\rangle \\
\langle \mathit{Direct\_T\_Link} \rangle & ::= & \text{“td:”}\langle \mathit{Name} \rangle\text{“:”}\langle \mathit{T\_Source} \rangle \\
 & & \text{“:”}\langle \mathit{T\_Destination} \rangle\langle\text{newline}\rangle \\
\langle \mathit{Indirect\_T\_Link} \rangle & ::= & \text{“ti:”}\langle \mathit{Name} \rangle\text{“:”}\langle \mathit{T\_Source} \rangle \\
 & & \text{“:”}\langle \mathit{T\_Destination} \rangle\text{“:”}\langle \mathit{T\_Via} \rangle\langle\text{newline}\rangle \\
\langle \mathit{Complex\_Server} \rangle & ::= & \text{“cs:”}\langle \mathit{Name} \rangle\langle\text{newline}\rangle \\
\langle \mathit{Complex\_Activity} \rangle & ::= & \text{“ca:”}\langle \mathit{Name} \rangle\langle\text{newline}\rangle \\
\langle \mathit{Simple\_Server} \rangle & ::= & \text{“ss:”}\langle \mathit{Name} \rangle\langle\text{newline}\rangle \\
\langle \mathit{Simple\_Activity} \rangle & ::= & \text{“sa:”}\langle \mathit{Name} \rangle\langle\text{newline}\rangle \\
\langle \mathit{Device} \rangle & ::= & \text{“dv:”}\langle \mathit{Name} \rangle\langle\text{newline}\rangle \\
\langle \mathit{Route} \rangle & ::= & \text{“rt:”}\langle \mathit{Name} \rangle\text{“:”}\langle \mathit{Route\_Type} \rangle\langle\text{newline}\rangle
\end{array}
$$

$$\begin{array}{rcl}
\langle Complex\_Path\rangle & ::= & \text{“cp:”}\langle Name\rangle\text{“:”}\langle Path\_Source\rangle \\
 & & \text{“:”}\langle Path\_Destination\rangle\langle\text{newline}\rangle \\
\langle Simple\_Path\rangle & ::= & \text{“sp:”}\langle Name\rangle\text{“:”}\langle Path\_Source\rangle \\
 & & \text{“:”}\langle Path\_Destination\rangle\langle\text{newline}\rangle
\end{array}$$

The final part of this syntax is filling in the gaps that were left unspecified in Section 3.3. Here, a $\langle String\rangle$ is a (possibly empty) string of letters, digits, and underscores.

$$\begin{array}{rcl}
\langle Name\rangle & ::= & \langle String\rangle \\
\langle Path\_Source\rangle & ::= & \langle String\rangle\text{“:”}\langle String\rangle \\
\langle Path\_Destination\rangle & ::= & \langle String\rangle\text{“:”}\langle String\rangle \\
\langle T\_Source\rangle & ::= & \langle String\rangle \\
\langle T\_Destination\rangle & ::= & \langle String\rangle \\
\langle T\_Via\rangle & ::= & \langle String\rangle \\
\langle Route\_Type\rangle & ::= & \langle String\rangle
\end{array}$$

B.2 Example

The following example is drawn from the input file to the DORIS prototype tool for the large case study (Section 8.5, Page 185). '[...]' denotes deleted text.

```
sy:EPU_Software:0
tp:T1
tp:T2
[...]
tp:T6
[...]
td::T1:T3
td::T1:T4
[...]
td::T5:T6
ti::T1:T2:T3
ti::T1:T6:T5
```

```
[. . . ]
ti::T4:T6:T5
dv:T5_T6_Link
dv:T4_T5_Link
[. . . ]
dv:T2_T3_Link
dv:T4_to_T6
dv:T6_to_T1
[. . . ]
dv:T5_to_T2
cp:Get_T6_to_T5:T5_T6_Link:w3:T5:p6
cp:Put_T5_to_T6:T5:p5:T5_T6_Link:w4
[. . . ]
cp:Put_T2_to_T5:T2:p9:T2_to_T5:w1
::

cc:T1:1
cs:T1_Tele
sa:Clone_Global_Mode_T1:H
sa:Image_State_Control:L
ca:Generate_and_Organize_Objects
rt:ISC_Missile_Mode:pool
sp:Rd_Report_Best_Trgts_Tele:p1:b:T1_Tele:p2
sp:Rd_Set_Trgt_Near_Tele:p1:c:T1_Tele:p3
[. . . ]
sp:Recv_PO_Body_Processed:p6:a:Generate_and_Organize_Objects:p1
::

[. . . ]

cc:Assess_Target_Objects:10
rt:CA_IP_State:Pool
rt:Previous_Thresholds:Pool
rt:Scene_Stats_Data:Pool
rt:Image_Control:Pool
rt:Process_Trgt_Data:Signal
rt:Primary_Number:Pool
rt:PPD_Body_Processed:Signal
```

```
rt:PPD_IP_State:Pool
rt:PPD_Image_TIO:Signal
rt:IP_State_Available:Signal
sa:Control_Assessment:L
sa:Process_Patch_Data:L
sa:Process_Target_Body:L
ss:Multiple_Readers_CA:H
sp::Process_Target_Body:p6:p1:
sp::Process_Target_Body:p7:p2:
[...]
sp::Multiple_Readers_CA:w2:Process_Patch_Data:p5
::

[...]

cs:Actuator_IF:20
rt:AS_Data_In:Pool
rt:AS_Data_Out:Pool
sa:Manage_Actuator:H
ss:Actuator
sp::w1::AS_Data_Out:w1
sp::Manage_Actuator:p7:p2:
[...]
sp::Actuator:w2:Manage_Actuator:p12
::

[...]

dv:T1_T3_Link:113
dv:T1_to_T3
dv:T3_to_T1
cp::w1::T1_to_T3:w1
cp::T1_to_T3:w2:w4:
cp::w3::T3_to_T1:w2
cp::T3_to_T1:w1:w2:
::

[...]
```

```
dv:T1_to_T2:212
rt:Tracking_Point:pool
sp::w1:a:Tracking_Point:w2
sp::Tracking_Point:w1:w2:a
::

[. . . ]
```

B.3 Algorithm

The algorithm in the prototype tool (dt) is as follows:

1. Read input files.
2. Write out some statistics about the input files.
3. Match the code stubs to the simple servers and simple activities.
4. Find the definitions of all complex components.
5. For each 'context run', *i.e.* for every system component, or complex component indicated on the command-line:
   (a) Build a tree of contexts, where every complex component has edges leading from it, until all the leaves are basic components.
   (b) Determine all the simple interfaces.
   (c) By a depth-first traversal of the context tree, resolve the simple interfaces.
   (d) Using the context information, construct forwarding and scheduling records.
   (e) Write out the various output formats.

Appendix C

PVS Model of Timed CSP

The interface for the PVS model of Timed CSP is described in this appendix. Section 2.4 introduces Timed CSP (Page 33).

Throughout this appendix, parts of the PVS source are included — the extracts are not complete, as the remains are omitted for brevity.

C.1 Basic Concepts

We now define a number of basic concepts.

C.1.1 Events and Alphabets

The theory `basic` is parametrised by the nonempty set `SEvents` ('events including success'), and the distinguished event, `success` (representing $\checkmark$).

```
basic [SEvents : NONEMPTY_TYPE,
       success : Events] : THEORY
```

This set contains all possible events that could occur (in the given context).

An `Alphabet` is then a (not necessarily strict) subset of `Events`, and `SAlphabet` is a subset of `SEvents`.

```
        Alphabet : TYPE = setof[Events]
        SAlphabet : TYPE = setof[SEvents]
```

```
SubAlphabet(A1 : Alphabet) : TYPE
        =  A2 : Alphabet |
             FORALL (e : Events) : A2(e)
                            IMPLIES A1(e)
SubSAlphabet(A1 : SAlphabet) : TYPE
        =  A2 : SAlphabet |
             FORALL (e : Events) : A2(e)
                            IMPLIES A1(e)
```

C.1.2 Observations: Untimed Traces

A `Trace` is a sequence of events (including ✓).

```
Trace : TYPE = list[SEvents]
```

A single observation in the untimed traces (UT) model, `ObsUT`, is a `Trace`. A set of such observations is `ObsSetUT`.

```
ObsUT : TYPE = Trace
ObsSetUT : TYPE = setof[ObsUT]
```

C.1.3 Observations: Untimed Failures

A `Refusal` set is a set of `Events` (including ✓).

```
Refusal : TYPE = setof[SEvents]
```

An observation in the untimed failures (UF) model is the ordered pair consisting of a `Trace`, and a `Refusal`.

```
ObsUF : TYPE = [Trace, Refusal]
ObsSetUF : TYPE = setof[ObsUF]
```

C.1.4 Observations: Timed Traces

We now introduce the type for `Time`.

```
Time : TYPE = { x : real | x >= 0 }
```

`Time`s are non-negative real numbers. Note that PVS cannot understand real number constants, but that real numbers can still be reasoned about.

A `TimedEvent` is then a pair with a `Time` and an `Event`.

```
TimedEvent : TYPE = [Time, SEvents]
```

Analogously to the untimed traces case, the timed traces (TT) model is defined in terms of observations on `TimedTrace`s.

```
TimedTrace : TYPE = list[TimedEvent]
ObsTT : TYPE = TimedTrace
ObsSetTT : TYPE = setof[ObsTT]
```

C.1.5 Observations: Timed Failures

The last model that we describe is the timed failures (TF) model. This requires a definition of `TimedRefusal`.

```
TimedRefusal : TYPE = setof[TimedEvent]
```

A `TimedRefusal` consists of a set of `TimedEvent`s.

An observation in the TF model is then an observation in the TT model augmented with a `TimedRefusal`.

```
ObsTF : TYPE = [TimedTrace, TimedRefusal]
ObsSetTF : TYPE = setof[ObsTF]
```

C.1.6 Useful Functions

In this theory, a number of useful functions are defined. These are described here, since they are used in the remaining sections.

