An Implementable Formal Language for Hard Real-Time Systems

Steven Bradley

September 1995

Abstract

A real-time computer system may be demanded not only to produce correct results, but also to produce these results at the correct time. If high levels of assurance are required that such requirements are met, then standard verification techniques, such as testing, may not be adequate. In this case, more rigorous techniques for demonstrating correctness are required, and *formal* (i.e. mathematical) methods have been suggested as an alternative to testing. There are well established analysis techniques for verifying low level properties of real-time systems, mainly concerned with the scheduling of processing and communication resources. Unfortunately, these existing analyses are restricted to a level fairly close to the system implementation, and have difficulty in verifying high level, system wide properties. At the other end of the scale, much work has been done on providing abstract models for real-time systems, and on providing a theoretical framework in which system wide verification can be achieved. This work, however, is often far removed from the implementation considerations which are very important for real-time systems.

In this thesis I present a new language, AORTA (**A**pplication **O**riented **R**eal-**T**ime **A**lgebra), which aims to bridge the apparent gap between high level abstract reasoning, and low level implementation considerations. The language is restricted to allow direct and verifiable implementation, whilst retaining enough expressivity to give design solutions to real problems. Simulation and model-checking (formal verification) are discussed as means to provide assurance that AORTA designs satisfy their high level requirements. Implementation methods are also presented, based on code generation and process multitasking, along with analyses which allow guarantees about timing to be given. Finally, a framework for including data into the formal model is given, and is used to integrate AORTA with VDM.

Contents

Chapter 1

Introduction

Real-time computer systems are very much a part of everyday life. They can be found in cars, trains, aeroplanes, washing machines, toasters, chemical plants, power stations, telephone systems; anywhere that computers are used to control or interact with an environment where time is important. People have become used to computers, and also to their fallibility. No one hesitates to point the finger at a computer which causes a bank error, or which sends a bill for 0.1p, and yet many people are relatively unconcerned that embedded computers are used in more and more aspects of our lives. A bank error or bill can be corrected, but where a computer is used to control an aircraft, or a life-support machine, or a nuclear power station, a serious fault can have unthinkable consequences.

In such *safety-critical* situations, correctness is paramount, so the question is, how can systems be built to be more reliable? Even in situations which are not safety-critical, it may be very expensive to change a design. Because software can so easily be changed, people have grown accustomed to 'maintenance' of software. This may be possible for a piece of payroll software, of which there is only one copy, but for an embedded system, which may be produced millions of times, the cost of recall is great. So then, even for relatively harmless embedded control systems, correctness can still be of great importance.

How, then, can systems be built to be correct? Standard approaches, which rely on testing alone to discover errors, are flawed. Testing can demonstrate incorrectness, but not correctness. Testing on its own is particularly difficult to apply adequately to control systems, which may have millions of states. With the introduction of timing as an important factor, reproducibility becomes even more difficult, making sufficient testing virtually impossible. Engineers in more tradi-

tional areas, such as bridge building, rely on mathematical analysis of the problem and the proposed solution to give assurances that the bridge will function properly. Although tests would also be carried out, no one would trust a bridge whose only guarantee of integrity was that the engineer who designed it had driven a few cars across it, and it seemed OK.

Such condemnation of testing as inadequate for control systems, is common among proponents of formal methods. They insist that the only way to be sure that a system is correct is to *prove* that it is correct, i.e. perform a mathematical analysis of the problem and proposed solution. A formal development will be more expensive than an informal development, although some argue that this cost is outweighed by the reduced cost of maintenance. If fixing a problem after the event is not satisfactory, then it may be that a formal development, with its increased assurance of correctness, may be the only alternative available.

Different levels of formality may be achieved: perhaps only the specification will be developed formally; the design and coding may also be subject to mathematical analysis; or perhaps a fully verified solution on verified hardware will be attempted. Very little existing work on mathematical analysis of real-time systems enables a full verified development, from specification to implementation, to take place, although there is much work on high level specification, and much work on low level analysis of implementations. The aim of this thesis is to link the two areas, by providing a practical formal language which is amenable to high level formal specification and verification, and yet which is verifiably implementable.

The layout of the thesis follows roughly the steps which would be taken in a formal development of a real-time system. Chapter 2 introduces the problem area, and surveys and classifies existing work in the area of formal methods for real-time systems. The conclusions of chapter 2 give the motivation for the new language, which is defined, informally and formally, in chapter 3. High level analysis, through both formal verification and informal validation via simulation, is discussed in chapter 5. Chapters 6 and 7 then describe how designs written in the new language (AORTA), can be implemented and analysed for correctness. The formal inclusion of data into AORTA, using the VDM notation, is the subject of chapter 8, and chapter 9 evaluates the language, and sets it in a broader context. In conclusion, chapter 10 briefly recaps, and presents areas for further work. Proofs of all theorems in the thesis are presented in appendix A.

Chapter 2

The Hard Real-Time Problem

2.1 Introduction

This chapter serves as an introduction to the area of real-time systems and formal techniques for real-time systems. Section 2.2 gives a brief description of what a real-time system is, and section 2.3 explains the relevance of formal methods to the area. The bulk of the chapter lies in section 2.4, which surveys previous work on using formal methods for real-time systems, gives a taxonomy of them, and assesses the suitability of the different approaches for different stages of the development life-cycle. The conclusions drawn from this survey are then presented in section 2.5.

2.2 Real-time systems

Many computer systems are required not only to deliver correct results, but to deliver those results at the correct time — such systems are called *real-time systems*. These systems are most often to be found in control situations, such as in an automatic washing machine, a fly-by-wire system, a life-support machine, a car braking system, or an industrial plant controller. A *hard real-time system* must always meet its timing constraints, rather than providing satisfactory average performance. Many *safety critical* applications, where correct functioning is of vital importance because of the hazardous results of malfunction, fall into the category of hard real-time systems; many of the applications already mentioned would be classified as safety critical hard real-time systems. It is important to distinguish

between real-time systems and high performance systems, although some real-time systems may require high performance. A high performance system simply tries to get through its work as quickly as possible, whereas a real-time system may have timing requirements which are not simply upper bounds on completion. For instance, it would not be appropriate for a traffic light controller to cycle through its light sequence as quickly as possible, or for a communications protocol to re-transmit as soon as possible after an unacknowledged first transmission. Also high performance systems usually aim for highest possible average performance, rather than trying to guarantee the time behaviour in all cases. The use of techniques such as cacheing may improve average performance, but may degrade worst case performance and make analysis difficult. For hard real-time systems it is *predictability* of performance which is important.

It is surprisingly difficult to be able to guarantee that a computer system will give the correct results, and this problem is compounded when time behaviour also has to be verified. It is often the case that computerised control systems are most easily built using *concurrency* techniques, as different aspects of control have to be handled simultaneously. For real-time systems engineers, this leads to the problem of verifying not only the performance in time of a single computer program, but several interacting programs. As other aspects, such as the reliability and performance of computer hardware, may have to be considered, the problem area can be extremely daunting. For this reason, and because of the interesting theoretical and technological considerations, methods for building real-time systems are currently the subject of intensive research.

The importance of this research area is undeniable, as more and more areas of our lives are influenced by computer control systems. A recent investigation into the Therac-25 medical accelerator showed that timing related computer failures (among other things) lead to patients being subjected to massive, and in some cases fatal radiation overdoses [65]. Similar errors in even more time-critical systems, such as nuclear power plant controllers, could lead to unthinkable results.

2.3 Formal methods

Formal methods for computer system design are based on mathematics, allowing rigorous and unambiguous descriptions, be they specifications, designs or models of implementations. Also, the mathematical rigour of proof can be used in the verification of correctness. There are several reasons why formal methods are particularly

useful for real-time systems. Faults that may occur in such systems are not easily detectable by testing, and can be very difficult to reproduce; formal verification offers a more rigorous alternative to testing. Also, concurrency often leads to an explosion in the number of states of a system to a point beyond usual intuition, and formality can help to ensure that no possible state or sequence of events is erroneously ignored. Finally, if a system is safety-critical, the level of reliability required may only be deliverable by formal methods, for the reasons already given.

2.4 Previous and Current Work

Recent interest in formal techniques for real-time has generated many ways of describing real-time systems, some of which are based on earlier untimed formalisms. Description of real-time systems (and all other computer systems) falls into three main categories: specification, design and modelling. This classification is not universal and is largely concerned with the expressivity of languages, so it is worth elaborating on it slightly before outlining existing methods within this framework.

A specification of a real-time system describes how it should behave in time, and does not provide any detail of how this behaviour should be achieved. Typically there will be more than one design/implementation that will satisfy a specification, indeed if this is not the case then it is likely that the system is over specified. A specification language, then, should be expressive enough to allow enough to be said about a system to guarantee its correct behaviour (which is usually determined by the environment within which the system is to operate), whilst abstract enough to allow some areas of behaviour to be left unspecified.

A design contains enough detail of a system to indicate how it can be implemented, including how the implementation must perform in time. This can be difficult as the timing details of the implementation will not be known at the design stage. However, an iterative approach to implementation will allow figures to be included as they become available; the ability to leave some slack, such as time bounds rather than exact figures in the design, makes this approach easier. A design language should be expressive enough to allow a reasonable range of implementation techniques, yet restrictive enough (or include a suitably restrictive subset) to ensure that designs can indeed be implemented.

Finally, a model of a real-time system should accurately represent all aspects of the behaviour of a system which are of interest (i.e. those included in the specification). Thus, a model of a system can be used to verify the correctness of that

system with respect to its specification. A modelling language needs to be expressive enough to describe the behaviour of as many different kinds of systems as possible; the main reason for restricting a modelling language is to allow automatic verification that a model satisfies its specification (model-checking).

Some formalisms are aimed at one specific description area, while others try to provide a general framework in which more than one of these problems can be approached (such notations are sometimes called *wide-spectrum languages*). An important consideration for a formal description language is how verification can take place; in other words, what proof techniques are available. It is part of my thesis that many timed formalisms do not provide adequate practical proof techniques. In particular, timed extensions to untimed formalisms which are used as wide-spectrum languages rely on equivalences and refinements for their proof techniques. These methods do not appear to yield practical solutions to the problem of formal verification of timing requirements. Such a point of view is backed up by Ostroff [78], and the subject will be examined in more detail in chapter 3.

The following subsections outline the most relevant recent work, in groups according to their presentation, rather than their use, as many of the formalisms are not aimed at any specific area or areas. Some grouping is based on the nature of the model of time adopted, with two important properties being whether the time domain is discrete or dense, and whether it uses linear or branching time. A discrete time domain is characterised by the existence of a 'next time', the natural numbers (0,1,2,..) being the usual example. Dense time domains, on the other hand are such that for any two given times there exists another time value in between them; the rational numbers and the real numbers both form dense time domains. The notion of linear versus branching time models is most important for specification languages, where a linear time model assumes that for each run of the system there is only one possible (timed) sequence of events, and a branching time model allows for more than one possible behaviour. As well as classifying the various notations and techniques, some comments on the applicability of them to specification, modelling and design are given, all of which are my own. An alternative review of formal methods for real-time systems is given by Ostroff [78].

2.4.1 Timed logics

A timed logic deals with timed sequences of events or states of a system, and may allow quantification over behaviours of that system. A sentence of a timed logic can be used as a specification for a timed system: the specification is met if the

statement is true when interpreted in a model of the system implementation. Some logics allow specifications to be automatically checked for a system, using so-called *model-checking algorithms*. The existence of a model-checking algorithm for a logic is a definite point in the logic's favour, although this may impose some restrictions on the logical language. Non-automatic proof systems for temporal logics [48] may also be of use. for temporal logics which

The way in which states or events are described and the way in which time is handled varies from logic to logic, as well as the model of time used (i.e. discrete or continuous, linear or branching). Many timed logics are based on existing temporal logics, which are not in general timed, but deal only with the ordering of events within a behaviour. *Computational tree logic* [25], usually abbreviated to CTL, is such a temporal logic which uses a branching time model, which has given rise to timed logics such as RTCTL [32] (with discrete time), and TCTL [1] (with dense time). *Propositional temporal logic* (PTL) is a similar logic, based on a linear time model which has inspired RTTL [79], TPTL [3] and XCTL [46], all of which use a discrete time model. Some discussion of these logics is given in [3, 46, 78], and although there are some important differences in the expressiveness and complexity/decidability of satisfiability and model-checking, they have many common features. Similar logics exist which allow quantification over explicit intervals of time within a behaviour, including ISL [42] (linear dense time), an extension of CTL with time intervals by Lewis [66] (discrete branching time), and an interval logic by Melliar-Smith [68] (discrete linear time).