These projections provide a more memorable and easy to read notation for extracting components of tuples.

```
Time(e : TimedEvent) : Time = proj_1(e)
Event(e : TimedEvent) : Events = proj_2(e)
Trace(b : ObsUF) : Trace = proj_1(b)
Refusal(b : ObsUF) : Refusal = proj_2(b)
```

This projection uses the `Event` projection to extract an untimed trace from a timed trace.

```
project(t : TimedTrace) : Trace
        = map(Event)(t)
```

The functions `sigma` provide the PVS version of σ ('set of'). They return the events that are referred to in the (timed) trace.

```
sigma(t : Trace) : SAlphabet =  list2set(t)
sigma(t : TimedTrace) : SAlphabet
                      = sigma(project(t))
```

`prefix` returns true if `s1` is a prefix of `s2`; otherwise false is returned.

```
prefix(s1 : Trace, s2 : Trace) : bool
        = EXISTS (s : Trace) :
                    s2 = append(s1, s)
```

C.2 Interface

The next four sections define the types `ProcessUT`, `ProcessUF`, `ProcessTT`, and `ProcessTF`. In each of those sections, we define the CSP processes for each type of process.

These definitions, with the exception of the process type, have identical formal parameters. These are described here. In each case, `Process` should be substituted with one of the four process types. Other parameter types that will be encountered are

`RenameFn` an injective function from `Events` to `Events`

`VProcess` a vector of `Process`

`PMap` a function from `Process` to `Process`

`VPMap` a function from `VProcess` to `VProcess`

`VAlphabet` a vector of `Alphabet`

C.2.1 Base Processes

These processes are defined as sets of observations.

$Stop$ **(deadlock)**

```
        Stop : Process
```

$Chaos$ **(most nondeterministic process)**

```
        Chaos(A : Alphabet) : Process
```

$Skip$ **(successful process)**

```
        Skip : Process
```

Run **(will engage in any event always)**

```
        Run(A : Alphabet) : Process
```

$e : E \to P_e$ **(general choice)**

```
        GChoice(E : Alphabet)
               (EP : [(E) -> Process]) : Process
```

$P \Box Q$ **(external choice)**

```
        EChoice(P, Q : Process) : Process
```

$P \sqcap Q$ **(internal choice)**

```
        IChoice(P, Q : Process) : Process
```

$P; Q$ **(sequential composition)**

```
        SComp(P, Q : Process) : Process
```

$P \|_E Q$ **(hybrid parallel)**

```
        Par(E : Alphabet)(P, Q : Process) : Process
```

$P \backslash E'$ **(hide)**

```
        Hide(E : Alphabet, P : Process) : Process
```

Note that the second parameter, `E`, is the alphabet of that is revealed to external observers, whereas E' is the set of events that are hidden.

Renaming

```
        Rename(P : Process, F : RenameFn) : Process
```

$\mu X \bullet F(X)$ **(recursion)**

```
        LFP(F : PMap) : Process

        VLFP(N : nat)(F : VPMap(N)) : Process
```

(Recursion is handled in this model by a least fixed point operator.) The first form is a single process; the second form allows mutually recursive processes.

$P \overset{t}{\triangleright} Q$ **(timeout)**

```
        Timeout(t : Time, P, Q : Process) : Process
```

Note that, in an untimed model, this can be defined as

$$P \overset{t}{\triangleright} Q = (P \,\Box\, Q) \sqcap Q$$

C.2.2 Derived Processes

These processes are defined in terms of other processes, rather than as observations.

$a \rightarrow P$ **(prefix)**

```
        Prefix(a : Events, P : Process) : Process
```

Prefix is defined as a general choice which offers only one event.

$P||Q$ **(synchronized parallel)**

```
        SPar(P, Q : Process) : Process
```

This is defined in terms of `Par`, with a synchronization alphabet `Sigma`.

$P|||Q$ **(asynchronous parallel)**

```
        APar(P, Q) : Process
```

Again, this is defined in terms of `Par`, this time with synchronization alphabet `emptyset`.

$||^i_{A_i} P_i$ **(network parallel)**

```
        NPar(N : nat)(A : VAlphabet(N),
                      P : VProcessUT(N)) : Process
```

Network parallel takes two vectors of equal cardinality, `A`, the interfaces, and `P`, the processes.

$||^i P_i$ **(general synchronized parallel)**

```
        SPar(N : nat)(P : VProcess(N)) : Process
```

We can define general (rather than binary) variants of many operators.

$|||^i P_i$ **(general asynchronous parallel)**

```
        APar(N : nat)(P : VProcess(N)) : Process
```

$||^i_E P_i$ **(general hybrid parallel)**

```
        Par(N : nat)(E : Alphabet)(P : VProcess(N))
                                         : Process
```

There is no reason why we shouldn't continue to define further general operators for sequential composition, choice, *etc.* They can be easily added to the theories: they have not been at this point because they were not needed.

$Wait\ t$ **(timed delay)**

```
        Wait(t : Time) : Process
```

$Wait$ is defined thus:

$$Wait\ t = Stop \overset{t}{\triangleright} Skip$$

Appendix D

IDA Programs

Figures D.1–D.4 are those referred to in Section 6.5.3 (Page 154).

$$\begin{tikzcd}
Constant \arrow[dr] & & & \\
& ss \arrow[r, "sw.d"] \arrow[d, "sr"] & ms \arrow[r, "mw.d"] \arrow[d, "sr"] & es \arrow[ll, bend right, "ew.d"'] \arrow[d, "sr"] \\
& sm \arrow[r, "sw.d"] \arrow[d, "mr"] & mm \arrow[r, "mw.d"] \arrow[d, "mr"] & em \arrow[ll, bend left, "ew.d"] \arrow[d, "mr"] \\
& se \arrow[r, "sw.d"] \arrow[uu, bend left, "er.d_0"] & me \arrow[r, "mw.d"] \arrow[uu, bend left, "er.d_0"] & ee \arrow[ll, bend left, "ew.d"] \arrow[uu, bend right, "er.d_0"']
\end{tikzcd}$$

Figure D.1: State machine for $Constant$

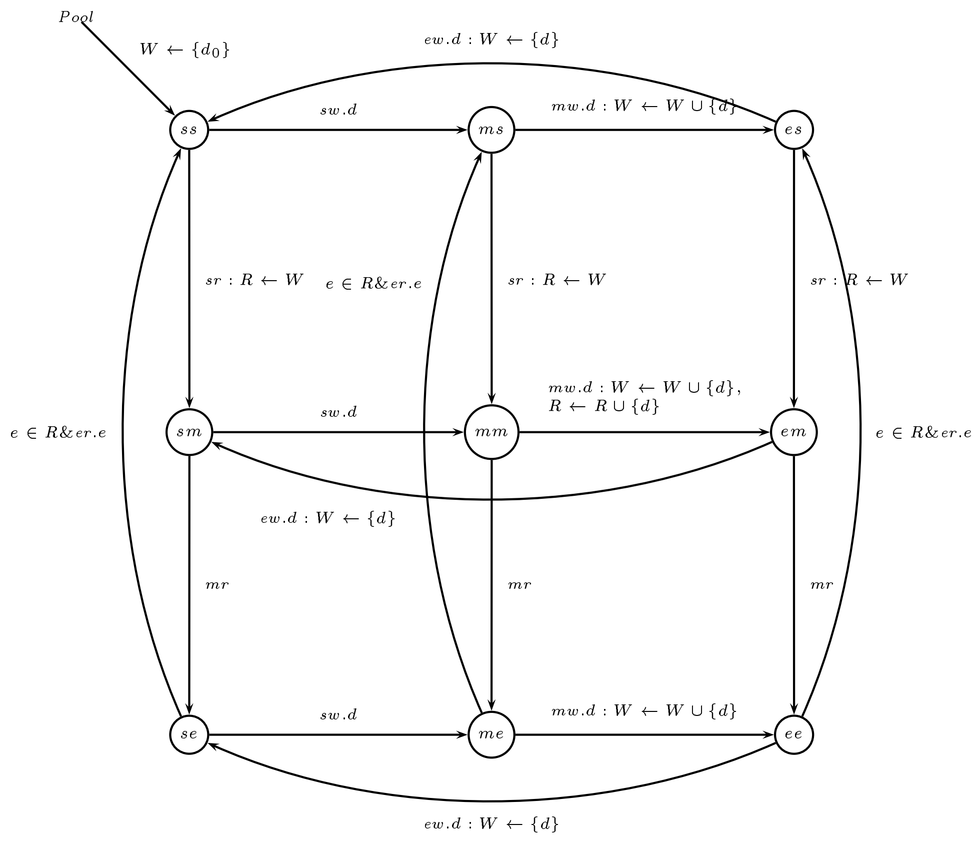

$Pool$
$W \leftarrow \{d_0\}$
$ew.d : W \leftarrow \{d\}$
$sw.d$
$mw.d : W \leftarrow W \cup \{d\}$
ss
ms
es
$sr : R \leftarrow W$
$e \in R \& er.e$
$sr : R \leftarrow W$
$sr : R \leftarrow W$
$mw.d : W \leftarrow W \cup \{d\},$
$R \leftarrow R \cup \{d\}$
$e \in R \& er.e$
sm
$sw.d$
mm
em
$e \in R \& er.e$
$ew.d : W \leftarrow \{d\}$
mr
mr
mr
$sw.d$
$mw.d : W \leftarrow W \cup \{d\}$
se
me
ee
$ew.d : W \leftarrow \{d\}$