Another common untimed temporal logic, with a branching time model, is the *modal μ-calculus* [61], as used by the Edinburgh concurrency workbench [27]. Extensions to include time are given in [62], which uses a discrete time model, and in [20], which allows a dense time model. Both of these logics use models based on timed extensions to the process algebra CCS [70]. TRIO [38] is a branching time temporal logic which can be used with dense, discrete, or finite time domains. The most interesting point about this logic is that if the time domain is finite then logic specifications can be executed.

Timed temporal logics are useful for specifying real-time systems; they are expressive enough to state most requirements, although some of them are difficult to read and write. A temporal logic specification will not usually give the whole behaviour of the system, but only those features which are considered important. Indeed, it is difficult to describe a system completely in temporal logic, which makes them unsuited to design and modelling. (Executable temporal logics such as

TRIO [38] are used for validation of specifications rather than design work.)

There are other timed logics which do not fall into the category of temporal logics, but can be used in various ways to specify or describe timed systems. RTL [54] is a logic for expressing the times at which occurrences of events occur, and has been used in giving the formal semantics of Modecharts [53] (see section 2.4.4) and for including timing information into Z specifications [34] (also in section 2.4.4). Techniques have also been developed for checking safety conditions of RTL specifications [55]. The *duration calculus* is based on occurrences and lengths of intervals, rather than events, and has been used for giving the formal semantics of a language similar to timed CSP [22]. These two logics are mainly intended as specification or modelling languages (as indicated by their use in giving formal semantics [22, 53]); it is difficult to see how they could be used as design languages in their own right.

Finally, there are timed extensions of Hoare logic, which give a specification as pre- and post-conditions including time information. Refinement rules are given which allow the design problem to be broken down into steps. Some work is based on a design language similar to timed CSP [51, 52], and offers a complete compositional proof system. The main drawback with this work is that the language does not yield practically implementable designs. In [95], a timed Hoare logic is used in the development of Algol-like programs, which is more practical, but does not allow concurrency.

In general, timed logics are useful for specification. Timed temporal logics cannot easily be used for modelling, but they do often have automatic model-checking algorithms, which makes the verification problem easy in principle. Timed Hoare logics are also useful only for specification, and although they do not have model-checking algorithms, the manual proof systems which are available are easier to use than those for temporal logics. Logics such as RTL and the duration calculus are aimed at modelling and specification, so are to some extent wide-spectrum languages; the duration calculus is used to show the equivalence of two logical expressions, whereas RTL is usually only used at one level of abstraction. None of the timed logics described here are suitable for design, as the languages are too expressive to guarantee implementability, and it is difficult to find restrictions of the languages that are implementable. Timed logics are more commonly used for specifying properties of systems given in other languages; in particular, timed process algebras are often used in conjunction with timed temporal logics, and are discussed in the next section.

2.4.2 Timed process algebras

The theory of untimed process algebras is quite well developed, with a range of proof methods and software tools supporting the theory. The three main algebras are CCS [70], CSP [50] and LOTOS [77], each of which has its own distinctive characteristics. They have been used for specification, modelling and design, using *bisimulation* as the main proof technique for relating different levels of abstraction. Model-checking has also been applied to process algebras [27], so that a model or design can be verified with respect to a temporal logic specification.

Timed process algebras have been introduced to try to build on the success of their untimed counterparts, allowing the techniques that have worked well for untimed systems to be applied to real-time problems. Many timed process algebras are explicitly based on an untimed algebra, although there are a few that have been developed from scratch. One of the main classifications of process algebras is based on the model of time adopted, which may either be discrete or dense (most of the algebras which can use a dense time domain can also be used with a discrete time domain). Time can be introduced in any of three ways, firstly by allowing explicit time delays to be included as a separate construct, secondly by attaching time information to an action prefix for communication, and thirdly by introducing a time-out operator. All of the algebras mentioned here use one of the first two ways, and some use the third as well.

Discrete time process algebras fall into two categories: those which allow only one (possibly composite) action to take place per discrete unit of time, and those which have a distinguished tick action to represent the passage of time, allowing many actions to take place between ticks.

In [70], the language of Synchronous CCS (SCCS) is introduced as an underlying algebra for CCS and many other untimed algebras, but it can also be considered as a timed algebra, with all processes proceeding in lock-step, where each step takes one unit of time. As SCCS was not designed as a timed algebra, the time constructs are crude, using a τ prefix to represent a unit time delay. Each process must offer an action at each step, with the actions of each parallel process contributing to the composite action of the whole system. Another such algebra is CCSR [36], which is strongly influenced by SCCS, but also has priorities associated with actions. Timing information is introduced in CCSR using one monolithic construct which can be used for delays, time-outs and interrupts. CCSR is used in a verification technique called CSR [37], which gives interpretations of high-level real-time programming constructs

in terms of CCSR, when made into a system using a configuration language.

Other discrete time algebras use a distinguished tick action to represent the passage of time, with one unit of time passing for every tick action that occurs. In [63] a timed extension of LOTOS is given, which preserves upward compatibility with LOTOS; similar extensions, which do not retain compatibility are given to CCS in [62] and in [44], which also includes probabilistic information. Other versions of timed LOTOS in [11, 12] (which introduce explicit timers), and [89] (which attaches time information to actions) use discrete time, although they do not use a tick-based semantics. All three of these papers describe algebras which allow intervals of time to be specified for the availability of actions, but they do not allow any time nondeterminism to be introduced. ATP [75], which is not based on any particular untimed algebra, was originally defined with a discrete semantics, but was later adapted to allow dense time [74].

Algebras which allow the use of a dense time model fall into two main groups: those which introduce time by attaching information to actions and those which use explicit time delays. Intervals of time over which actions are available are used in [29] and [24] while a figure for 'internal rearrangement' time is used in [100]. These three algebras are all extensions of CCS, as are [71, 101, 104], which use time delays. A timed extension of CSP which uses delays and time-outs is given in [93], and [69] describes LOTOS with time delays, along with probabilistic information. A general time extension to ACP is given in [5], which uses an integration operator to introduce choices over time, and [49] outlines a new algebra called PARTY which uses time delays and time-outs.

Untimed process algebras have been used as wide spectrum languages, using bisimulation as the proof technique for relating different levels of abstraction. Unfortunately, timed bisimulations do not appear to be as useful, as the level of detail of the equivalence is much higher [78], so timed process algebras are not as useful in this respect. This point has not been addressed by the timed process algebras mentioned here, which are still aimed at being wide spectrum languages. Timed model-checking algorithms have been developed which may be more useful [1], although the complexity of these algorithms, and the number of states in a timed system, may render them impractical.

2.4.3 Graph-based formalisms

This group of work is based around the mathematical concept of a graph, that is a collection (set) of *nodes* or *vertices*, which may be connected pairwise by *edges*.

Graphs can be used to model the execution of a computer system by representing the state of the system by a node or set of nodes, and allowing transitions between states depending on the edges which connect the nodes. The classic example of a graph-based formalism is the *Petri net* [82], where the state is represented by the distribution of *tokens* among the nodes (referred to as the *marking*), which may enable or disable various *transitions* between states. (Technically speaking the transitions of a Petri net should also be considered as nodes of a graph, but they can be considered as compound edges.)

Time can be introduced into Petri nets by placing limits on when a token may enable a transition, or on when a transition may fire after having been enabled, or on how long a transition takes to fire once started. In [9], upper and lower bounds are placed on how long a transition may be enabled before firing, while [64] puts figures on how long a transition takes to complete once it starts. Both enabling times and completion times are used in [91], whilst [12] and [33] include nets which can place limits on the age of a token. A different approach is taken in [72], where the firing times are modelled as random variables, yielding a probabilistic analysis based on Markov chains.

The use of time within graphs is not restricted to timed Petri nets, and some theories have included time from the outset. Two such theories have been used with a timed temporal logic: model-checking in TCTL uses a version of timed graphs [1], and RTTL uses TTMs (Timed Transition Models) for representing systems [79]. Communicating Real-Time State Machines (CRSMs) [90, 96] use a graph-based notation for each of the state-machines in the system, but the treatment of a whole system is more closely related to a process algebra approach. Safety checking of RTL formulae can be carried out using a graph-theoretic approach [55], but the graphs used here are used to model the constraints placed on the system rather than the system itself.

Graphs are in general only used for system modelling, as they contain too much detailed information for use as a specification language, and allow structures which are too general to be implemented, particularly if a concurrent implementation is to be found. Proof techniques for graphs are usually based around checking that certain undesirable states are not reached (reachability analysis) or that a more general temporal formula is satisfied. For these reasons, timed graphs of some form are often used as an intermediate step when verifying a design. This approach is adopted in [75], where a process algebra term is translated into a timed graph, before using the results of [1] to verify its correctness with respect to a timed temporal

logic specification.

2.4.4 Others

Other timed formalisms exist which do not fall neatly into one of the above categories. Statecharts [58, 45] are, as they suggest, a state-based graphical formalism; hierarchical and composite states may be defined, and a small amount of timing information may be included. Modecharts are an extension to Statecharts which allow more timing information to be included, and the semantics of Modecharts is defined in terms of RTL [53]. Whilst useful for modelling and to some extent specification, there is little evidence that Statecharts can be used for providing an implementation which satisfies timing requirements.

Some work has been done on extending Z to include time. In [28] temporal lattices are defined to introduce timing information, whilst in [67] partial functions from time to states are used. Two papers by Fidge are based on Z, firstly [34], which links Z with RTL, and secondly [35], which describes how the Z refinement calculus can be adapted to include time. Z has been successfully used as a specification language, and these attempts to allow specification of timing requirements may also prove useful. However, the implementation of Z specifications has proved difficult without the refinement calculus, which cannot currently handle concurrency. VDM has also been extended, to include both time and concurrency, by the addition of some CCS constructs with time [99]. Although useful for specification, there are no available proof methods for timing properties. In [94] TAM is introduced, with a process algebra like language and a refinement calculus. However, the refinement calculus presented here raises similar problems to those associated with timed bisimulations, as the time behaviour of refined systems must be identical.

2.5 Conclusion

The bulk of this chapter has been taken up with a survey of formal techniques for specification, design and modelling of real-time systems. A parallel survey, on the subject of real-time scheduling and code timing techniques, could also have been given, as an introduction to mathematical techniques for the analysis of real-time implementations. This work, however, generally deals with low level concerns, and does not allow system level properties to be verified. It is the combination of low level analysis techniques with formal techniques such as have already been described, which is the subject of this thesis.

Given the many existing techniques for formal development of real-time systems just reviewed, it might at first appear that there is little scope for original research in the area. However, many of these formalisms have been developed as extensions of untimed methods, without consideration of whether or not the techniques that are useful for untimed systems can be usefully applied to real-time systems. Some languages aim for maximum expressivity, without examining the implications that this has for implementability and decidability of verification. Other approaches adapt untimed proof techniques without examining the applicability of these techniques by way of examples. The result of these two particular phenomena is that there is a general lack of practical techniques for verifiably implementing the formally defined systems, due to inappropriate languages or proof techniques. It is this lack of continuity from high-level specification to implementation through a set of verifiable steps that forms the focus of this thesis. Most of the work discussed in this chapter concentrates on high-level aspects, so the aim in the following chapters is to present a practical formal framework in which to build verifiable real-time systems.

Chapter 3

An Application Oriented Real-Time Algebra

3.1 Introduction

Following on from the review of existing formal techniques for real-time systems, this chapter develops a new algebra, AORTA (**A**pplication **O**riented **R**eal **T**ime **A**lgebra), which tries to address some of the problems identified in chapter 2. The basic approach taken is that of a timed process algebra, so section 3.2 discusses the advantages and limitations of timed process algebras as design languages. Section 3.3 introduces the concrete syntax of AORTA, and provides an explanation of its intuitive meaning, including a description of a mouse button driver in AORTA. The formal semantics of AORTA is given in section 3.4, along with an explanation of the semantics of the mouse button driver. Some properties of the transition systems which make up the formal semantics are presented in section 3.5, and section 3.6 presents the conclusions of the chapter. Much of the work of sections 3.3 and 3.4 of this chapter has been previously published [15, 16].

3.2 Timed Process Algebras for Design

Timed process algebras were reviewed in section 2.4.2, where the point was made that many of the algebras rely on timed bisimulation as a proof technique, despite a lack of evidence of the usefulness of this technique, and some indications to the contrary. In a bisimulation a relation is made between terms which have the same behaviour, and in a timed bisimulation related terms must have the same behaviour

in time. It seems that the level of detail given in existing timed process algebras is such that bisimulation equivalence makes too fine a distinction between systems, as is borne out by the profusion of definitions of timed bisimulations, but the lack of examples of equivalent systems (this view is supported in [79]). In providing a model which specifies exactly how a system must behave in time, it becomes very difficult to write an abstract specification. Even timed pre-orders for relating levels of abstraction suffer from the level of detail of the model, as refinement of timing of operations still requires a time for the operation to be specified at the highest level, making it very difficult to build an abstract specification. This does not mean that timed process algebras cannot be used at all, but that other methods for relating levels of abstraction must be found. In the conclusions of the previous chapter (section 2.5), it was noted that relatively little effort has gone in to producing techniques for the design and implementation stages of development; combining these two points leads to considering using a timed process algebra purely as a design language. There is very little difference in adopting this approach, to giving a concurrent timed programming language with a formal semantics

Process algebras in general, and timed process algebras in particular, have several points which make them suitable for use as design languages for real-time systems:

- Parallelism and communication are very natural concepts in process algebras, so the concurrency found in many real-time systems is easily handled, as is interaction with the environment.

- Nearly all of the untimed constructs in a process algebra can be interpreted computationally: parallel composition represents concurrency, actions represent inter-process communication, and choice represents branching. Work has been done on direct implementation of untimed process algebra terms [39, 98, 102].

- Compositionality of the parallel operator allows concurrent designs to be built up in a modular way.

- Proof techniques other than bisimulation can be applied, most notably pre-ordering and model-checking.

- A variety of semantic interpretations are available, including operational, denotational and axiomatic semantics.

- The language of a process algebra is small and can easily be understood without a detailed knowledge of the formal semantics.

There are, of course some drawbacks to using a general timed process algebra for designing real-time systems, including

- Some constructs (particularly those to do with introducing time) can have nonintuitive computational meanings. For instance, choice between time delays either requires both branches of the delay to be executed at once, which does not tie in with the notion of a sequential process, or requires an arbitrary choice to be made when time is allowed to progress, giving a nonintuitive interpretation.

- The specification of exact figures for timings does not correspond to best current practice in performance prediction, even in sequential systems. The best techniques for performance prediction can only give time bounds [80, 88, 81].

- The ability to have arbitrarily many processes created dynamically, by using parallel composition and recursion, makes the verification of timing in implementation a serious problem.

- It has proved difficult to provide a model for data which both corresponds well to possible implementations and has appropriate proof techniques (although some have tried [99]).

The approach of this thesis is to use a timed process algebra as a real-time design language, by introducing some restrictions and some extra expressivity to overcome these limitations, whilst still retaining as many of the benefits of using a process algebra as possible. In restricting AORTA to be a design language, timed bisimulations are not used as a proof method in the verification of systems, but rather model-checking with respect to a timed temporal logic. All of the basic features of AORTA are introduced in the next section, but facilities for describing and reasoning about data properties are given in chapter 8.

3.3 Concrete Syntax and Informal Semantics of AORTA

In the next section an abstract syntax is defined for AORTA, along with a formal interpretation of its meaning (its semantics). In this section an informal introduc-

tion is given to AORTA, including the concrete version of its syntax. An important feature of the concrete syntax is that it only uses ASCII characters, which makes writing designs using a computer much more straightforward, as well as giving the language more of a programming feel, rather than a mathematical one. Such feelings are not of great importance to many theoretical computer scientists, but they are important to non-specialists who have to use the language. The use of ASCII notation is emphasised by using `typewriter script` for expressions in the concrete syntax.

The syntax of AORTA is based on that of CCS [70], with extra constructs for the inclusion of time. There are some important differences, however, in the use of parallel composition and communication. Parallel composition, using the | operator may only appear at the top level, so that the number of processes within a system is statically defined. This is principally because it is much more difficult to carry out a timing analysis of a system which may have dynamic process creation, but it does have other spin-off advantages for modelling data and model-checking. Each component of a system is a sequential process, which may communicate with other processes or the environment, and is defined by a set of mutually recursive definitional equations.

All communication takes place through named *gates*, and causes the process to wait until the communication is completed — in other words it is *blocking* communication. A process which repeatedly engages in communication on the gate `a` is written with the single recursive equation

```
A = a.A
```

where the dot (`.`) is used to indicate communication prefixing, and the occurrence of `A` on the right hand side of the equation denotes recursion, in exactly the same way as CCS [70]. A similar condition is placed on recursion, namely that all recursion must be *guarded* (i.e. it must only appear inside an action prefix). Two other constructs which are identical to CCS are `+`, which is used for choice over communications, and `0`, which represents the nil process which can engage in no communications. Using these together, the process

```
Buffer = in.out.Buffer + stop.0
```

accepts communication on gates `in`, and `stop`; if `in` is accepted first then `out` is offered before restarting as `Buffer`, but if `stop` occurs then the process offers no further communications. In the basic language of AORTA, all communication is modelled as pure synchronisation, so although there is some intuition attached to

whether data is input or output during communication at a particular gate, this is not formally represented in the language. For implementation purposes, such information has to be provided with annotations, as described in chapter 6, and chapter 8 gives a formal semantics to the language extended to handle data.

Time can be introduced into expressions in two ways: via delays (representing computation or other time-consuming activity) or time-outs on communications. It is much easier to verify that an implementation satisfies time bounds rather than exact figures for times, so bounds can be given for both delays and time-outs. A time delay is denoted by the bounds on the elapsed time enclosed in square brackets before the subsequent behaviour expression. Adapting the above `Buffer` process to perform some computation in between communications gives

```
Buffer = in.([0.1,0.2]out.Buffer) + stop.0
```

where the computation takes between 0.1 and 0.2 elapsed time units (as opposed to required CPU time units) to complete. Time units for AORTA are not fixed, and the time domain may be discrete or dense (integers or reals); the requisite properties are given in section 3.4. Again, as data is not modelled in AORTA, no attempt is made to describe what takes place during the time delay.

Time-outs can be used to make a time dependent branch within a process. When communication is offered (possibly with choice), a time-out can be added, which enforces a different branch to be taken if no communication occurs within the specified time. Bounds can be used to give the time-out, as in an implementation the accuracy of the time-out would be at best the resolution of the system clock. The meaning of the bounds is that the time-out will not take place before the earlier time given, and will certainly have taken place by the later time if no communication has occurred. Our `Buffer` process can be further extended by a time-out on the `out` communication, which will ensure that any data associated with the output is up to date.

```
Buffer = in.([0.1,0.2](out.Buffer)[5.0,5.01>Buffer) + stop.0
```

The time-out value is specified within `[ .. >` with the communication bracketed to the left being timed, and the time-out behaviour to the right. Time-out can be thought of as an extension of choice, and is subject to the same restrictions: choice and time-out can only be applied to behaviours that start with communication. This is to ensure that the language has an obvious intuitive interpretation (see the first point on the list of drawbacks of using timed process algebras for design) and a straightforward implementation.

computation delay	`[t]S`
bounded computation delay	`[t1,t2]S`
communication	`a.S`
choice	`S1 + S2`
time-out	`(S1 + ... + Sn)[t>S`
bounded time-out	`(S1 + ... + Sn)[t1,t2>S`
data dependent choice	`S1 ++ S2`
recursion	equational definition

Table 3.1: Summary of AORTA sequential process concrete syntax

There are other kinds of branch which may need to be expressed in a real-time design language, which depend on data values within the system, rather than purely on synchronisation events. Because there is no data language within basic AORTA, this leads to a problem in the representation of such data-dependent choices, which is resolved by introducing a new choice operator, represented by the `++` symbol. This operator is used when there are two (or more) possible behaviours, and the choice between them is not based on a communication event. In terms of the basic language this is a nondeterministic choice, as there is no information as to how to resolve the choice; it is very similar to the nondeterministic choice ($\sqcap$) of CSP [50]. As an example of its use in AORTA, consider the problem of accepting data input and sending an alarm signal if the value exceeds a threshold safety level. The decision whether to send an alarm or not depends on data, and so uses the `++` operator:

```
Buffer = in.([0.1,0.2](out.Buffer ++ alarm.Buffer)) + stop.0
```

A process similar to this is used in the example described in section 4.2, and in the (AORTA) papers [13, 14]. The `++` operator is the last of the operators for sequential processes, and a summary of the syntax is given in table 3.1. There is clearly some ambiguity in the meaning of some expressions, but this is overcome by assigning precedences to the operators in the following order (tightest binding first): `. + ++ [..]`, and making timeouts (`[..>`) apply to the nearest communication. Round brackets can be used to override or emphasise this ordering where appropriate.

It may appear strange to introduce AORTA without a data language, and spend time developing detailed ideas for implementation without considering data, when chapter 8 shows how data can be included formally into the system. The reason for this is that the kind of systems which AORTA is aimed at — hard real-time systems — often have relatively little data dependency, and more often respond only to external events. By excluding data from the system a much simpler model is built,

which captures the essential features of the system (timing and communication) without too much extra baggage.

Having defined the syntax for defining individual processes, the syntax for composing these processes into a whole system is required. There is really only one operator required to do this, the parallel composition operator (|). Processes are placed in parallel, and the appropriate connections made between gates to allow communication. One of the distinctive features of AORTA as opposed to other process algebras, is that explicit connections must be made between pairs of gates that wish to communicate, using a *connection set*. These links are statically defined, and each gate must be connected exactly once (including external connections, which are also defined by the connection set). As different processes may have clashing gate names, gates in the connection set are referred to by the process name and the gate name. Each connection is given a bounded communication delay time, which is expressed as pair of bounds after a colon following the gate identifiers for each connection. The whole connection set is placed between angle brackets (`<..>`) after all of the parallel processes. (Strictly speaking each gate within the connection should be given its own communication delay, but as these are often the same within a connection, a single delay is a useful abbreviation.) External connections are listed using the keyword `EXTERNAL` in place of a gate identifier in the connection set. An example of a connection set is given in section 3.3.1, and several larger examples appear in chapter 4.

AORTA departs most markedly from other process algebras in its use of fixed parallel composition and explicit pairwise communication links. These restrictions are in place to make verifiable implementation easier, particularly with respect to timing. With a fixed number of processes, it becomes possible to give a straightforward means of verifying computation times when implementing a multitasking system, as shown in chapter 7. Similarly, having fixed channels of communication rather than a 'pool' of available actions looking for synchronisation partners (as in CCS or ADA) makes verification of time delays for communications much more straightforward. Questions may be raised as to whether these restrictions cause too great a loss in expressivity, which can be partially answered by the examples given in chapter 4; certainly some useful systems can be defined using AORTA. However, only prolonged and detailed exercising of the notation, along with associated tools, can truly evaluate its worth. See chapter 9 for a discussion of the limitations of, and possible extensions to, AORTA.

3.3.1 A Mouse Button Driver

Writing a mouse button driver which can differentiate between single and double clicks is clearly a real-time problem. This example is quite easily expressed in AORTA, and appears in [15]. If the mouse button is clicked twice within about 250 milliseconds then a double-click event will be offered; otherwise a press of the button will yield a single click event. This system quite naturally uses a time-out in its implementation, which is reflected in its expression in AORTA:

```
Mouse = click.(click.double.Mouse)[0.245,0.255>single.Mouse
```

`Mouse` is a recursive process, with recursion defined using an equational format. The time-out `[0.245,0.255>` ensures that if a click is not followed by another within about 250 milliseconds then a single click is offered. If the second click occurs within 245 milliseconds of the first then it will definitely be accepted as a double, and it may be accepted as such up to 255 milliseconds after the first. An implementation would follow quite easily from this, with a process which waited for a click, and then read a clock, before waiting for either another click or the current clock value to exceed the old one by 250 milliseconds. Depending on which happens first, a double or single click event will be offered before returning to wait for another click. As long as the clock was accurate to within 5 milliseconds it would have behaviour modelled by the AORTA expression, so any reasoning done on that expression would apply to the implementation. To define a system which uses this mouse, let us use a very simple computer, which reacts to one or two clicks on the mouse by performing a piece of computation.

```
Computer = one.[0.4,0.5]Computer + two.[1.2,1.4]Computer
```

Putting these in a system, we get

```
( Mouse | Computer )
<(Mouse.single,Computer.one:0.001,0.003),
 (Mouse.double,Computer.two:0.001,0.003),
 (Mouse.click,EXTERNAL:0.001,0.003)>
```

which specifies a 1 to 3 ms time delay for each possible communication. Figure 3.1 shows the graphical representation of this connection set.