Figure D.2: State machine for $Pool$

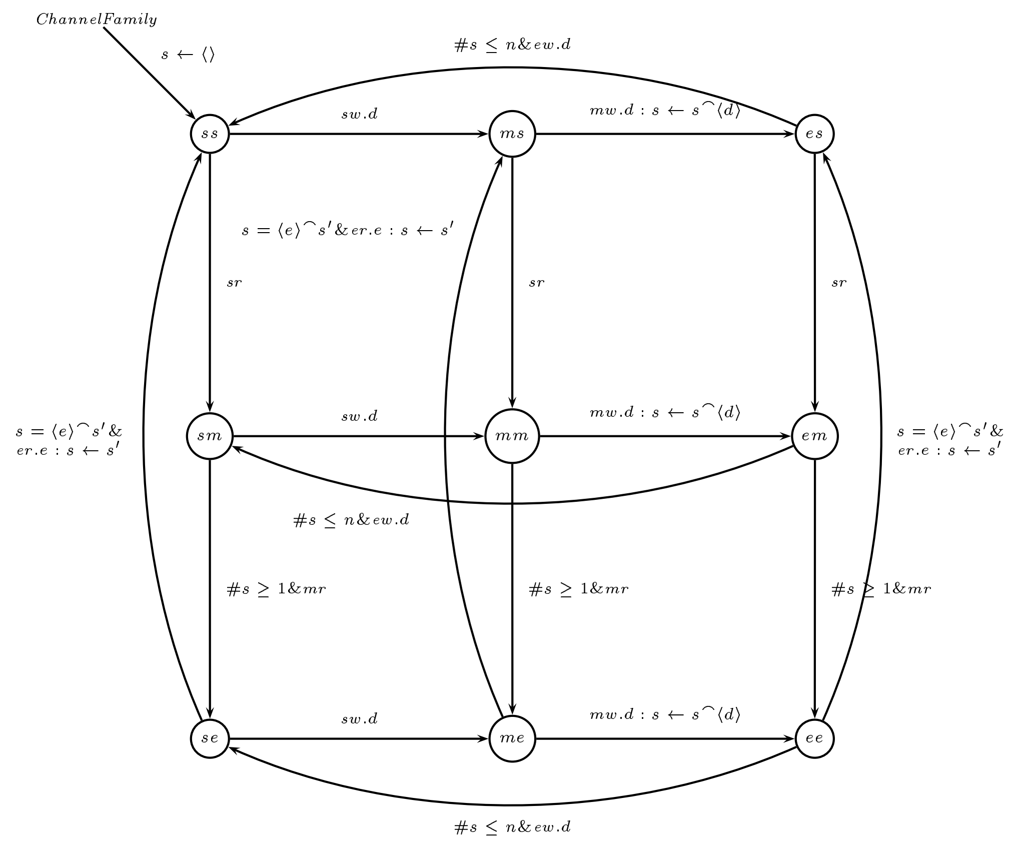
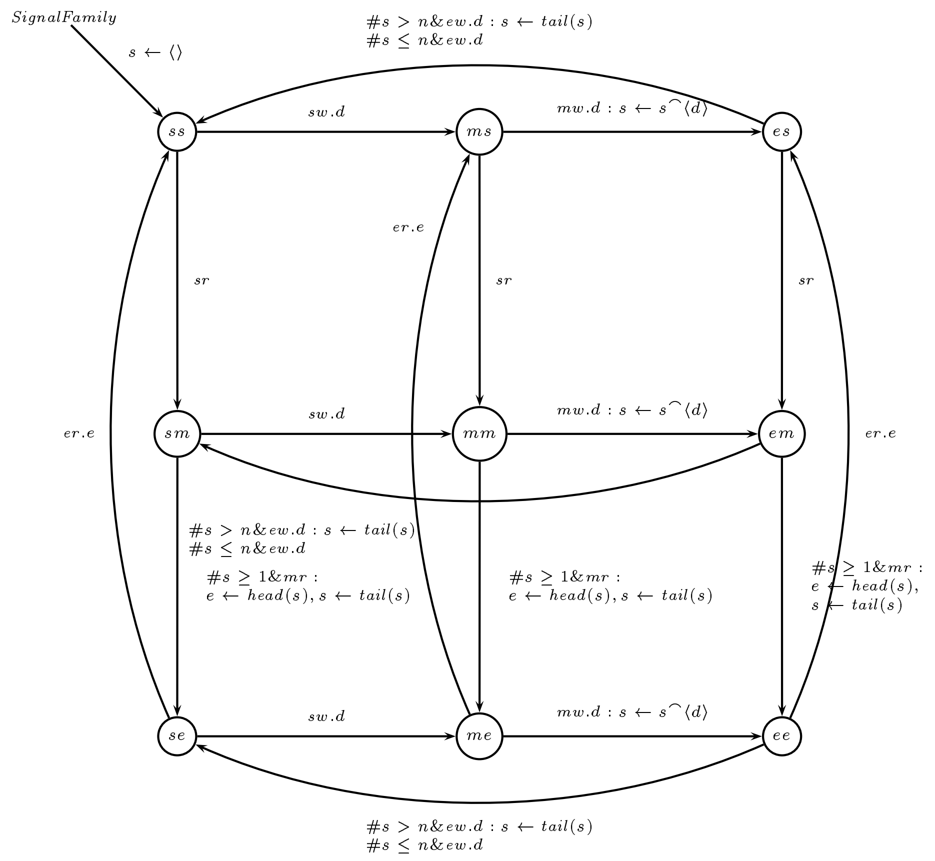

$ChannelFamily$
$s \leftarrow \langle\rangle$
$\#s \leq n \& ew.d$
$sw.d$
$mw.d : s \leftarrow s ^\frown \langle d \rangle$
ss
ms
es
$s = \langle e \rangle ^\frown s' \& er.e : s \leftarrow s'$
sr
sr
sr
$s = \langle e \rangle ^\frown s' \&$
$er.e : s \leftarrow s'$
sm
$sw.d$
mm
$mw.d : s \leftarrow s ^\frown \langle d \rangle$
em
$s = \langle e \rangle ^\frown s' \&$
$er.e : s \leftarrow s'$
$\#s \leq n \& ew.d$
$\#s \geq 1 \& mr$
$\#s \geq 1 \& mr$
$\#s \geq 1 \& mr$
se
$sw.d$
me
$mw.d : s \leftarrow s ^\frown \langle d \rangle$
ee
$\#s \leq n \& ew.d$

Figure D.3: State machine for *ChannelFamily*



$SignalFamily$
$s \leftarrow \langle\rangle$
$\#s > n \& ew.d : s \leftarrow tail(s)$
$\#s \leq n \& ew.d$
ss
$sw.d$
ms
$mw.d : s \leftarrow s ^\frown \langle d \rangle$
es
$er.e$
sr
sr
sr
$er.e$
sm
$sw.d$
mm
$mw.d : s \leftarrow s ^\frown \langle d \rangle$
em
$er.e$
$\#s > n \& ew.d : s \leftarrow tail(s)$
$\#s \leq n \& ew.d$
$\#s \geq 1 \& mr :$
$e \leftarrow head(s), s \leftarrow tail(s)$
$\#s \geq 1 \& mr :$
$e \leftarrow head(s), s \leftarrow tail(s)$
$\#s \geq 1 \& mr :$
$e \leftarrow head(s),$
$s \leftarrow tail(s)$
se
$sw.d$
me
$mw.d : s \leftarrow s ^\frown \langle d \rangle$
ee
$\#s > n \& ew.d : s \leftarrow tail(s)$
$\#s \leq n \& ew.d$

Figure D.4: State machine for *SignalFamily*

Glossary

A

access interface The generalized *interface* (not just one reader-one writer) for a *MASCOT*-3 *IDA* (Page 79).

activity A (simple or complex) *DORIS* entity that carries out a task (Page 111).

ADL Activity Description Language [83] (Page 113).

atomic A Lamport variable that appears to have atomic interactions with its *environment* [59] (Page 81).

B

BAe British Aerospace.

BNF Backus-Naur Form; used in Chapter 3 (Page 62).

C

CASE Computer Aided Software Engineering.

CCS Milner's Calculus of Communicating Systems [73, 74] (Page 43).

channel A basic *route*; both the writer and the reader may be held up (Page 95).

code stub Code written in either *Timed CSP*, or a language translatable to *Timed CSP* (*e.g.* the language presented in Chapter 5) and used to represent the actions of a *server* or *activity* (Pages 216 and 119).

complex activity An *activity* that can be decomposed into further *DORIS* entities (Page 61).

complex path A multiplexed *path* that 'carries' many simple *paths* (Page 217).

complex port A *port* that connects to either *complex paths*, or many simple *paths* (Page 61).

complex server A *server* that can be decomposed into further *DORIS* entities (Page 57).

complex window A *window* that connects to either *complex paths*, or many simple *paths* (Page 61).

constant A basic *route*; the writer is always held up, but the reader can always read (Page 92).

CORE Controlled Requirements Expression.

critical A piece of software of which the failure would have 'serious' repercussions.

CSP Hoare's Communicating Sequential Processes [46] (Section 2.4, Page 33).

D

device A collection of *IDAs* (Page 57).

DIA Data Interaction Architecture (Page 55).

direct link (*i.e.* direct *transputer* link) A (direct) link between two *transputers*.

divergence An observation of a *CSP process* where an unbounded amount of internal work occurs (Page 38).

DORIS Simpson's Data Orientated Requirements Implementation Scheme (Page 53).

dt 'DORIS-to-Timed CSP': the bespoke tool described in this thesis (Page 155).

E

embedded A dedicated computer system, typically built into a larger system (*e.g.* the engine controller of an aircraft) [12].

environment A conceptual *process* with which *CSP processes* are placed in parallel composition.

event The basic element of observation in *CSP* (Page 34).

F

failure An observation of a *CSP process* consisting of a *trace* and a *refusal set* (Page 37).

FDR Failures-Divergence Refinement [28]; a model checker for *CSP* (Page 47).

FIFO First in-first out.

FM See Page 28.

formal methods See Page 28.

forwarding process A *process* that 'forwards' *IDA* interactions across *transputer* links (Page 103).

four-slot An implementation of the *pool IDA* (Page106).

FQUN Fully-qualified unique name (Page 73).

H

hierarchical The property of a notation where some constructs can be decomposed hierarchically: they can be reduced into other constructs which may themselves be decomposed.

I

IDA Intercommunication Data Area. A *protocol* or *route*; generally interchangeable with *protocol* and *route*.

indirect link (*i.e.* indirect *transputer link*) A link between two indirectly-connected *transputers*, via a third *transputer* (which is itself directly linked to the two *transputers*).

interface An *access interface*, *window*, or *port*, depending on context.

internal event An *event* that is hidden from external observers by the *CSP* hiding operator '\' (Page 36).

M

MADGE MASCOT-3 Design Generator [65, 66] (Page 48).