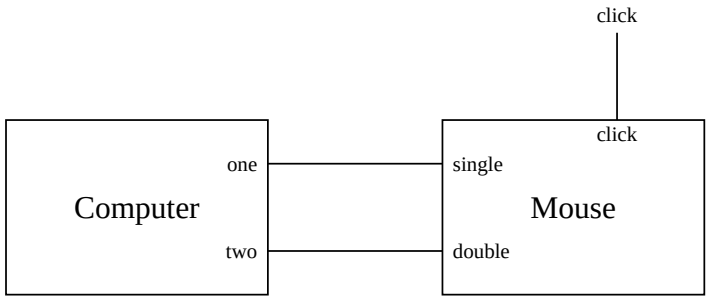

Figure 3.1: The Mouse Button System

3.4 Formal Semantics of AORTA

The advantages of giving a formal semantics are that the language is unambiguously defined, and that formal proofs of correctness can be provided for systems. Timed process algebras can be given a formal semantics in a variety of ways, including operational semantics (as in TCCS [71]), denotational semantics (as in TCSP [93]), axiomatic semantics (as in ACP_ρ [5]), and translation to another formalism (as in Modecharts [53]). Although the differences between these approaches are important, particularly as each has different proof techniques available, it is usually possible to provide corresponding semantics in different styles so the choice of semantic style is not necessarily crucial. A structured operational style of semantics is used here, defined by constructing a least relation which satisfies a set of transition rules. The advantage of this form of semantics is that it has a strong computational feel, and the definition of the semantics is relatively intuitive. It would be possible to provide a translation from AORTA to TCSP [93], say, and to use the derived (denotational) semantics. However, the translation would involve the use of nondeterministic summation over time bounds, leading to problems with automatic verification techniques, and a semantic model which would not correspond as closely with the computational one. In order to define the semantics, a formal abstract syntax is first presented.

3.4.1 Abstract Syntax and Time Domain Assumptions

Some care must be taken in giving an abstract syntax in association with a concrete syntax, to make sure that the two correspond closely, whilst maintaining the bias

of the concrete syntax to ease of use, and the bias of the abstract syntax to ease of semantic definition. In this case, some of the restrictions we place on the concrete syntax (such as the restriction of choice to occur only over communication) can be built in to the abstract syntax. Also, as we shall see in section 3.5, a more abstract representation of the syntax (such as a set of choices rather than a list of choices) leads immediately to some basic properties of the semantics. The abstract syntax is now given, and then the translation from the concrete to the abstract syntax is discussed.

In the abstract syntax, a system is expressed as a product (parallel composition) of sequential processes, each of which has a set of gates; gates of processes can be connected pairwise to allow communication. A system expression is then written

$$P = \prod_{i \in I} S_i < K >$$

where I is a set of process identifiers (typically names) S_i is a sequential process, and K is a set of unordered pairs of gates to be linked internally. Each gate is specified by its process (i.e. an element of I) and its gate name. At this level, the communication delay bounds for each gate must also be specified; for linked pairs the bounds may well be the same and will depend on internal communication delays, but for gates that communicate externally the delay will depend on what is being accessed and how. The delay bounds are specified by giving a function *delays* which takes a gate identifier, and returns an interval of possible communication delay times.

The structure of sequential expressions is given by the syntax

$$S ::= \sum_{i \in I} a_i.S_i \;\mid\; [t]S \;\mid\; \sum_{i \in I} a_i.S_i \rhd^t S \;\mid\; [t_1, t_2]S \;\mid\; \sum_{i \in I} a_i.S_i \rhd_{t_1}^{t_2} S \;\mid\; \bigoplus_{j \in J} S_j \;\mid\; X$$

where S and S_i are sequential processes, I is a finite indexing set, J is a non-empty finite indexing set, the a_i are gate names, t, t_1 and t_2 are non-zero time values taken from the time domain ($t_1 < t_2$), and X is taken from a set of process names used for recursion. These constructs correspond to action summation (a combination of action prefixing and choice), deterministic delay, deterministic time-out, nondeterministic delay, nondeterministic time-out, nondeterministic (data dependent) choice and recursive definition respectively.

Translation from the concrete syntax to this abstract syntax is reasonably direct. A list of processes to be composed in parallel, using $|$, are written with the product notation $\prod$. The abstract syntax enforces some conditions on the concrete syntax for sequential expressions; in particular, time-outs and choice can only be applied

to expressions which start with a communication. Choice (+) is generalised to summation ($\sum$) and data dependent choice (++) is generalised to nondeterministic summation ($\bigoplus$). The time-out

```
a.P[0.1,0.2>Q
```

is translated to

$$\sum_{i \in \{1\}} a.P \rhd_{0.1}^{0.2} Q$$

and the nil process `0` has the usual translation as summation over the empty set. It is possible to have a time-out on the nil process (this is sometimes used to force a delay), and is translated as a time-out over an empty summation. Some would argue that there is no need for an explicit delay construct, as this should behave in the same way as a time-out over the nil process, but they are considered as separate because they correspond to different computational objects: delays represent activities such as computation which take time to complete, whereas time-outs only occur after a calculated period of inactivity. This is an important distinction when it comes to implementation, as described in chapter 6. The distinction between time-out and time delay is also needed when reasoning about data within the system (see chapter 8). Similarly, choice without time-out can be derived from a choice with an infinite valued time-out, but this does not correspond to the way it would be implemented, so a separate construct is given to preserve the notion of distinct computational objects as distinct semantic objects. Each process is defined as a closed set of mutually recursive definitions within the abstract syntax. Parallelism is easily converted to the product form $\prod$, and the *delays* function is derived from the communication delay bounds attached to each connection or gate.

Whilst some researchers argue that a discrete time model is adequate for most computer systems, as computer hardware is generally based on a discrete clocking mechanism, others say that a dense time model is necessary to capture possible environment behaviour, and that the level of modelling does not usually take into account such low-level details as processor clocks. AORTA allows the time domain to be dense or discrete, and while these are usually associated with the non-negative reals and the non-negative integers respectively, any time domain $\mathcal{T}$ which satisfies the following conditions can be used:

1. There is an operation $+ : \mathcal{T} \times \mathcal{T} \to \mathcal{T}$ and an element $0 \in \mathcal{T}$ such that the structure $(\mathcal{T}, +, 0)$ is a commutative monoid (i.e. 0 is an identity w.r.t. $+$ on $\mathcal{T}$, and $+$ is associative and commutative).

$$
\begin{aligned}
regular(\sum_{i \in I} a_i.S_i) &= true \\
regular([t]S) &= regular(S) \\
regular(\sum_{i \in I} a_i.S_i \rhd^t S) &= regular(S) \\
regular([t_1, t_2]S) &= false \\
regular(\sum_{i \in I} a_i.S_i \rhd_{t_1}^{t_2} S) &= false \\
regular(\bigoplus_{j \in J} S_j) &= false \\
regular(X) &= false
\end{aligned}
$$

Figure 3.2: Definition of the *regular* function

2. There is a total order $\leq$ on $\mathcal{T}$ which has least element 0 (i.e. $\forall t \in \mathcal{T}.0 \leq t$), and which makes + monotonic (i.e. $\forall t_1, t_2, t_3 \in \mathcal{T}.t_1 \leq t_2 \Longrightarrow t_1 + t_3 \leq t_2 + t_3$).

There is an important subset of the sequential expressions called the *regular* expressions, which have no nondeterminism or recursion before the next possible communication. These are important because the semantics is only defined on regular expressions. In practice this often corresponds quite naturally to the computational model, as most nondeterminism (time or branching) comes from scheduling or data dependencies, and these do not change between communication events. Regular expressions are formally defined to be those which evaluate to true under the *regular* function given in figure 3.2. The usual condition on recursion, that it must be *guarded* by a communication event, is captured by the *regular* function.

3.4.2 Transition Rules

The semantics of AORTA is presented as a transition system defined in the structural operational style [84]. Transitions correspond to communication events (internal or external) or time transitions, based on an interleaving semantics (as opposed to a *true concurrency* semantics). One novel feature is the use of stratifications [43] of the transition system to enforce the *maximum progress principle* which insists on internal communications taking place as soon as they are enabled. Transitions of sequential expressions are calculated first, using the structural rules given in figure 3.3. Rules are only given for *regular* expressions (see section 3.4.1), and in particular no rule is given here for recursion. Recursion and nondeterminism and dealt with by the auxiliary *Poss* function, which defines possible regularisations of

$$\frac{}{\sum_{i\in I} a_i.S_i \xrightarrow{a_j} [t]S'_j} \quad \begin{array}{l} j \in I, S'_j \in Poss(S_j) \\ t \in delays(a_j) \end{array} \qquad \frac{}{\sum_{i\in I} a_i.S_i \xrightarrow{(t)} \sum_{i\in I} a_i.S_i}$$

$$\frac{}{[t]S \xrightarrow{(t')} [t\text{-}t']S} \; t' < t \qquad \frac{}{[t]S \xrightarrow{(t)} S}$$

$$\frac{}{\sum_{i\in I} a_i.S_i \rhd^t S \xrightarrow{a_j} [t']S'_j} \quad \begin{array}{l} j \in I, S'_j \in Poss(S_j) \\ t' \in delays(a_j) \end{array}$$

$$\frac{}{\sum_{i\in I} a_i.S_i \rhd^t S \xrightarrow{(t')} \sum_{i\in I} a_i.S_i \rhd^{t-t'} S} \; t' < t \qquad \frac{}{\sum_{i\in I} a_i.S_i \rhd^t S \xrightarrow{(t)} S}$$

$$\frac{S \xrightarrow{(t_1)} S' \quad S' \xrightarrow{(t_2)} S''}{S \xrightarrow{(t_1+t_2)} S''}$$

Figure 3.3: Transition rules for regular sequential expressions

non-regular expressions, including the unwinding required for recursion. The *Poss* function is defined in figure 3.4; as all elements of the sets defined by *Poss* are regular (provided mutual recursion is guarded) all transition rules lead to regular expressions, so the rules preserve regularity and the transitions are well-defined.

Communication transitions take the form $S \xrightarrow{a} S'$ and time transitions take the form $S \xrightarrow{(t)} S'$. Internal communications are like normal communications, but have the special label $\xrightarrow{\tau}$. Choice expressions can have communication transitions or time transitions. If a communication transition takes place, a communication time delay is chosen, and the appropriate branch expression is regularised. A similar communication rule applies to time-outs. Time transitions do not affect choice: the process waits indefinitely for communication unless a time-out is in effect. The time behaviour of time-outs and time delays is similar, with small time transitions reducing the amount of time left in the time-out/delay, and transitions equal to the given value passing control to the correct expression. Finally there is a general rule for time transitions which allows large time transitions to be made up from smaller transitions, simply by adding the times together (this is known as *time transitivity*).

The semantics of sequential expressions is completely defined by the transition rules and auxiliary function (*Poss*) just given, and this semantics is used to derive the semantics of system which contain processes composed in parallel. Enforcing the maximum progress principle, which insists that all internal actions must take place as soon as they become available, is the main technical concern. The most direct way to achieve the maximum progress principle is to add a side-condition to

$$
\begin{aligned}
Poss(\sum_{i\in I} a_i.S_i) &= \{\sum_{i\in I} a_i.S_i\} \\
Poss([t]S) &= \{[t]S' \mid S' \in Poss(S)\} \\
Poss(\sum_{i\in I} a_i.S_i \rhd^t S) &= \{\sum_{i\in I} a_i.S_i \rhd^t S' \mid S' \in Poss(S)\} \\
Poss([t_1\,,t_2]S) &= \{[t]S' \mid t \in [t_1,t_2], S' \in Poss(S)\} \\
Poss(\sum_{i\in I} a_i.S_i \rhd_{t_1}^{t_2} S) &= \{\sum_{i\in I} a_i.S_i \rhd^t S' \mid t \in [t_1,t_2], S' \in Poss(S)\} \\
Poss(\bigoplus_{j\in J} S_j) &= \{S'_j \mid j \in J, S'_j \in Poss(S_j)\} \\
Poss(X) &= Poss(S) \quad \text{if } X \stackrel{\text{def}}{=} S
\end{aligned}
$$

Figure 3.4: Definition of *Poss*

the rule for time delay, insisting no internal actions can take place. This leads to two immediate problems:

1. If the time domain is dense there is no notion of a next state or of a single step, so checking for internal actions has to take place over an interval in the future, and not just at the present instant. This can be achieved by introducing a *timed sort* [104], but this adds another layer of complexity to the semantics.

2. The use of conditions which refer to negative premises on transitions means that the usual least relation construct on operational rules may not be well-defined. For example, if there is a rule such as

$$\frac{S \stackrel{a}{\not\longrightarrow}}{S \stackrel{a}{\longrightarrow} S'}$$

   then there is no transition relation which satisfies this rule. Clearly such a rule is a silly one, but if there are any negative premises on transitions it must be shown that there is no chain of transition rules which might make a transition depend on its own negation, and hence make it impossible to find a satisfactory transition relation.

Figure 3.5 shows the operational rules used to give the semantics of AORTA systems. There is a negative premise on internal actions on the rule for time transitions, and a dense time domain may be used, so both of the problems just mentioned affect the definition of the semantics. The first problem is dealt with fairly directly: the

$$\frac{S_j \xrightarrow{a} S'_j \quad S_k \xrightarrow{b} S'_k}{\prod_{i\in I} S_i < K > \xrightarrow{\tau} \prod_{i\in I} S'_i < K >} \quad \begin{array}{l}(j.a, k.b) \in K \\ S'_i = S_i \text{ if } i \neq j, k\end{array}$$

$$\frac{S_j \xrightarrow{a} S'_j}{\prod_{i\in I} S_i < K > \xrightarrow{a} \prod_{i\in I} S'_i < K >} \quad \begin{array}{l}j \in I \\ (j.a, \_) \notin K \\ S'_i = S_i \text{ if } i \neq j \\ \prod_{i\in I} S_i < K > \overset{\tau}{\not\longrightarrow}\end{array}$$

$$\frac{\forall i \in I. S_i \xrightarrow{(t)} S'_i}{\prod_{i\in I} S_i < K > \xrightarrow{(t)} \prod_{i\in I} S'_i < K >} \quad \forall t' < t. \prod_{i\in I} Age(S_i, t') < K > \overset{\tau}{\not\longrightarrow}$$

Figure 3.5: Transition rules for system expressions

side condition on the rule for a time transition $\xrightarrow{(t)}$ can be written

$$\forall t' < t. \prod_{i\in I} S_i < K > \xrightarrow{(t')} \prod_{i\in I} S'_i < K > \implies \prod_{i\in I} S'_i < K > \overset{\tau}{\not\longrightarrow} \quad (\dagger)$$

This condition says that at all times between the present instant and t time units in the future, no τ transitions (internal communications) can take place. In this way, all internal communications are forced to occur immediately. Unfortunately, this side condition is rather unwieldy, so a new function *Age* is introduced, which calculates the state of a process in the future, and can be used to re-express the condition (†) as

$$\forall t' < t. \prod_{i\in I} Age(S_i, t') < K > \overset{\tau}{\not\longrightarrow}$$

The definition of the *Age* function (which only need be defined on regular expressions) is given in figure 3.6, and the use of the *Age* function to replace the side condition relies on the theorem

Theorem 1 *For all regular sequential expressions S and S', and all times t*

$$S \xrightarrow{(t)} S' \Leftrightarrow Age(S, t) = S'$$

where $=$ is syntactic identity on abstract syntax terms modulo equality on time expressions

The proof of this theorem can be found in appendix A, along with the proofs of all other theorems in this thesis.

The second problem, associated with negative premises in transition system specifications, has been studied by Groote and others [10, 43], who have provided

$$
\begin{aligned}
Age(\sum_{i \in I} a_i.S_i, t') &= \sum_{i \in I} a_i.S_i \\
Age([t]S, t') &= \begin{cases} [t\text{-}t']S & (t' < t) \\ S & (t' = t) \\ Age(S, t'\text{-}t) & (t' > t) \end{cases} \\
Age(\sum_{i \in I} a_i.S_i \rhd^t S, t') &= \begin{cases} \sum_{i \in I} a_i.S_i \rhd^{t\text{-}t'} S & (t' < t) \\ S & (t' = t) \\ Age(S, t'\text{-}t) & (t' > t) \end{cases}
\end{aligned}
$$

Figure 3.6: Definition of Age

techniques for defining transition systems in the presence of rules with negative premises. A simple technique called *stratification* [43] is sufficient to deal with negative premises as used in the definition of AORTA. In order to ensure that there are no cycles of dependency within the transition rules, a layering (stratification) of transitions is defined: this is typically a function from transition symbols (such as $S \xrightarrow{\alpha} S'$) to natural numbers. All positive premises within rules must be in the same stratum as the resultant transition, or in a lower stratum, and all negative premises must be in a lower stratum than the resultant transition. The transition relation is then built from the bottom up, with the lowest stratum transitions being evaluated first. For each stratum, the least relation satisfying all rules with resultant transitions within that stratum is calculated, which will depend on transitions from lower strata for all but the lowest stratum. The conditions on the defining rules guarantee that no cycles can exist, so there is a unique, well-defined transition relation for any stratified transition system which satisfies the conditions given. Groote goes on to prove that if a stratification of a transition system exists, then any other stratification will produce the same transition relation [43].

This theory applies very easily to the AORTA transition system to demonstrate its soundness. All sequential expression transitions are assigned to stratum 0. Clearly all sequential expression rules (given in figure 3.3) satisfy the stratification conditions, as there are no negative premises, and all other premises are on sequential expression transitions (in the same stratum). All system transitions which are labelled with τ fall into stratum 1. The only rule which defines a τ transition has premises which are in the lower sequential expression stratum, so the conditions are satisfied. Finally, all other system transitions, i.e. external communications or time transitions, fall into stratum 2. Again, all premises (positive or negative) for rules which define these kind of transitions come from stratum 0

(sequential expression transitions) or stratum 1 (τ transitions), so the conditions are satisfied. The AORTA transition relation is then calculated in stages: first the sequential transitions are evaluated, then the derived system τ transitions, and finally the other system transitions, which depend on the non-occurrence of certain τ transitions.

The use of a negative premise in the rule for external communication in systems is there to enforce a priority of internal communications over external ones. There is no deep theoretical reason why the priority should be this way round, but practical experience with AORTA has shown that internal communications more often have an urgency attached to them. The use of a more general priority scheme is possible (although some practitioners argue that too many levels of priority can lead to unnecessarily complex systems), and falls out quite easily with an extended stratification. If some priority ordering is placed on internal and external communications, as can be done quite easily in the concrete syntax connection set, this can be reflected within the stratification. A separate internal and external transition rule is then defined for each priority level, with a side condition that no higher-priority communications are enabled. A separate stratum is defined for each priority level, with higher priority levels associated with lower strata. The highest stratum is used for time transitions, which depend on no internal communications being enabled.

3.4.3 Semantics of the Mouse Button

The intuitive interpretation of the transition system formed by these rules is worth mentioning, as there is often some ambiguity, particularly in untimed algebras, as to the precise meaning. The two types of transition, $\xrightarrow{a}$ and $\xrightarrow{(t)}$ correspond to ability to communicate and ability to age. If $S \xrightarrow{a} S'$ then S is ready to communicate externally on gate a, and if this communication takes place the process will then become S'. If a system can communicate internally then it does (maximum progress principle, as enforced by the side condition on the delay rule), and this is represented by the distinguished action $\xrightarrow{\tau}$. If more than one τ action is possible then a nondeterministic choice is made between the available actions unless a priority ordering is given. The transition $\xrightarrow{(t)}$ describes how a system or process may age in time, and it is a property of the system that any process has only one way in which to age (see section 3.5). The behaviour of a system is then represented by a series of transitions, with the behaviour of the environment affecting which external communication events ($\xrightarrow{a}$) take place.

The mouse button process, which was described in section 3.3.1, is used here to

show how the formal semantics of a process can be derived. Its definition was

```
Mouse = click.(click.double.Mouse)[0.245,0.255>single.Mouse
```

and it was defined in parallel with a simple model of a computer

```
Computer = one.[0.4,0.5]Computer + two.[1.2,1.4]Computer
```

with the system definition

```
( Mouse | Computer )
<(Mouse.single,Computer.one:0.001,0.003),
 (Mouse.double,Computer.two:0.001,0.003),
 (Mouse.click,EXTERNAL:0.001,0.003)>
```

Firstly, the sequential transitions within the system can be worked out, using the rules of figure 3.3. As both `Mouse` and `Computer` begin with choice, the time transitions are very straightforward (note that we use the concrete syntax rather than the abstract syntax here):

$$\texttt{Mouse} \xrightarrow{(t)} \texttt{Mouse}$$

$$\texttt{Computer} \xrightarrow{(t)} \texttt{Computer}$$

and the action transitions include

$\texttt{Mouse} \xrightarrow{click} \texttt{[0.0025](click.double.Mouse)[0.249>single.Mouse}$

$\texttt{Computer} \xrightarrow{one} \texttt{[0.0012][0.41](one.[0.4,0.5]Computer + two.[1.2,1.4]Computer)}$

$\texttt{Computer} \xrightarrow{two} \texttt{[0.003][1.38](one.[0.4,0.5]Computer + two.[1.2,1.4]Computer)}$

There are many more possible action transitions of `Mouse` and `Computer`, as the *Poss* function allows the resolution of nondeterminism to any time value within the bounds, but they can only be via a `click`, `one` or `two` action.

The initial system transitions are derived from the initial sequential transitions by the rules of figure 3.5. As `click` does not appear in the connection set, and it is the only possible action transition of `Mouse`, no internal communication (τ transitions) can take place. Also

$$Age(\texttt{Mouse}, t) = \texttt{Mouse}$$

and

$$Age(\texttt{Computer}, t) = \texttt{Computer}$$

so we can derive the time transition

$$\texttt{(Mouse | Computer)} \xrightarrow{(t)} \texttt{(Mouse | Computer)}$$

as well as the action transition

$$\texttt{(Mouse | Computer)} \xrightarrow{click}$$

$$\texttt{([0.0025](click.double.Mouse)[0.249>single.Mouse | Computer)}$$

If the mouse button is not pressed again, the subsequent time transitions of the mouse process are

$$\texttt{[0.0025](click.double.Mouse)[0.249>single.Mouse} \xrightarrow{(0.0025)}$$

$$\texttt{(click.double.Mouse)[0.249>single.Mouse} \xrightarrow{(0.249)}$$

$$\texttt{single.Mouse}$$

so by the time additivity rule we can deduce the transition

$$\texttt{[0.0025](click.double.Mouse)[0.249>single.Mouse} \xrightarrow{(0.2515)}$$

$$\texttt{single.Mouse}$$

For these first 0.2515 seconds, only the `click` action is available, which is not internally connected, so we can derive the system time transition

$$\texttt{([0.0025](click.double.Mouse)[0.249>single.Mouse | Computer)} \xrightarrow{(0.2515)}$$

$$\texttt{(single.Mouse | Computer)}$$

At this point an internal communication between `single` and `one` becomes available, so no more time transitions can take place until the following transition has taken place

$$\texttt{(single.Mouse | Computer)} \xrightarrow{\tau}$$

$$\texttt{([0.0018](click.(click.double.Mouse)[0.245,0.255>single.Mouse) |}$$

$$\texttt{[0.0012][0.41](one.[0.4,0.5]Computer + two.[1.2,1.4]Computer)}$$

The computer can now execute for 0.41 seconds after its communication delay, so that after a time transition of 0.4112 we are back to the starting point of `( Mouse | Computer )`

3.5 Properties of AORTA Transition Systems

The previous section gave the formal semantics of AORTA in terms of a transition system which included time-labelled transitions. In this section we present various properties of the transition systems which are obtained, and discuss other results which cannot hold. First we deal with properties that relate specifically to timed transition systems, namely finite and bounded variability, time determinism, time additivity, and temporal deadlock, for which some definitions are required; these definitions are based on definitions given elsewhere [74].