MASCOT Modular Approach to Software Construction, Operation, and Test (Page 30).

mutex Mutual exclusion; only one of many activities competing for a single resource may use it at any single time.

O

observation A *CSP process* interacts with its *environment*: an observation is the entire history of these interactions for a particular execution. The *CSP* operators can be defined in terms of these observations. The nature of the observation is dependent on the semantic model being used, *e.g.* the observations of a *process* in the *UT* model are *traces*.

OO Object oriented (a programming paradigm) [54].

P

path A connection between specific *DORIS* entities that represents data flow (Page 61).

pool A basic *route*; neither the reader or the writer are held up.

port An *interface* onto an *activity* or *server*.

pragmatics The usability of a language (Section 2.5).

private IDA An *IDA* which is written to, and read from on the same *processor* (Page 80).

process An element of a *CSP semantic* domain (Section 2.4.1).

processor A CPU.

program A term of the *CSP syntax* (Section 2.4.1).

proof script A file of instructions that can be replayed by a theorem prover (*e.g. PVS*) to prove a lemma.

protocol Strictly, the specification of an *IDA*.

PVS 'Prototype verification system'; a theorem prover [78] (Page 48).

R

real-time A system of which its correctness depends not only on the correctness of a result but also on the time that the result is produced. This can be in the dense ($\mathbb{R}$) time domain, or the discrete ($\mathbb{Z}$) time domain. In this thesis, 'real-time' generally refers to observations in the $\mathbb{R}$ domain.

refusal set A set of *events* that a *CSP process* is not prepared to engage in at that observation (Page 37).

regular A Lamport variable that is a realistic compromise between *safe* and *atomic* variables [59] (Page 81).

remote IDA An *IDA* which is read on one *processor*, and written to via an *activity* on another *processor* linked to the first *processor* via a *indirect link* (Page 80).

resolved path A simple *path* representing a sequence of paths connecting a port to a window via other ports and windows (Section 3.8.1).

route Strictly, an implementation of a *protocol*. Generally, interchangeable with *IDA* and *protocol*.

RTL Real-Time Logic [53].

S

safe A Lamport variable that returns any type-correct value when a write conflicts with a read [59] (Page 81).

safety critical A system of which failure would cause injury or loss of life.

scheduler Software that allocates resources of concurrent programs on the same *processor* (Section 5.6).

semantics The assignment of meanings to legal terms of a language (Section 2.5).

server A specialised *activity* that interacts with a piece of hardware (Page 79).

shared IDA An *IDA* which is read on one *processor*, and written to via an *activity* on a *processor* directly linked to the first *processor* (Page 80).

signal A basic *route*; the writer is never held up, but the reader may be (Page 96).

stability An alternative treatment of *divergence* (Page 38).

subsystem A distinct part of a *system* when it is decomposed.

syntax The appearance and structure of legal terms of a language (Section 2.5).

system The final product of a software design.

T

tag A two-character string denoting the 'type' of a DORIS component in the formal semantics (Section 3.3.1).

TCSP See *Timed CSP*.

TCC See *type-correctness condition*.

testing The method where an input is given to a system, and the output checked against the desired value. This is (ideally) repeated for every possible system input.

TF The timed-failures model of *Timed CSP*.

theory A *PVS* object which can be type-checked, may contain definitions, lemmas, and may import other theories.

three-point Modelling interactions with three distinct events: start, 'middle', and end (Section 4.5).

Timed CSP An extension of *CSP* involving real-time [102] (Section 2.4.3).

timed event The *CSP* observation consisting of a time value, and an *event*.

timed failure The *CSP* observation consisting of a *timed trace*, and *timed refusal*.

timed refusal A set of *timed events* representing those events that a *process* was not prepared to engage in at particular times.

timed trace A sequence of *timed events* representing the execution of a *process*.

TLA Lamport's Temporal Logic of Actions (Page 44).

T-link A *Transputer* link.

trace A sequence of *events* representing the execution of a *process*.

transputer Specifically, a specialised microprocessor manufactured by INMOS, which communicates with other transputers on a network via interprocessor links. Used in this thesis to refer to either a transputer (in the large case study) or the syntactic element $\langle Transputer \rangle$ in the distributed-level of the DIA.

TT The timed-traces model of *Timed CSP*.

two-point Modelling interactions with two distinct events: a start and an end (Section 4.3).

type-correctness condition A proof obligation generated by *PVS* to ensure type-correctness, consistency, and termination (Page 144).

U

UF The untimed-failures model of *Timed CSP*.

UT The untimed-traces model of *Timed CSP*.

W

window An *interface* onto an *IDA*.

Glossary of Symbols

Timed CSP Programs

(see Sections 2.4.2 and 2.4.3, Pages 35 and 38 respectively.)

$Stop$ The deadlock process.

$Skip$ The successful process.

$Wait$ Delay.

$Chaos$ The most nondeterministic process.

Run The process which will engage in any event.

$\|$ Parallel composition.

$|||$ Asynchronous parallel.

$\|_A$ Alphabetized parallel.

$\|^i_{A_i}$ Network parallel.

$\triangleright$ Timeout.

$\swarrow$ Timed Interrupt.

$\rightarrow$ Prefix.

$\backslash$ Hiding.

$\Box$ External (general) choice.

$\sqcap$ Internal (nondeterministic) choice.

μ Recursion.

$;$ Sequential composition.

Notation for Traces and Refusal Sets

tr A trace.

rf A refusal set.

τ A timed trace.

$\aleph$ A timed refusal set.

$\sqsubseteq$ Refined by....

$Traces$ Traces of a program.

sat Satisfaction.

$\#$ Event count.

Σ The universal set of events.

$\checkmark$ The termination (success) event.

$\frown$ Concatenation of traces.

$\downarrow$ Restriction of a (timed or untimed) trace or refusal set to an alphabet.

$\uparrow$ Restriction of a timed trace or refusal set to a time interval.

σ Set of events in a (timed) trace.

seq Set of sequences of a given set of events.

$\preceq$ Subtrace.

$\ll$ Prefix of a trace.

$\preccurlyeq$ Information order.

$first$ First event in a (timed) trace.

$last$ Last event in a (timed) trace.

$begin$ First time in a timed trace.

end Last time in a timed trace.

$head$ First (timed) event in a (timed) trace.

$foot$ Last (timed) event in a (timed) trace.

$\mathbb{R}$ The set of real numbers.

$\mathbb{Z}$ The set of integers.

δ Krönecker delta[1].

BNF Notation

$\langle \ldots \rangle$ Syntactic terms.

$\{\ldots\}$ Zero or more terms.

$\{\ldots\}^+$ One or more terms.

$[\ldots]$ Optional term.

$|$ Choice.

$(\ldots)$ Grouping.

"…" Literal.

⟨newline⟩ One or more newlines.

DORIS Semantics

$[\![\ldots]\!]$ Semantic brackets.

$\mathcal{S}$ Top-level tuple.

[1]The Krönecker delta is defined as

$$\delta(e) = \begin{cases} 1 & \text{if } e \text{ is true} \\ 0 & \text{otherwise} \end{cases}$$

$\mathcal{SY}$ System construct.

$\mathcal{CS}$ Complex server construct.

$\mathcal{CA}$ Complex activity construct.

$\mathcal{DV}$ Device construct.

$\mathcal{M}$ Meaning function of a system.

$\mathcal{C}$ Meaning function of a complex component.

$\mathcal{H}$ Hiding alphabet for a complex component.

$\mathcal{B}$ CSP meaning for a basic component.

$\mathcal{A}$ CSP meaning of an activity code stub.

$\mathcal{NCS}$ CSP meaning for a complex server of a given name.

$\mathcal{NCA}$ CSP meaning for a complex activity of a given name.

$\mathcal{NDV}$ CSP meaning for a device of a given name.

$\mathcal{NSS}$ Code stub for a simple server of given $\langle Name \rangle$.

$\mathcal{NAS}$ Code stub for a simple activity of given $\langle Name \rangle$.

$\mathcal{NRS}$ Code stub for a route of given $\langle Name \rangle$ and $\langle Route\_Type \rangle$.

$\mathcal{F}$ Fully qualified unique name of a simple server, simple activity, or route.

$\mathcal{R}$ The route and port or window associated with a server or activity's port or window.

$Scheduler$ A scheduler process.

$tock$ Clock ticking.

Bibliography

[1] Martín Abadi and Leslie Lamport. Conjoining specifications. Technical Report 118, DEC SRC, December 1993.

The specification of components of a system using TLA [60] is illustrated. In this report, the specification of the system is the conjunction of the specifications of the components.

The differences between decomposing a complete system, and composing open parts of a system together are discussed. (A complete system is one which is totally self-contained; it may be observed, but does not interact with the observer. Open systems interact with their environment.)

This paper is very detailed, and is specific to Lamport's TLA.

[2] James Armstrong and Leonor Barroca. Specification and verification of reactive system behaviour: The railroad crossing example. *Real-Time Systems*, 10(2):143–178, 1996.

This paper combines Timed Statecharts, RTL, and Proofpower HOL. Specification and verification of reactive system behaviour is described in a human-readable rigorous style, and supplemented by automated (formal) proofs.

[3] Luciano Baresi and Maruo Pezzè. Toward formalizing structured analysis. *ACM Transactions on Software Engineering and Methodology*, 7(1):80–107, Jan 1998.

This paper asserts that the advantages of structured analysis (SA) are limited by the informality of the notations used, leading to ambiguous specifications. The assertion is illustrated with a list of such features from common notations.

[4] Jon Barwise. Mathematical proofs of computer system correctness. *Notices of the American Mathematical Society*, 36:844–851, 1989.

This paper summarizes the main arguments concerning the concept of program verification. In particular, it starts with the assertions in Fetzer's paper [29], which claims that the aims of program verification are, in principle, quite impossible.

Although the author disagrees with Fetzer's claims, (and indeed states 'I think that program verification *is* an effective way of getting more reliable programs') he sounds a note of caution concerning the modelling of computer systems.

[5] Twan Basten and Jozef Hooman. Process algebra in PVS. Technical Report 98/10, Eindhoven University of Technology, 1998.

This paper explores two approaches to using PVS to support an ACP-like process algebra (one using 'rewrite' definitions; the other using datatypes and equivalence relations). The authors conclude that such use is feasible, and identify work required (*e.g.* defining the underlying theories of the process algebra).

[6] Michael von der Beeck. A comparison of Statecharts variants. In H. Langmaack, W.P. de Roever, and J. Vytopil, editors, *Proceedings of the 3rd International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, pages 128–148. Springer-Verlag, 1994.

This paper describes briefly the problems of the Statecharts formalism [40]. It also surveys 21 variants in 13 papers [14, 24, 39, 40, 49–51, 57, 64, 68, 69, 85, 90], and notes the differences between them. This is useful as a reference paper.

[7] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL: A tool suite for automatic verification of real-time systems. In *4th DIMACS Workshop on Verification and Control of Hybrid Systems*, October 1995.

The authors present their tool, UPPAAL, which takes timed automata and can perform safety and (bounded) liveness checks. It has an X Windows interface, and is able to cope with relatively large systems by using constraint-solving methods.

[8] Jonathan P. Bowen and Michael G. Hinchey. Seven more myths of formal methods. *IEEE Software*, 12(4):34–41, July 1995.

Another seven myths to supplement Hall's original seven [36]. The seven myths in this paper are:

1. Formal Methods delay the development process.
2. Formal Methods are not supported by tools.
3. Formal Methods mean forsaking traditional engineering design methods.
4. Formal Methods only apply to software.
5. Formal Methods are not required.
6. Formal Methods are not supported.
7. Formal Methods people always use Formal Methods.

[9] P.J. Brooke, J.L. Jacob, and J.M. Armstrong. An analysis of the four-slot mechanism. In *Proceedings of the BCS-FACS Northern Formal Methods Workshop*, electronic Workshops in Computing. Springer-Verlag, 1996.

This paper models Simpson's four-slot mechanism in CSP [46, 108, 109]. This four-slot mechanism is an implementation of the DORIS pool [110].

It is noted that even in this small model without real time, the analysis of the mechanism is difficult due to the size of the state space.

[10] Phillip J. Brooke. Literature survey on hierarchical timed transition systems for high-integrity real-time systems, 1996. First Year MPhil/DPhil Qualifying Dissertation, Department of Computer Science, The University of York; also issued as BAe report DCSC/BG/96/6.

This literature survey covers the semantics of graphical design notations such as Statecharts [39] in the context of critical systems. Process algebras are examined as a semantic domain for graphical notations.

[11] Jeremy Bryans and Steve Schneider. CSP, PVS and a recursive authentication protocol. DIMACS Workshop on Formal Verification of Security Protocols, September 1997.

This paper describes an authentication protocol, and the analysis of that protocol using CSP. The proofs are mechanized using PVS.

[12] Jean Paul Calvez. *Embedded Real-Time Systems*. Wiley, 1993.

This book introduces the Electronic Systems Design Methodology (MCSE — from its original French name). As the title suggests, this is aimed at embedded real-time systems.

[13] Albert John Camilleri. Mechanizing CSP in HOL. *IEEE Transactions on Software Engineering*, 16(9):993–1004, 1990.

This is an optimistic paper concerning the embedding of (traces) CSP in the HOL theorem prover. The approach is similar to later papers on embedding CSP in PVS, except that the structure of the programs is used as a first-class datatype.

Towards the end of the paper, the seperate definition of the syntax of CSP permits an easy definition of failures and divergences semantics. Camilleri comments that the increasing power of theorem provers should enable reasoning about more complex semantics.

[14] A. Classen. Modulare Statecharts: Ein formaler rahmen zur hierarchischen prozeßspezifikation. Master's thesis, Aachen University of Technology, 1993.

Referred to in von der Beeck's paper [6].

[15] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and Madnayam Srivas. A tutorial introduction to PVS. Presented at WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques, April 1995.

This is an introduction, tutorial, and reference to the PVS theorem prover.

[16] P. Curwen, S. Mallon, and T. Smith. The official handbook of the CORE method. Technical Report DCSC/BG/91/25, DCSC, September 1991.

'CORE' is the 'Controlled Requirements Expression' method.

[17] Defence standard 00-55. MoD CIS Procurement Board, July 1995. Draft revision (version 6).

This document sets out the requirements (and related guidance) for Safety Related Software used in Defence Equipment in the UK armed forces. This standard has been informally adopted in other domains, *e.g.* railways and civil aviation.

[18] J. Davies, D.M. Jackson, G.M. Reed, J.N. Reed, A.W. Roscoe, and S.A. Schneider. Timed CSP: Theory and practise. In *Real-Time: Theory in Practise*, volume 600 of *Lecture Notes in Computer Science*, pages 640–675. Springer-Verlag, 1992.

This paper is an overview of work on Timed CSP performed at Oxford University.

[19] Jim Davies and Steve Schneider. A brief history of Timed CSP. *Theoretical Computer Science*, 138:243–271, 1995.

This is an updated version of the previous paper of the same title [20].

[20] Jim Davies and Steve Schneider. A brief history of Timed CSP. Technical Report PRG-96, Programming Research Group University of Oxford, April 1992. Also available by FTP from ftp.comlab.ox.ac.uk.

This paper is 'a comprehensive introduction to the language of Timed CSP'.

[21] Jim Davies and Steve Schneider. Real-time CSP. Available by FTP from ftp.comlab.ox.ac.uk.

A real-time variant of CSP is described in detail. It is illustrated with two case studies: a distributed watchdog timer, and the railroad crossing example.

[22] Jim Davies and Steve Schneider. Using CSP to verify a timed protocol over a fair medium. Available by FTP from ftp.comlab.ox.ac.uk.

A timed failures model of CSP that permits infinite observations is introduced so that fairness can be adequately treated. The example used in this paper is the alternating bit protocol.

[23] Jim Davies. *Specification and Proof in Real-Time Systems*. DPhil thesis, Oxford University Computing Laboratory Programming Research Group, 1991. PRG-93.

Davies substantially extends Reed and Roscoe's work [92]. The motivation for this is to enable the specification and proof of complex real-time systems. This thesis is an excellent presentation of the language of TCSP.

[24] N. Day. A model-checker for Statecharts: Linking CASE tools with formal methods. Technical Report 93-95, Department of Computer

Science, University of British Columbia, Vancouver, Canada, October 1993.

Of particular interest in this reference is Chapter 2, which covers the semantic issues of Statecharts. This includes a discussion of what actually constitutes a 'step'.

[25] Edsger Wybe Dijkstra and Carel S. Scholten. *Predicate calculus and program semantics*. Springer-Verlag, 1989.

This text is "a reasonable self-contained theory of predicate transformer semantics". Predicate transformers are introduced as a means of defining programming language semantics in a way that would directly support the systematic development of programs from their formal specifications. The notions of 'weakest precondition', 'weakest liberal precondition', and 'strongest postcondition' are introduced. The presentation of proofs is elegant.

[26] Bruno Dutertre and Steve Schneider. Using a PVS embedding of CSP to verify authentication protocols. In *International Conference on Theorem Proving in Higher Order Logics*, 1997.

This is a report of the use of an embedding of the traces model of CSP in PVS. This embedding is subsequently used in the verification of security protocols.

[27] Bruno Dutertre and Steve Schneider. Embedding CSP in PVS: An application to authentication protocols. Technical Report CSD-TR-97-12, Royal Holloway, University of London, 1997.

This is a detailed account of using CSP embedded in PVS as applied to authentication protocols.

[28] Formal Systems (Europe) Ltd. Failures-divergence refinement. http://www.formal.demon.co.uk/, October 1997.

FDR is a model checker that uses a widely-accepted machine-readable CSP [46] syntax, and performs refinement (and other) checks upon models.

This document describes the use of this tool.

[29] James H. Fetzer. Program verification: The very idea. *Communications of the ACM*, 31(9):1048–1063, September 1988.

This paper provoked a long-running argument: it asserted that the whole concept of trying to verify programs was flawed because a real program running on a real computer could not be appropriately modelled.

[30] Simon Fowler. *A Development Method for Trusted Real-Time Kernels*. DPhil thesis, Department of Computer Science, University of York, 1998.

Fowler introduces a PVS model of RTL [53], and uses this to develop a provable trusted real-time kernel.

[31] Eric Goubault and Thomas P. Jensen. Homology of higher dimensional automata. In *CONCUR '92*, volume 630 of *Lecture Notes in Computer Science*, pages 254–268. Springer-Verlag, 1992.

This paper describes the use of geometry, specifically, higher dimensional automata (HDA), for describing concurrency. HDAs are a generalisation of nondeterministic finite automata.

[32] T.R.G. Green and M. Petre. When visual programs are harder to read than textual programs. In G.C. van der Veer, M.J. Tauber, S. Bagnarola, and M. Antavolits, editors, *Human-Computer Interaction: Tasks and Organisation. Proceedings of ECCE6 (6th European Conference on Cognitive Ergonomics)*. CUD, 1992.

This paper starts with the comment that

> "Claims for the virtues of visual programming languages have generally been strong, simple-minded statements that visual programs are inherently better than textual ones."

The paper continues with a comparison between textual and visual programs, and concludes that the visual programs are harder to comprehend (at least in this case).

[33] T.R.G. Green and R. Navarro. Programming plans, imagery and visual programming. In *Proceedings of INTERACT '95*, 1995.

This paper discusses the mental models that programmers using different types of language use (*i.e.* textual languages *vs.* visual languages).

[34] Alan Grigg. End-to-end timing analysis case study: ASRAAM EPU. Technical Report DCSC/TR/97/13, University of York DCSC, December 1997. Version 1.

This is a report of the application of a form of timing analysis to a case study. The case study concerned is based on the same system as the large case study of this thesis (Section 8.5). The analysis is based on end-to-end timing.

[35] Corin A. Gurr. Supporting formal reasoning for safety-critical systems. *High Integrity Systems*, 1(4):385–396, 1995.

This paper combines both Milner's CCS and Statecharts. It also briefly discusses the reasons why formal methods are not widely used. A correspondence from certain CCS constructions to Statecharts is devised, and is used to investigate small fragments of specifications.