Definition 1 (Run, Segment, Duration) *A* run *of a transition system is a sequence of pairs of states and labels* $(P_0, \alpha_0), (P_1, \alpha_1), \ldots$ *(where* α *is an action or time transition), such that for each* i *the transition*

$$P_i \xrightarrow{\alpha_i} P_{i+1}$$

is valid.

A segment *of a run is a finite subsequence of a run* $(P_m, \alpha_m), \ldots, (P_n, \alpha_n)$*, along with a final state* P_{n+1} *such that*

$$P_n \xrightarrow{\alpha_n} P_{n+1}$$

The duration *of a run segment is the sum of the time values of all time transitions in the segment.*

Definition 2 (Bounded variability) *A timed transition system is said to have* bounded variability *if for every time* $t \in \mathcal{T}$ *there is an upper bound on the number of action transitions that can take place in a run segment of duration* t

Bounded variability guarantees that only boundedly many actions may take place within a certain time, and is considered to be a necessary property for implementability, although very few timed process algebras satisfy it [74]. A related, but strictly weaker, condition is *finite variability*, which implies that an infinite number of actions cannot take place within a finite time, and is sometimes known as the *non-Zeno* condition, or *stutter-freeness*.

Theorem 2 *For all AORTA systems, the resultant transition system has bounded variability.*

Time determinism concerns the way in which a system can evolve in time: if there is only one 'direction' of evolution, then the transition system is said to have time determinism.

Definition 3 (Time determinism) *A timed transition system is said to be* time deterministic *if for all* P, P', P'' *and all* t

$$P \xrightarrow{(t)} P' \wedge P \xrightarrow{(t)} P'' \Longrightarrow P' = P''$$

where $=$ *is syntactic identity modulo equality on arithmetic expressions.*

Although AORTA allows the expression of nondeterministic timing information, the derived transition systems are themselves time deterministic, as all nondeterminism is resolved when an action transition takes place, using the *Poss* function.

Theorem 3 *For all AORTA systems, the resultant transition system is time deterministic.*

Time additivity concerns the use of consecutive time transitions, and requires that two short transitions are equivalent to one longer transition. This is a property which is necessary for any reasonable timed transition system.

Definition 4 (Time additivity) *A timed transition system is said to have* time additivity *if for all P, P', P'' and all t_1, t_2*

$$P \xrightarrow{(t_1)} P' \wedge P' \xrightarrow{(t_2)} P'' \Longrightarrow P \xrightarrow{(t_1+t_2)} P''$$

The time additivity property means that a run can be reduced to a form which contains no two consecutive time transitions.

Theorem 4 *For all AORTA systems, the resultant transition system has time additivity.*

In some timed process algebras, a state may be reached where no actions can take place, and in particular there is no way in which the system can evolve in time: such a condition is known as *temporal deadlock*.

Definition 5 (Temporal deadlock) *A transition system has* no temporal deadlock *if for all P there is some label α (a time or action label) and a state P' such that*

$$P \xrightarrow{\alpha} P'$$

There is some discussion as to whether this situation should be allowed. From our point of view, all systems should be implementable, and as there can be no real system with temporal deadlock, it should be excluded from the formalism. However, in a formalism which is intended as a modelling or specification language, temporal deadlock can be used to model an undesirable condition.

Theorem 5 *For all AORTA systems, the resultant transition system has no temporal deadlock.*

The other properties mentioned in [74] are *bounded control* and *action persistence*, but these are of less interest; suffice it to say that AORTA transition systems have bounded control, but do not have action persistence.

An important part of the thesis which I am presenting is that if timed process algebras are to be used as a language for design, then timed bisimulation is not an appropriate proof technique, as I hope to explain. It is easy to define timed

bisimulation, and in fact the definition for this style of timed transition system is identical to that for untimed transition systems as defined by Milner [70]:

Definition 6 *Timed bisimulation A relation $\mathcal{R}$ is a* timed bisimulation *if for all P, Q and α (where α is an action or time transition label) $P\mathcal{R}Q$ implies*

- *For all P', $P \xrightarrow{\alpha} P'$ implies there exists Q' such that $Q \xrightarrow{\alpha} Q'$ and $P'\mathcal{R}Q'$*
- *For all Q', $Q \xrightarrow{\alpha} Q'$ implies there exists P' such that $P \xrightarrow{\alpha} P'$ and $P'\mathcal{R}Q'$*

One way in which to use bisimulation as a proof technique, is to define a bisimulation on all expressions on the language, and to prove equations relating different language expressions as theorems. If the relation is a congruence (like this timed bisimulation), then equational reasoning can be used to verify the equivalence of, say, a model of an implementation and its specification. The usefulness of this approach depends upon how many equations hold within the system. Some theorems, such as the associativity and commutativity of `+`, `|` and `++` (as operators in the concrete syntax) follow directly from the translation into the set notation of the abstract syntax, as set union ($\cup$) is associative and commutative. However, these equivalences hold under the very strong relation of syntactic identity modulo set and arithmetic equality on the abstract syntax, which we do not expect to be a useful proof tool. Such a relation only allows the order in which choices and processes are written to be changed, apart from removing copies in choice (e.g. $a.P + a.P = a.P$). Other, weaker relationships, corresponding to untimed observation congruence [70], are difficult to find for timed process algebras, as the introduction of time adds greatly to the level of detailed demanded of a congruence. The kind of theorem which is necessary if any 'interesting' equivalences are to be proved is the *expansion theorem*, which relates purely sequential processes to processes which involve parallelism. However, other work has shown that such theorems cannot exist for certain classes of timed algebras [40], as the number of timers required cannot be reduced. Languages which allow only one delay or time-out per process, such as AORTA, fall into the class of languages which cannot have expansion theorems.

3.6 Conclusion

In this chapter the language and semantics of AORTA have been introduced, with the aim of providing a formally defined language suitable for real-time design. Much of the motivation for the distinctive features of AORTA (time bounds, static parallelism, static connection set) comes from the difficulty of implementing real-time

systems from existing process algebras. Many of the important features of process algebras have been preserved, in particular the use of abstract parallel composition to represent multitasking processes, parallel and distributed systems, and hardware processes. Although only multitasking implementations are considered in this thesis, it is hoped that other paradigms can be used to implement AORTA designs. The balance between expressivity and implementability is an important one: the language needs to be expressive enough to allow solutions to a reasonable range of hard real-time problems to be given, and yet restrictive enough to guarantee implementability. In order to demonstrate that a suitable balance has been achieved, chapter 4 gives a range of examples in AORTA, and attempts to classify the kind of problems that AORTA can solve, and chapters 6 and 7 give techniques for implementation and verification. Another important reason for restricting the language is to ensure that automatic verification by model-checking can take place: this is the subject of chapter 5.

Chapter 4

Examples in AORTA

4.1 Introduction

Chapter 3 introduced AORTA as a design language for real-time systems, giving informal and formal descriptions of the syntax and semantics, and provided some motivation for the language constructs chosen. One of the important points of chapter 2 was that expressivity in a language is not the only criterion to be used in judging its usefulness, and other criteria, which often conflict with expressivity, may also need to be applied. For example, in a timed logic, an additional criterion is the availability of automatic verification algorithms; for a design language such as AORTA, implementability is of great importance. At the centre of my thesis is the claim that it is possible to have a formal real-time design language which is both expressive enough to allow useful systems to be designed, yet which is implementable in general. This chapter attempts to demonstrate that AORTA is suitably expressive, by presenting a series of examples in sections 4.2 to 4.4, and a summary (section 4.5) of the kind of the problems which can be tackled successfully with AORTA; the implementability issue is tackled in chapters 6 and 7. A secondary function of the examples given here is to familiarise the reader with the language constructs which were introduced in chapter 3.

All of the systems described here have appeared in papers which have either been published or which are under consideration for publication. The chemical plant controller (section 4.2) has been used as an introductory example in several papers [13, 14, 16, 17] and the car cruise controller (section 4.3) has appeared, along with the mouse button driver (see section 3.3.1) in a journal article [15]. A paper on timed protocol design [19] describes the alternating bit protocol (section 4.4) and

another paper [18] is based on the use of AORTA to design a submersible controller; this example is not included here for reasons of space.

4.2 Chemical Plant Controller

As a first example, the chemical plant controller fills two purposes:

1. it describes a simple yet believable problem which is readily solved in AORTA
2. it exercises all of the language constructs of AORTA.

The controller has to monitor and log temperatures within a reaction vessel, and respond to dangerously high temperatures by sounding an alarm. Two rates of sampling must be provided, to be selected by the plant operator, each of which has its own corresponding output format for the logging function. If a reading lies outside the safety threshold the alarm must be sounded; the temperature must be sampled at least every two seconds, to ensure that the alarm is sounded within three seconds of a dangerous temperature condition.

The proposed solution to this problem is a system which contains two processes: one accepts data, checks and converts it, and manages changes of mode; another process logs the data, controls the logging rate, and prepares data for external down-loading when required. The first process, called `Convert` is defined by

```
Convert = in.(Convert2 ++ warning.Convert2)
          +
          mode.(changespeed.[0.3,0.4]Convert)[1.5,1.505>Convert
Convert2 = [0.001,0.004]
             (out.Convert)[1.5,1.505>Convert
```

Here raw data is accepted on the `in` gate, and is checked to see if it lies outside an acceptable range. As the check depends on the value of the data supplied, a data dependent choice (`++`) is used to determine whether a normal conversion should take place (by continuing to `Convert2`), or whether a warning signal should be sent first (via the `warning` gate). In addition to accepting data input, the `Convert` process can accept mode change instructions via the `mode` gate, which toggles between the two modes of operation. Upon receiving a mode change instruction, the process sends a `changespeed` message to the data logging process, and reconfigures itself (this computation takes between 0.3 and 0.4 seconds). Once control has passed to `Convert2`, the actual conversion is performed, taking between 0.001 and 0.004

seconds, and the resulting temperature is offered to the data logging process on the `out` gate. A time-out is placed on this communication, so that a new value is sampled from the reaction vessel regardless of how long the data logger takes to accept this data — it may be down-loading a long packet of data, so may not be ready to accept the data for some time.

The second process, `Datalogger` (shown below), accepts data from `Convert` on the `getdata` gate, and adds this item to its history. Once this is complete it waits for either a `speed` event, which is a signal from `Convert` to change the logging rate, or a `download` event, which is an external signal that the history to date should be down-loaded. If communication occurs on `speed`, then the process continues at the other sampling rate, by switching from the `Datalogger` section to `Datalogger2`, or *vice-versa*. A `download` event causes the whole history to be assembled into a single packet, taking between 0.5 and 1.0 seconds, and sent out on `senddata`. If neither of these events occurs within the sampling period (1.0 or 0.25 seconds), a time-out occurs, and a new data value is sampled.

```
Datalogger = getdata.[0.01,0.015]
                (speed.Datalogger2
                 +
                 download.[0.5,1.0]senddata.Datalogger)
                 [1.00,1.005>Datalogger
Datalogger2 = getdata.[0.001,0.015]
                (speed.Datalogger
                 +
                 download.[0.5,1.0]senddata.Datalogger2)
                 [0.25,0.255>Datalogger2
```

These processes are connected together and externally by defining the connection set, which is given as

```
(Convert | Datalogger)
<(Convert.changespeed,Datalogger.speed:0.001,0.003),
 (Convert.out,Datalogger.getdata:0.001,0.003),
 (Convert.in,EXTERNAL:0.001,0.003),
 (Convert.warning,EXTERNAL:0.001,0.003),
 (Convert.mode,EXTERNAL:0.001,0.003),
 (Datalogger.download,EXTERNAL:0.001,0.003),
 (Datalogger.senddata,EXTERNAL:0.5,10.0)>
```

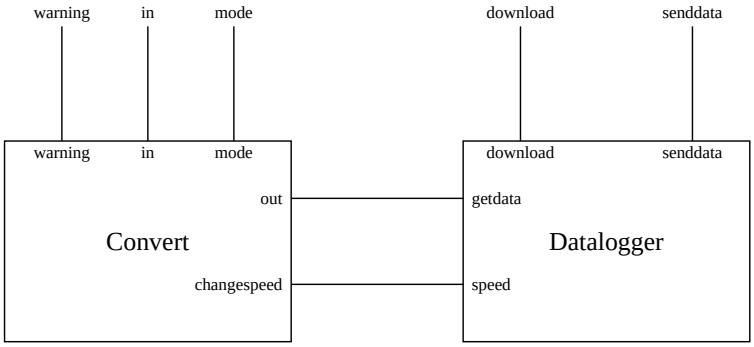

Figure 4.1: Chemical Plant Controller

Note that all communication delays are bounded by 0.001 and 0.003 seconds, apart from on the `senddata` gate, which has a much longer communication delay, as a reasonably large amount of data may be transmitted. Figure 4.1 shows a graphical representation of this connection set.