[36] J.A. Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, September 1990.

This paper describes seven 'myths' of formal methods, and explains why they do not hold. The myths are:

1. Formal Methods can guarantee that software is perfect.
2. Formal Methods are all about program proving.
3. Formal Methods are only useful for safety-critical systems.
4. Formal Methods require highly trained mathematicians.
5. Formal Methods increase the cost of development.
6. Formal Methods are unacceptable to users.
7. Formal Methods are not used on real, large-scale software.

[37] Drew Hamilton. Programming languages do make a difference. Speech at Tri-Ada '97, November 1997.

This speech covered topics primarily relating to the Ada audience, but also touched upon the realm of safety critical software, and how it should be designed and implemented.

[38] Michael R. Hansen and Zhou Chaochen. Semantics and completeness of duration calculus. In *Real-Time: Theory in Practise*, volume 600 of *Lecture Notes in Computer Science*, pages 209–225. Springer-Verlag, 1992.

The Duration Calculus (an extension of the Interval Temporal Logic) is described. A formal syntax and denotational semantics is then given. The

Duration Calculus has been used to give a semantics to a real-time variant of Hoare's CSP [46].

[39] D. Harel, A. Pnueli, J. Schmidt, and R. Sherman. On the formal semantics of Statecharts. In *Proceedings of 2nd IEEE Symposiom on Logic in Computer Science*, pages 54–64, 1987. Extended abstract.

This paper suggests a syntax and operational semantics for Statecharts [40]. There are formal treatments of the syntax and semantics in an appendix. Of particular interest is the definition of a 'step' as a sequence of micro-steps, and the distinction between external and internal events.

This is one of many attempts at defining the semantics of Statecharts, and it exhibits the problem of generating the negation of the triggering event on a transition.

[40] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

This paper by Harel is a full description of the Statechart formalism, as it was in 1987.

[41] E. C. Hehner and C. A. R. Hoare. A more complete model of communicating processes. *Theoretical Computer Science*, 26:105–120, 1983.

Hoare's previous work on CSP is supplemented with predicate definitions. These are consistent with the earlier axioms and proof rules, and are more powerful to reason with.

[42] Matthew Hennessy. *The Semantics of Programming Languages: An Elementary Introduction using Structural Operational Semantics*. Wiley, 1990.

Hennessy's exposition is clear and understandable. It covers syntax and basic mathematical concepts, and then surveys several semantic theories, which he calls: concrete operational semantics, evaluation semantics, computation semantics, and denotational semantics. The book has a large number of examples, although some terminology differs from other texts.

[43] M. Hennessy. Concurrent testing of processes. In *CONCUR '92*, volume 630 of *Lecture Notes in Computer Science*, pages 94–107. Springer-Verlag, 1992. (extended abstract).

Hennessy develops a non-interleaving semantics based on actions with a strictly positive duration in time. The language is based on CCS [74], and describes a notion of testing known as 'Characteristic ST-testing'.

[44] Martin Hesketh. Synthesis of Petri box expressions from Petri boxes. University of Newcastle upon Tyne.

This paper covers the problem of translating (synthesising) from Petri Boxes (labelled Petri nets) to Box expressions (a semantic domain). Hesketh proposes an algorithm for this problem.

[45] C.A.R. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall, 1998.

This book demonstrates that different specification and proof methods can be used for different parts of a design project.

[46] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International UK, 1985.

This classic text is a description of Hoare's Communicating Sequential Processes (CSP). CSP is a process algebra which is useful for the study of concurrency and communication. A system evolves by interacting with its environment via events. Concurrent components are able to synchronise on these events.

The language of CSP is very rich, enabling the modelling of diverse systems. A proof theory is provided, and there a several semantic models (traces, failures, and divergences).

[47] C.A.R. Hoare. Algebraic specifications and proofs for communicating sequential processes. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, volume F36 of *NATO ASI Series*. Springer-Verlag, 1987.

In this paper, Hoare defines processes by their algebras, similar to the definition of the natural numbers using the Peano axioms.

[48] C. Michael Holloway and Ricky W. Butler. Impediments to industrial use of formal methods. *IEEE Computer*, 29(4):25–26, April 1996.

This short article (one of a series on formal methods in this journal issue) discusses why formal methods have not been used greatly in industry. The authors identify three major impediments:

1. Inadequate tool support.
2. Inadequate examples.
3. The gulf between industry and research.

[49] J.J.M. Hooman, S. Ramesh, and W.P. de Roever. A compositional axiomatization of Statecharts. *Theoretical Computer Science*, 101(2):289–335, 1992.

This paper describes a compositional axiomatisation for Statecharts. It is large, and covers most areas of Statecharts. The authors state: 'Although the meaning of a statechart is usually intuitively clear, there are a few cases where the specified behaviour is not completely obvious.'

[50] C. Huizing, R. Gerth, and W.P. de Roever. Modelling Statecharts behaviour in a fully abstract way. In *CAAP '88*, volume 299 of *Lecture Notes in Computer Science*, pages 271–294. Springer-Verlag, 1988.

This paper describes a syntax for a restricted version of Statecharts [40]. A denotational semantics in terms of sets of 'history triples' (each corresponding to a possible execution of a Statechart) is given. The authors claim that this semantics is compositional.

[51] C. Huizing and R. Gerth. Semantics of reactive systems in abstract time. In *Real-Time: Theory in Practise*, volume 600 of *Lecture Notes in Computer Science*, pages 291–314. Springer-Verlag, 1992.

This paper describes three criteria for real-time reactive systems: responsiveness, modularity, and causality. The difference between transformational and reactive systems is described: the former read their input; produce a (possibly nondeterministic) output, then terminate. The latter maintain an interaction with the environment.

In particular, the authors note that several attempts have been made to formalise the development of reactive systems (e.g. Statecharts), and that there have been problems. It is asserted that the three criteria cannot be combined in one semantics. The three criteria are described are:

Responsiveness 'Meaning that a system's output comes simultaneously with the input that causes it'. This is essentially the synchrony hypothesis [90].

Modularity 'All parts of the system should be treated symmetrically. . . every part of the system should have the same view of the events occurring in the total system at any moment'. This leads to the notion of communication by immediate broadcast.

Causality There must be some input event preceding any 'internal' event.

A simple reactive language is then described, and five semantics proposed. None of them satisfy all three criteria. The proposed solution is that the criteria are satisfied at different 'levels' of view.

[52] Michael Jackson. *Software Requirements and Specifications*. Addison-Wesley, 1995.

This book contains 75 short articles concerning the principles and techniques of software engineering. The comments on the roles of mathematics and formalism are particularly interesting.

[53] Farnam Jahanian and Aloysius K. Mok. Safety analysis of timing properties on real-time systems. *IEEE Transactions on Software Engineering*, 12(9):890–904, September 1986.

This paper introduces Real-Time Logic, a formalism which is intended to be 'especially amenable to reasoning about the timing behaviour of systems'.

[54] Pankaj Jalote. *An Integrated Approach to Software Engineering*. Springer, 2nd edition, 1997.

This book is an introduction to software engineering, and is aimed at undergraduates. Its emphasis is on a case-study approach in which a project is developed through the course of the book. All typical software activities, including quality assurance and control, are described. The author introduces metrics for controlling and assessing the software process.

[55] C. B. Jones. Accomodating interference in the formal design of concurrent object-based programs. *Formal Methods in System Design*, 8:105–122, 1996.

Ideally, complex concurrent systems would be composed of separate components. However, they tend to interfere with each other to the detriment of the design. Jones describes two approaches: minimizing the interference by isolating components; and living with the problem by documenting and proving the design.

[56] Andrew Kay. A theory of rely and guarantee for timed CSP. Technical report, Oxford University Computing Laboratory Programming Research Group, November 1993.

This paper introduces the notion of CSP programs satisfying a specification of the form

$$P\ \mathbf{sat}\ (Reply_P \Rightarrow Guarantee_P)$$

This means that a weaker specification can be given, which is easier to implement, provided that the environment satisfies $Reply_P$.

[57] Y. Kesten and A. Pnueli. Timed and hybrid statecharts and their textual representation. In *Lecture Notes in Computer Science*, volume 571, pages 591–620. Springer-Verlag, 1991.

This paper defines a textual representation for Statecharts. Transition relation rules are defined with diagrams.

[58] Ralf Kneuper. Limits of formal methods. *BCS Formal Aspects of Computing*, 9(4):379–394, 1997.

Kneuper gives a realistic discussion about how formal methods can be realistically applied in the 'real world'. In particular, he highlights the problem with scaling-up small-grain mathematical models of systems without encapsulation.

[59] Leslie Lamport. On interprocess communication. Technical Report 8, DEC SRC, 1985.

Lamport describes a formalism which is not based on atomic actions for specifying and reasoning about concurrent systems. System execution is considered to consist of a set of operation executions (which are not assumed to be atomic) and certain temporal precedence relations on the operation executions. A 'precedes' B is taken to mean that all actions of A occur before any actions of B; A 'can affect' B means that some action of A precedes some action of B.

Communication is divided into 'transient' and 'persistent' types. It is claimed that communication between truly asynchronous processes must be persistent. Requiring mutual exclusion for some operations will, at some lower level, require the problem of concurrent access to be addressed. Three types of register which permit concurrent reads and writes are defined: safe (will get a type-valid value), regular (will get the previous or the new value), and atomic (reads and writes appear to occur in a definite order). Lamport then describes the construction of stronger registers from weaker registers.

[60] Leslie Lamport. The temporal logic of actions. Technical Report 79, DEC SRC, December 1991.

This paper presents the temporal logic of actions (TLA). It is a small logic, but powerful, as Lamport illustrates. Specifications and systems are described in the same language, thus it is easier to reason about satisfaction (and also refinement).

[61] Leslie Lamport. TLA in pictures. Technical Report 127, DEC SRC, September 1994.

Lamport describes predicate-action diagrams, which are interpreted in terms of TLA [60]. The diagrams are closer to TLA than they are to traditional state-transition diagrams.

[62] Leslie Lamport. Proving possibility properties. Technical Report 137, DEC SRC, December 1995.

This short report discusses the use of TLA [60] for proving properties in a traces model.

[63] J. van Leeuwen. *Handbook of Theoretical Computer Science*, volume B: Formal Methods and Semantics. MIT Press/Elsevier, 1990.

A large volume, covering topics such as automata, rewriting, functional programming, the lambda calculus, type systems, semantics, temporal and modal logics, and distributed and concurrent issues. An excellent reference text.

[64] Nancy G. Leveson, Mats Per Erik Heimdahl, Holly Hildreth, and Jon Damon Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, 20(9):684–707, September 1994. Also Technical Report 92-106 (University of California).

This paper uses the TCAS II (aircraft collision avoidance system) as a case study. It notes that formal requirements need to be described in a way understandable by applications specialists (i.e. not formal methods specialists). The approach given is to model the 'black-box' behaviour of the system, the interface, and no more.

The paper includes the following design criteria for such a language:

- blackbox
- minimal
- semantically simple
- coherent, consistent, and concise
- unambiguous underlying language with a formal foundation for analysis
- readable, reviewable, and usable by application experts and developers
- flexible notations (graphical, tabular, and symbolic) tied to the best way to provide a particular type of information
- readability given priority over writeability
- user needs given priority over personal preferences
- information exposure

The Requirements State Machine Language (RSML) is described. The authors note that certain parts of Statecharts were left out, e.g. history and event selectors. Several features that are used well include AND/OR tables in DNF (disjunctive normal form), and transition buses (another way of reducing the number of transition arrows on a diagram). A comparison of the step semantics of RSML and Statecharts is given.

[65] Matra BAe Dynamics. *User Guide for the MASCOT 3 Design Generator (MADGE)*, October 1994. Issue 7.

MADGE is a tool which allows software designs to be captured graphically. MADGE supports the MASCOT-3 method, with extensions drawn from DORIS (primarily the routes which are more developed in DORIS than MASCOT).

In addition to the X-Windows interface and MASCOT design features, the tool provides version control and file management facilities, and provision for code generation, including the generation of SPARK Ada code.

[66] Matra BAe Dynamics. *MASCOT Design Generator (MADGE) Release Notification*, October 1997. Issue 9.

This document updates previous MADGE manuals [65] to correspond with the current version of the MADGE tool.

[67] Joint IECCA and MUF Committee on MASCOT (JIMCOM). *The Official Handbook of MASCOT*, June 1987. Version 3.1, Issue 1, Crown Copyright.

This document sets out the MASCOT-3 method. This is aimed at building concurrent, real-time systems in a structured, hierarchical manner. A design notation is given, as well as comments about development facilities, run-time features, and an overall 'method'. (MASCOT is the 'Modular Approach to Software Construction, Operation, and Test'.)

[68] A. Maggioli-Schettini and A. Peron. Semantics of full Statecharts based on graph rewriting. In *5th International Workshop on Graph Grammars and their Application to Computer Science*, 1994.

Referred to in von der Beeck's paper [6].

[69] F. Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In *CONCUR '92*, volume 630 of *Lecture Notes in Computer Science*, pages 550–564. Springer-Verlag, 1992.

This paper describes the Argos language, which is based on the state-transition paradigm. Level-crossing transitions are suppressed, in contrast to Statecharts [40]. There is a modular decomposition through the hierarchy. The operational semantics are compositional, and refinement of specifications is possible.

[70] John A. McDermid. *Software Engineer's Reference Book*. Butterworth Heinemann, 1991.

This handbook is a collection of specialist articles covering the theory and practise of software engineering.

[71] Nicholas A. Merriam and Michael D. Harrison. Evaluating the interfaces of three theorem proving assistants. In *Proceedings of* DSV-IS*'96*, 1996.

This paper explores how users interact with a theorem prover. Concepts such as planning and reuse in the context of proofs are discussed.

[72] George Milne. *Formal Specification and Verification of Digital Systems*. McGraw-Hill, 1994.

This book describes several complementary approaches to formal specification and verification. It is applied to hardware design, and uses VHDL, HOL, and Circal (a process algebra for circuit design).

[73] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

This is Milner's original exposition of CCS. See his 1989 book [74] for further details.

[74] Robin Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.

The Calculus of Communicating Systems (CCS) is a classic mathematical model of concurrency and communication.

[75] John C. Kelly. *Formal Methods Specification and Verification Guidebook for Software and Computer Systems, Volume I: Planning and Technology Insertion*. NASA, July 1995. Available via HTTP: http://eis.jpl.nasa.gov/quality/Formal_Methods/.

This guidebook is an excellent general introduction to Formal Methods, and includes a comprehensive list of associated tools.

[76] John C. Kelly. *Formal Methods Specification and Analysis Guidebook for the Verification of Software and Computer Systems, Volume II: A Practitioner's Companion*. NASA, May 1997. Available via HTTP: http://eis.jpl.nasa.gov/quality/Formal_Methods/.

This volume gives a more mathematically-inclined introduction to Formal Methods. Chapters on formal specification and formal analysis include comprehensive introductions to a wide-range of topics.

[77] Peter G. Neumann. *Computer-Related Risks*. Addison-Wesley, 1995.

Neumann gives a technical account of many computer mishaps, and importantly, explains how these mishaps could have been avoided. This book addresses similar material to Wiener's [119], but with more technical analysis.

[78] S. Owre, N. Shankar, and J.M. Rushby. *The PVS Specification Language*. Computer Science Laboratory, SRI International, April 1993.

A reference manual for the PVS specification language.

[79] S. Owre, N. Shankar, J.M. Rushby, and D.W.J. Stringer-Calvert. *PVS Language Reference*. Computer Science Laboratory, SRI International, September 1998.

A reference manual for the PVS specification language.

[80] S. Owre, N. Shankar, J.M. Rushby, and D.W.J. Stringer-Calvert. *PVS System Guide*. Computer Science Laboratory, SRI International, September 1998.

A description of the PVS system.

[81] Richard F. Paige. When are methods complementary. Accepted for publication in *Information and Software Technology*, 1999.

This paper discusses *when* (formal) methods integration should be carried out, as opposed to *how* to carry out methods integration. It is a relatively brief note describing some of the issues involved.

[82] Richard F. Paige. *Formal Method Integration via Heterogeneous Notations*. PhD thesis, Graduate Department of Computer Science, University of Toronto, 1997.

This thesis discusses 'method integration' in the domain of formal (and 'semi-formal') methods. The approach is based on 'heterogeneous notations' — compositions of compatible notations. Examples in the areas of system specification, design and implementation are given, and a general 'meta-method' is constructed.

[83] Stephen Paynter, Jim Armstrong, and Jan Haveman. ADL: The activity description language. Draft; intended to be published in BCS Formal Aspects of Computing, October 1997.

The Activity Description Language (ADL) gives a (graphical) notation with a formal RTL semantics that can be used to define the activities in the DORIS method.

[84] Stephen Paynter. *The Formalisation of Software Development Using MASCOT*. PhD thesis, Mathematics Department, Southampton University, September 1993.

This thesis uses graphical grammars to give a CSP-based semantics to a subset of MASCOT-3.

[85] A. Peron. Synchronous and asynchronous models for Statecharts. Technical Report TD-21/93, Dipartimento di Informatica, Universita di Pisa, Italy, 1993.

Referred to in von der Beeck's paper [6].

[86] Carsta Petersohn and Luis Urbina. A timed semantics for the StateMate implementation of Statecharts. In *FME '97: Industrial Applications and Strengthened Foundations for Formal Methods*, volume 1313 of *Lecture Notes in Computer Science*, pages 553–572. Springer-Verlag, 1997.

This paper gives a faithful model of the Statemate variant of the Statechart language.

[87] M. Petre, A.F. Blackwell, and T.R.G. Green. Cognitive questions in software visualisation. In J. Stasko, J. Domingue, B. Price, and M. Brown, editors, *Software Visualisation: Programming as a Multi-Media Experience*. MIT Press, 1997.

Petre *et al.* describes 'Software visualisation', and raises some open questions on the subject, *e.g.*

- What is software visualisation suitable for?
- Does visualisation mean pictures?
- Why are experts often resistant to other people's visualisations?
- Are visualisations trying to provide a representation that is more abstract, or more concrete?
- What do we know about perception, anyway?
- When are two representations better than one?
- Why do people like graphical widgets?

This is a thought-provoking paper.

[88] Marian Petre. Why looking isn't always seeing: Readership skills and graphical programming. *Communications of the ACM*, 38(6):33–44, June 1995.

This article discusses how textual and visual representations differ. It notes that different viewers may see different meanings in a given picture. A number of arguments are referred to, including the phenomena of 'secondary notation' (layout, typographic cues, and so on). It is suggested that the freedom of expression afforded by graphical notations also allows for greater mis-cueing and confusion.

[89] C.A. Petri. Communication with automata. Technical report, Princeton University, 1966. Translation by Clifford F. Greene, Jr. of original 1962 text (*Kommunikation mit Automaten*).

This is the seminal paper on the subject of Petri nets.

[90] A. Pnueli and M. Shalev. What is in a step: On the semantics of statecharts. In *Proceedings of the IEEE Symposium on Theoretical Aspects of Computer Science*, volume 526 of *Lecture Notes in Computer Science*, pages 244–264. Springer-Verlag, 1991.

This paper proposes an operational semantics for Harel's Statecharts [40]. Notions such as the synchrony hypothesis, causality, expressing priorities and global consistency are considered.

An informal description of Statecharts is followed by two definitions of 'step'. This is an improvement on an earlier proposed semantics [39].

[91] Vaughan Pratt. Modelling concurrency with partial orders. Available by FTP from boole.stanford.edu, 1986.

This paper models concurrent systems with an algebra based on partially ordered multisets (pomsets). The author does not give any formal rules for reasoning with these constructs, but an example of a workshop problem shows that for at least some types of problem, the algebra permits elegant expressions. In particular, notions such as continuous time and hierarchy are easily incorporated. This is a very abstract formulation.

[92] G.M. Reed and A.W. Roscoe. A timed model for communicating sequential processes. In *Proceedings of ICALP'86*, volume 226 of *Lecture Notes in Computer Science*, 1986.