Note that four different kinds of external connection are assumed here:

Input: always available. This kind of gate is used to input a data value from the environment via a sensor which can always be read. In this example the `in` gate is expected to behave like this.

Input: sometimes available. This kind of gate is used for handling stimuli from the environment which are simple events such as button presses. In this example the `mode` and `download` gates are expected to behave like this.

Output: always available. This kind of gate can be used to send a simple signal, or to influence the environment by data passed with the signal. A simple signal is used in the `warning` gate, which indicates that an alarm should be sounded.

Output: sometimes available. This kind of gate can be used for outputting to a device which may not be ready, or may use a synchronisation protocol. In this example the `senddata` gate has to wait until the receiver is ready.

These different types of external connection correspond roughly to asynchronous and synchronous input and output. Asynchronous outputs and inputs may have time constraints relating successive occurrences of input or output events. In this example, the temperature must be sampled at least every two seconds, so there must be at most two seconds between successive temperature input events. Despite the

use of synchronous communication in AORTA, modelling asynchronous inputs and outputs is not a problem: it is simply assumed in these cases that the environment is always able to synchronise.

4.3 Car Cruise Controller

A slightly larger example than the chemical plant controller is a car cruise controller, which was originally defined to make comparisons between real-time design methods. The basic scenario is a car which may have its throttle automatically adjusted by a cruise control, which may be required to fix the current speed, or to perform a controlled acceleration or deceleration. Part of the specification as given by Hatley and Pirbhai [47] is

> The cruise control function is to take over the task of maintaining a constant speed when commanded to do so by the driver. The driver must be able to enter several commands, including: Activate, Deactivate, Start Accelerating, Stop Accelerating, and Resume. The cruise control function can be operated any time the engine is running and the transmission is in top gear. When the driver presses Activate, the system selects the current speed, but only if it is at least 30 miles per hour, and holds the car at that speed. Deactivate returns control to the driver regardless of any other commands. Start Accelerating causes the system to accelerate the car at a comfortable rate until Stop Accelerating occurs, when the system holds the car at this new speed. Resume causes the system to return the car to the speed selected prior to braking or gear shifting.
>
> The driver must be able to increase the speed at any time by depressing the accelerator pedal, or reduce the speed by depressing the brake pedal. Thus, the driver may go faster than the cruise control setting simply by pressing the accelerator pedal far enough. When the pedal is released, the system will regain control. Any time the brake pedal is depressed, or the transmission shifts out of top gear, the system must go inactive. Following this, when the brake is released, the transmission is back in top gear, and Resume is pressed, the system returns the car to the previously selected speed. However, if a Deactivate has occurred in the intervening time, Resume does nothing.

It also adds that in the implementation

> The system controls the car through an actuator attached to the throttle. This actuator is mechanically in parallel with the accelerator pedal mechanism, such that whichever one is demanding greater speed controls the throttle ...For smooth and stable servo operation the system must update its outputs at least once per second.

An AORTA implementation of such a system can be given by breaking the system into four processes: the speedometer subsystem, the controller subsystem, the throttle subsystem and the brake and gear subsystem.

Looking firstly at the speedometer subsystem, we assume that there is a measurement that can be made which will give the speed, but the acceleration is also needed (for the Accelerate function), and this is calculated and offered by this process. About every half a second the speed is reread and a new value for the acceleration calculated. For the rest of the time there are two gates on which the speed is available (one for the controller and one for the throttle mechanism) and one gate with the acceleration. Note that with a CCS-style communication mechanism [70], only one `speedout` gate would be needed, as the controller and throttle processes could share the same gate.

```
Speedo = (speedout1.Speedo2 + speedout2.Speedo2 +
             accelout.Speedo2)[0.4,0.5>Speedo2
Speedo2 = speedin.[0.2,0.3]Speedo
```

The delay `[0.2,0.3]` corresponds to the time taken to calculate the new value for the acceleration.

The subsystem which monitors the brakes and gears is also quite simple: it checks the state of the gears and brakes, and depending on what it finds offers a `fast` action, indicating that everything is fine, or a `slow` action, indicating that either the brakes are on or the transmission is not in top gear. The choice as to which to offer depends on the data which comes from `gearstate` and `brakestate` actions, and so is represented by a data dependent choice `++`. As in the speedometer, new readings are taken about every half a second, giving the definition

```
Brakengear = (fast.Brakengear)[0.4,0.5>
                gearstate.brakestate.(Brakengear ++ Brakengear2)
Brakengear2 = (slow.Brakengear2)[0.4,0.5>
                gearstate.brakestate.(Brakengear2 ++ Brakengear)
```

A slightly more complex subsystem is the throttle subsystem, which has to monitor the speed and ensure that it is kept at the speed specified by the controller. Also, it sometimes has to enter an accelerating phase, when it must keep control of the acceleration. Because of the parallel accelerator pedal mechanism, manual control will be resumed if the accelerator pedal is down further than the cruise control actuator, and in particular, if the actuator is set to zero the driver has complete manual control over the throttle. The output of the throttle controller subsystem is the `setthrottle` action, which sets the value for the actuator. In usual operation, the subsystem monitors the speed (via `getspeed`) and adjusts the throttle about every half a second. It may also allow a new speed to be set (by `setspeed`), the speed to be set to zero (`resetspeed`), or put the subsystem into an accelerating phase (`accelon`). During the accelerating phase the acceleration is monitored and adjusted until it is told to stop accelerating (`acceloff`), when the current speed is read for use as the new control speed and control returned to the usual state. If a `resetspeed` is encountered, the subsystem waits for a new value to be passed via `setspeed` before returning to usual operation. All of this is described in AORTA by

```
Throttle = (setspeed.Throttle + accelon.Throttle2 +
             resetspeed.Throttle3)[0.4,0.5>
               getspeed.setthrottle.Throttle
Throttle2 = (acceloff.getspeed.Throttle)[0.4,0.5>
             getaccel.setthrottle.Throttle2
Throttle3 = setthrottle.setspeed.Throttle
```

The subsystem which has overall control, and which accepts commands from the driver (or at least from an interface to the driver) has a more complicated logical structure, but does not have any real computation to do nor any time dependent behaviour in the form of timeouts. There are four major states of the controller, corresponding to the controller being inactive, the controller being active, the controller being in an accelerating state, and the controller waiting for a Resume after the brake being pressed or the transmission leaving top gear. Before entering an activated state, the controller has to check that the brakes are not on and that the transmission is in top gear, which it does by looking for a `fast` from the brake and gear subsystem — if a `slow` is encountered the subsystem must be reactivated. The speed is checked, and if it is greater than 30 mph the controller becomes active, otherwise the subsystem must be reactivated. As this decision is based on the data

about the speed, it is modelled by a data dependent choice ++, giving the structure

```
Cont = activate.(fast.checkspeed.(Cont ++ setspeed.Cont2) +
           slow.Cont)
```

Once activated, the subsystem must allow itself to be deactivated, or suspended due to a brake/transmission event, or put into the accelerating step. This is achieved by

```
Cont2 = (deactivate.setspeed0.Cont +
         startaccel.Cont3 + slow.Cont4)
```

During acceleration the subsystem can be deactivated, or stopped from accelerating, or suspended by a brake/transmission event

```
Cont3 = accelon.(deactivate.acceloff.setspeed0.Cont +
                 stopaccel.acceloff.Cont2 + slow.acceloff.Cont4)
```

Finally, if a brake/transmission event occurs, the speed must be reset and resumption or deactivation allowed, checking that the brakes and gears are in order if necessary.

```
Cont4 = setspeed0.(resume.fast.setspeed.Cont2 +
                   deactivate.Cont)
```

The whole system is then defined as

```
(Cont | Speedo | Brakengear | Throttle)
   <(Cont.checkspeed,Speedo.speedout2:0.001,0.004),
    (Cont.acceloff,Throttle.acceloff:0.001,0.004),
    (Cont.accelon,Throttle.accelon:0.001,0.004),
    (Cont.setspeed0,Throttle.resetspeed:0.001,0.004),
    (Cont.setspeed,Throttle.setspeed:0.001,0.004),
    (Cont.slow,Brakengear.slow:0.001,0.004),
    (Cont.fast,Brakengear.fast:0.001,0.004),
    (Speedo.accelout,Throttle.getaccel:0.001,0.004),
    (Speedo.speedout1,Throttle.getspeed:0.001,0.004),
    (Speedo.speedin,EXTERNAL:0.001,0.004),
    (Brakengear.brakestate,EXTERNAL:0.001,0.004),
    (Brakengear.gearstate,EXTERNAL:0.001,0.004),
    (Cont.activate,EXTERNAL:0.001,0.004),
```

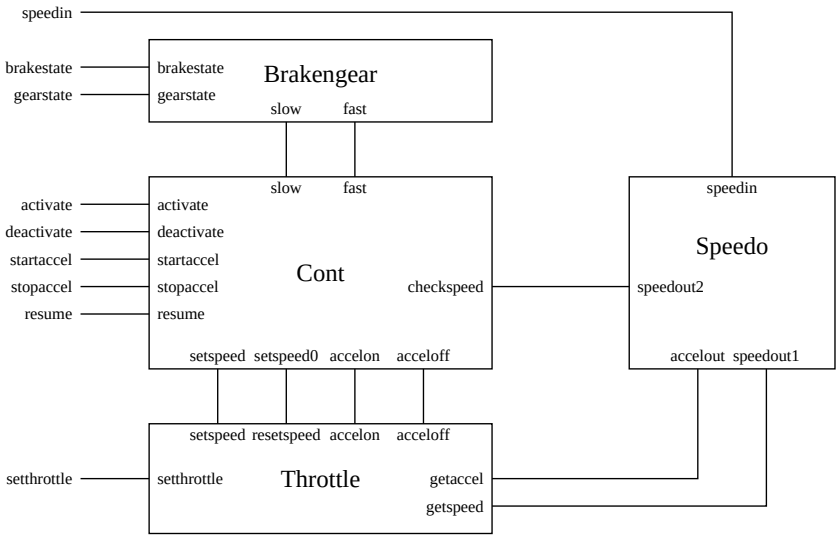

Figure 4.2: Car Cruise Controller

```
    (Cont.deactivate,EXTERNAL:0.001,0.004),
    (Cont.startaccel,EXTERNAL:0.001,0.004),
    (Cont.stopaccel,EXTERNAL:0.001,0.004),
    (Cont.resume,EXTERNAL:0.001,0.004),
    (Throttle.setthrottle,EXTERNAL:0.001,0.004)
>
```

Figure 4.2 shows the graphical representation of the complete system. All of the external connections of the cruise controller fall into one of the four categories given in section 4.2: `speedin`, `brakestate` and `gearstate` are inputs which are always available; `activate`, `deactivate`, `stopaccel` and `resume` are inputs which are sometimes available, and `setthrottle` is an output which is always available. There are many similar combinations of constructs in the chemical plant controller and the cruise controller, such as the use of an external input which is always available followed by a data dependent choice, and a time-out driving a loop, with exceptional events (such as `setspeed` and `accelon`) offered in choice.

4.4 Alternating Bit Protocol

This section describes a slightly different kind of example, and illustrates a different way in which AORTA can be used. Previous examples involved the design of systems which contained parallelism simply to make design easier. All of the processes which were defined would be implemented as part of the solution to the given problem, and all interaction with the environment was modelled by the availability of external actions. In this example, two processes must run in parallel, as they are physically remote from one another, and two other processes are used to model the behaviour of the environment. Although it has been previously and repeatedly stated that AORTA is meant for design of real-time systems, this section demonstrates that some useful system modelling can be done with AORTA. In a sense this is only an interesting novelty, at least as far as this thesis is concerned, but the possibility of modelling environmental activity by AORTA processes is one which is worth exploring.