Referred to by a number of authors, including Davies [23] and Schneider [100].

[93] G.M. Reed. *A uniform mathematical theory for real-time distributed computing*. DPhil thesis, Oxford University, 1988.

Referred to by a number of authors, including Davies [23] and Schneider [100].

[94] Wolfgang Reisig. *Petri Nets*. Springer-Verlag, 1985.

This is a reasonable introduction to Petri nets [89].

[95] A.W. Roscoe. *The Theory and Practice of Concurrency*. Series in Computer Science. Prentice Hall, 1998.

Roscoe covers untimed CSP in much detail, with many examples, both large and small. This book includes several chapters on different semantic styles, and also discusses mechanization (with FDR). Discrete time (through a $tock$ event) is covered, but not real-time CSP.

[96] J.M. Rushby and D.W.J. Stringer-Calvert. A less elementary tutorial for the PVS specification and verification system. Technical Report CSL-95-10, Computer Science Laboratory, SRI International, August 1996.

This tutorial introduces some of the more powerful features of PVS.

[97] John Rushby. Formal methods and their role in the certification of critical systems. Technical Report CSL-95-1, Computer Science Laboratory, SRI International, March 1995.

This report introduces formal methods, and explains how they can be used to aid in the specification, verification, and ultimately, the certification of (in this case) civil aviation computer systems.

[98] Bryan Scattergood. *Tools for CSP and Timed CSP*. DPhil thesis, Oxford University Computing Laboratory Programming Research Group, 1995. (Draft thesis).

This thesis deals with the issues that must be addressed before useful tools can be developed for Timed CSP. These include formally defining a syntax for the language that can be parsed. This must also include stating how the many mathematical expressions that are often used in blackboard expositions are expressed for the machine.

[99] David A. Schmidt. *Denotational semantics*. Allyn and Bacon, Inc., 1986.

Schmidt starts by stating that a programming language consists of syntax, (the appearance and structure of sentences,) semantics, (the assignment of meanings to the sentences,) and pragmatics, (the usability of the language).

Syntax is briefly addressed in the first chapter, with a description of Backus-Naur form (BNF). The remainder of the book is a thorough description of Strachey's methodology of denotational semantics.

Two other forms specification of semantics are noted: operational semantics (where an interpreter is used to define the language; and axiomatic semantics (where properties about the language constructs are given, expressed in terms of axioms and inference rules).

[100] Steve Schneider. *Correctness and Communication in Real-Time Systems*. DPhil thesis, Oxford University Computing Laboratory Programming Research Group, 1989. PRG-84.

This well-written thesis contains an excellent summary of Timed CSP, as well as describing a number of useful properties in real-time systems.

[101] Steve Schneider. Specification and verification in Timed CSP. In Mathai Joseph, editor, *Real-time Systems: Specification, Verification and Analysis*, International Series in Computer Science, chapter 6, pages 147–181. Prentice-Hall, 1996.

This is probably the most up-to-date version of Timed CSP [19, 20, 46]. The chapter also includes the 'Mine Pump' example, obviously described in Timed CSP.

[102] Steve Schneider. *Concurrent and Real Time Systems: The CSP Approach*. John Wiley, (to appear).

When published, this book will make a significant contribution to the understanding of models of time and concurrency. The book contains a theory of Timed CSP that is more mature than previous descriptions.

[103] Steve Schneider. An operational semantics for timed CSP. Available by FTP from ftp.comlab.ox.ac.uk.

A detailed operational semantics for Timed CSP is given in this paper. Two relations are used: an evolution relation, where a process becomes another by waiting for time to pass, and a timed transition relation, where a process becomes another process by carrying out an action at some time. The relationship of several models of TCSP to testing is discussed.

[104] Steve Schneider. Rigorous specification of real-time systems. Available by FTP from ftp.comlab.ox.ac.uk.

This paper is a fairly short introduction to using TCSP for the rigorous specification of real-time systems.

[105] N. Shankar, S. Owre, J.M. Rushby, and D.W.J. Stringer-Calvert. *PVS Prover Guide*. Computer Science Laboratory, SRI International, September 1998.

A documentation introducing proving with PVS.

[106] Alan C. Shaw. Communicating real-time state machines. *IEEE Transactions on Software Engineering*, 18(9), September 1992.

This paper defines communicating real-time state machines (CRSMs), which are claimed to be 'a new complete and executable notation for specifying concurrent real-time systems'. They are essentially state machines that communicate synchronously in a manner much like the input-output in CSP.

The author provides many examples. A graphical notation, and an operational semantics are given. It is asserted that such specifications are amenable to formal analysis.

[107] Hugo Simpson. The MASCOT method. *Software Engineering Journal*, pages 103–120, May 1986.

The MASCOT method uses functional and structural decomposition, and is considered to be especially suitable for large systems.

[108] Hugo Simpson. Four-slot fully asynchronous communication mechanism. *IEE Proceedings*, 137 Part E(1):17–30, January 1990.

This paper describes a mechanism which enables a reader and a writer to communicate asynchronously. The author claims that this is fully asynchronous, but there are subtleties depending on the assumptions. The terminology used is described at length, as are several motivating examples. The one-, two- and three-slot mechanisms are examined and discarded due to problems, which are illustrated. The algorithms themselves are described in pseudo-code.

[109] Hugo Simpson. Correctness analysis for class of asynchronous communication mechanisms. *IEE Proceedings*, 139 Part E(1):35–49, January 1992.

This paper describes a variant of the four-slot mechanism as presented in Simpson's original exposition [108]. A correctness analysis is carried out using 'role-transition diagrams'.

[110] Hugo Simpson. *Methodological and Notational Conventions in DORIS Real Time Networks*. Dynamics Division, British Aerospace, February 1994.

The data orientated requirements implementation scheme (DORIS) is a methodology for the design of systems, primarily, real-time networks. This incorporates work on the data interaction architecture (DIA).

[111] Hugo Simpson. Interaction protocols, January 1996. Memo.

A number of protocols are described and categorised. RTL and pseudo-code descriptions are given of the various protocol families: channels, signals and pools.

[112] Hugo Simpson. Layered architecture(s): Principles and practise in concurrent and distributed systems. In *International Conference on the Engineering of Computer Based Systems, Monterety*, March 1997.

This is a brief exposition of the DORIS method.

[113] Eugene W. Stark. A proof technique for rely/guarantee properties. In *Foundations of Software Technology and Theoretical Computer Science*, volume 206 of *Lecture Notes in Computer Science*, pages 369–391. Springer-Verlag, 1985.

Stark describes a method where the proof of a finite collection of rely/guarantee statements can be used to prove a more substantial rely/guarantee statement.

[114] Joseph E. Stoy. *Denotational Sematics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.

A text covering the subject of denotational semantics very thoroughly. Although a relatively old text, it is still very relevant.

[115] David W. J. Stringer-Calvert. *Mechanical Verification of Compiler Correctness*. DPhil thesis, Department of Computer Science, University of York, March 1998.

Stringer-Calvert's thesis reruns an earlier (hand) proof of a high-integrity compiler. Several interesting points emerge, such as the benefits of formal, machine proof (implicit assumptions were uncovered and made explicit); and the difficulties with carrying out such proofs.

[116] Dirk Taubner. *Finite Representations of CCS and TCSP Programs by Automata and Petri Nets*. Number 369 in Lecture Notes in Computer Science. Springer-Verlag, 1989.

This book is a revised version of Taubner's doctoral thesis. The central idea is essentially the translation of process algebras to Petri nets and automata. Note that 'TCSP' here refers to 'Theoretical CSP' and not 'Timed CSP'. Theoretical CSP is the version described in Hoare's 1985 text [46].

[117] H. Tej and B. Wolff. A corrected failure-divergence model for CSP in Isabelle/HOL. In *FME '97: Industrial Applications and Strengthened Foundations for Formal Methods*, volume 1313 of *Lecture Notes in Computer Science*, pages 318–337. Springer-Verlag, 1997.

This work uncovered a previously unknown fault within the failures-divergence model of CSP involving the termination event. This illustrates the strength of mechanical formal methods: the requirement for extreme precision found this problem.

[118] F. Javier Thayer. An approach to process algebra using IMPS. Technical Report MP-94B193, MITRE Corporation, April 1995.

A mechanical implemention of CSP is constructed within the IMPS theorem prover using monoids. The author suggests a possible model for timed CSP, also within this framework.

Although mathematically elegant, the treatment using monoids is potentially a hindrance to wider use of the work.

[119] Lauren Wiener. *Digital Woes: Why we should not depend on software*. Addison-Wesley, 1993.

This is an intelligent and lively discussion of the problems with software, and society's increasing reliance on such software. A large number of examples illustrate this issue.

[120] Concurrency workbench. http://www.dcs.ed.ac.uk/home/cwb/index.html.

The Concurrency Workbench allows CCS [74] systems to be analyzed in a model-checking environment. It has several process semantics available, including Temporal CCS and SCCS.

[121] York Software Engineering. CADiℤ.

This tool supports the Z notation. It includes a syntax and type-checker, type-setting facilities, and a proof assistant. The interface to the proof assistant is based upon context-sensitive hypertext documents.

[122] Ada Core Technologies. GNAT, the GNU Ada compiler. http://www.gnat.com/.

GNAT is a member of the GCC compiler family.

[123] HOL. http://www.cl.cam.ac.uk/Research/HVG/HOL/HOL.html.

HOL is an interactive proof environment for higher-order logic developed at Cambridge. It is based on the meta-Language (ML) and is highly programmable.

[124] Isabelle. http://www.cl.cam.ac.uk/Research/HVG/isabelle.html.

Isabelle is a generic theorem prover developed at Cambridge. It permits new logics to be defined by specifying their syntax and inference rules. Proof procedures are expressed in terms of 'tactics' and 'tacticals'.

[125] Qiwen Xu, Willem-Paul de Roever, and Jifeng He. The rely-guarantee method for verifying shared variable concurrent programs. *BCS Formal Aspects of Computing*, 9(2):149–174, 1997.

This is a systematic illustration of the rely-guarantee approach to proving properties of systems.