Rather than an industrial controller, the alternating bit communication protocol is considered. This protocol relies on a timer to control retransmission of messages which may be duplicated or lost by a noisy buffer, and relies on message and acknowledgement tagging to ensure that all messages are eventually transmitted. The alternating bit protocol was originally proposed as a technique for giving reliable full duplex communication [7], and has been studied by many people as an exercise for their notation. Clarke *et al* [25] discuss automatic verification of the alternating bit protocol, and the AORTA design given here is based on Milner's CCS version [70], extended to include time explicitly.

The problem is to construct `Send` and `Reply` processes which can be connected by possibly noisy buffers `Trans` and `Ack`. The way the alternating bit protocol works is that messages and acknowledgements are tagged with a bit (0 or 1), with successive messages and acknowledgements being tagged with alternating bits. Having sent a 0-tagged message, the sender waits for a 0-tagged acknowledgement; if one does not arrive within a certain amount of time, the message is sent again. Once an acknowledgement does arrive, the next message can be sent, this time tagged with a 1. The replier, meanwhile, waits for a 0-tagged message, delivers the data, and sends a 0-tagged acknowledgement. If it receives another 0-tagged message it simply sends another 0-tagged acknowledgement, but a 1-tagged message causes it to deliver the message and return a 1-tagged acknowledgement and so on.

The `Reply` process is the simpler of the two, and is defined in AORTA as follows

```
Reply = trans0.Deliver0
Deliver0 = deliver.Reply0
Reply0 = reply0.(trans1.Deliver1+trans0.Reply0)
Deliver1 = deliver.Reply1
Reply1 = reply1.(trans0.Deliver0+trans1.Reply1)
```

A time-out is added to the `Send` process to resend messages if acknowledgements are not sent within a certain amount of time.

```
Send = accept.Send0
Send0 = send0.Sending0
Sending0 = (ack0.Accept1 + ack1.Sending0)[100.0,101.0>Send0
Accept1 = accept.Send1
Send1 = send1.Sending1
Sending1 = (ack1.Accept0 + ack0.Sending1)[100.0,101.0>Send1
Accept0 = accept.Send0
```

The buffer processes `Trans` and `Ack` can also be modelled in AORTA. A simple model of these processes has them accepting transmissions or replies, and passing them on after a certain delay, but without loss or replication. In AORTA, this is written

```
Ack = reply1.([25.0,75.0]ack1.Ack)
      +
      reply0.([25.0,75.0]ack0.Ack)

Trans = send0.([25.0,75.0]trans0.Trans)
        +
        send1.([25.0,75.0]trans1.Trans)
```

Notice here that if the minimum delay of 25 time units is taken by both buffers then the acknowledgement will reach the `Send` process before the 100 unit time-out expires, but if the maximum delay of 75 is incurred, then the sending process will retransmit its data. These processes are connected up with the `Send` and `Reply` processes, by defining the parallel composition and connection set

```
( Send | Reply | Ack | Trans )
<(Send.send0,Trans.send0: 0.5,1.0),
 (Send.send1,Trans.send1: 0.5,1.0),
 (Send.ack0,Ack.ack0: 0.5,1.0),
```

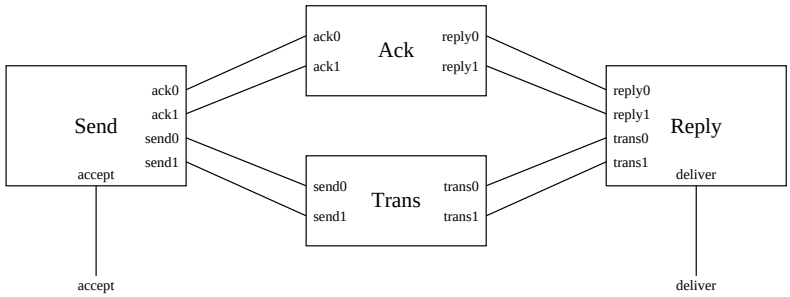

Figure 4.3: Alternating Bit Protocol System

```
(Send.ack1,Ack.ack1: 0.5,1.0),
(Reply.reply0,Ack.reply0: 0.5,1.0),
(Reply.reply1,Ack.reply1: 0.5,1.0),
(Reply.trans0,Trans.trans0: 0.5,1.0),
(Reply.trans1,Trans.trans1: 0.5,1.0),
(Send.accept,EXTERNAL: 0.5,1.0),
(Reply.deliver,EXTERNAL: 0.5,1.0)>
```

The layout of the system is shown in figure 4.3.

Different assumptions about buffer behaviour can be built into the system by altering the `Trans` and `Ack` processes and simulating the behaviour of the new system. For example, to include the possibility of the transmit buffer losing one of the messages we can use non-deterministic choice to represent failure. using the `++` operator, which chooses between the branches non-deterministically, we can have one branch as normal behaviour and another as faulty behaviour. A version of the `Trans` process which allows for the possibility of messages being lost, but which has no delays involved except internal communication delays, is given by

```
Trans = send0.((trans0.Trans) ++ Trans)
        +
        send1.((trans1.Trans) ++ Trans)
```

but this process allows arbitrarily many messages to be lost. This is the general case for such a channel, but no bounded response theorems can be proved of a system which may have to repeat a message arbitrarily many times. To describe a channel which may lose at most one copy of each message, this must be changed to

```
Trans = send0.((trans0.Trans) ++ (send0.trans0.Trans))
```

```
+
send1.((trans1.Trans) ++ (send1.trans1.Trans))
```

Such use of the `++` operator to represent failures is a powerful tool in modelling assumptions that can be made about a system, and can be used to evaluate to what extent a system may be perturbed before losing functionality. The duplication of messages can be handled in a very similar way.

Having modelled the behaviour of the environment by these two processes, the behaviour of the whole system can be simulated or formally verified (see chapter 5). Once it has been demonstrated, by whatever means, that the whole system is satisfactory, the `Send` and `Reply` can be implemented, using the techniques described in chapter 6. The paper which defines the alternating bit protocol in AORTA [19] also describes how simulation, verification and code generation are applied to this example.

The only external gates listed in the connection set are `accept` and `deliver`, which are respectively inputs and outputs which are sometimes available. However, as the `Send` and `Reply` processes would be implemented separately from the `Ack` and `Trans` processes, all the gates of `Send` and `Reply` would have to be implemented as external. All of these gates are also inputs and outputs which are sometimes available, but they could not be implemented directly, as different gates are used for differently labelled messages (e.g. `reply0` and `reply1`). In fact, they would have to check the tag before accepting or sending the data. This can be achieved by the output function which is attached to the external gate (see chapter 6), or it could be written more explicitly within the original AORTA design. In this case, any message would be accepted, and the future behaviour would depend on the tag (i.e. the data associated with the message). Adapting the `Reply` process to do this gives

```
Reply = trans.Deliver0
Deliver0 = deliver.Reply0
Reply0 = reply.(trans.(Deliver1 ++ Reply0))
Deliver1 = deliver.Reply1
Reply1 = reply.(trans.(Deliver0 ++ Reply1))
```

with appropriate tags attached to the data associated with `reply` and `trans`. The advantage with the original approach is that the control of the system is deterministic, and properties of the system can be checked directly without including a formal language for specifying and verifying data properties. Chapter 8 discusses how data can be included into AORTA designs, and how this data can be reasoned about.

4.5 Conclusion

The aim of this chapter was to demonstrate that AORTA is expressive enough to be used in the design of real systems, by presenting some semi-realistic examples. A classification of the kind of environmental behaviour that can be expected has gone alongside these examples, and has focussed on the expected behaviour of externally presented gates. These gates fall into four categories (detailed in section 4.2), characterised as inputs and outputs which are always and sometimes available. Inputs and outputs which are always available are often constrained by time requirements on successive access via the gate. In processor scheduling theory external events trigger the execution of *tasks* which run to completion. Tasks which manage inputs and outputs which occur regularly are referred to as *periodic*; in AORTA a periodic input or output is handled with a time-out round a loop. Synchronous inputs, i.e. those which are only occasionally available, correspond more directly with the AORTA internal communication mechanism, and generate tasks which are referred to as *aperiodic*. If an aperiodic task has a lower bound on the time between successive triggerings, it is called *sporadic*. A set of independent periodic tasks can easily be analysed using the *rate monotonic* approach [59], but where tasks intercommunicate, and have many sporadic inputs, AORTA seems to be more appropriate.

The examples show that AORTA can usefully be applied to situations with a mixture of periodic and sporadic inputs, and also to model some situations where environmental behaviour cannot be classified simply as periodic or sporadic. As time bounds, rather than exact figures, are given by AORTA, and as time-outs occur with reference only to the most recently offered communication, it is difficult to design a system which requires an *exactly* periodic input or output. Systems which have specifications which place upper bounds on the time between event occurrences are much more easily handled.

Chapter 5

Validation and Verification of Designs

5.1 Introduction

One of the justifications that is given for using a formal approach to system development is that errors can be found earlier in the life cycle of the product, and so are easier to correct. Although the novelty of the work reported here lies in the fact that the formal language is *implementable*, the use of formality to find errors *before* implementation is still of crucial importance. If the ability to find errors early on in development is a perceived advantage of formal methods, then the major perceived disadvantage is that the techniques involved are only to be attempted by specialists, and take an inordinate amount of time to complete. It is certainly true that a formal refinement, verified by computer-checked proof, is a very large undertaking, and not one to be attempted by an untrained software engineer. In order to address these problems, there are two obvious alternatives:

1. don't attempt formal verification
2. make formal verification easier and quicker

The first approach has been adopted, with some success, in IBM's CICS project [83], which uses Z [87] to specify components of a major commercial software system. No proofs of correctness are attempted, but the act of stating requirements formally has given productivity benefits. A more involved approach, but one which is still far short of formal verification, is that of using the formal semantics of the language to provide a simulation of the design. Section 5.2 looks at how to provide simulations

of AORTA designs, which can be used to validate the correctness of designs before they are implemented.

The second approach is obviously a challenge: everyone in computer science would like to make formal verification easier. One point that should be stressed is that it is (hopefully) impossible to verify formally the correctness of an incorrect system, and any attempt to do such a thing can waste a lot of time and effort. Only the most hardened formal methodologist would claim that people who use formal methods do not make mistakes, so some way must be provided of finding and eliminating mistakes before verification is attempted. This is another reason for using simulation at an early stage. A second point to make is that formal proofs of correctness of computer systems are extremely long, detailed, and almost entirely devoid of the 'clever' steps that are to be found in proofs in mainstream mathematics. For this reason it is a good idea to use a computer to keep track of the proofs, as humans are slow and prone to making mistakes in such circumstances, whilst computers are very good at keeping track of large amounts of information, and can do simple things extremely quickly. In fact, real-time systems can often be modelled as finite state systems which can be verified entirely automatically using *model-checking* techniques. In principle, the user of a model-checker simply has to state the result they are interested in, press a button, and out comes the answer. To do this, little specialised knowledge is needed, and little human time is taken up. In practice, model-checking is extremely demanding on time, particularly if quantitative time is included; all the same it is a useful technique, and is discussed in section 5.3. Work on the use of simulation and model-checking for AORTA has also been published [19].

5.2 Validation by Simulation

Simulation can be thought of as high-level testing, and has the same merits and pitfalls as testing. The advantages of simulation over more standard methods of testing is that errors can be detected early in the development, and that it is possible to 'look inside' the system under test. As formal methods aim to find errors early in the development, but can waste a large amount of effort if an attempt is made to verify an incorrect system, simulation at an early stage seems to be a useful complementary tool. Simulation can also have the advantage of being informative to the non-specialist; in particular, it is a useful vehicle for design validation by a specialist in the application area who does not know about the design or specification

language in use. Indeed, much work has been done on execution of temporal logic specifications for validation of specifications [6, 38, 73]. Two approaches to simulating AORTA designs are described in this section. Firstly a simulation based on the formal semantics is described, where a menu is used to choose which transitions are to take place (section 5.2.1). Secondly a more interactive type of simulation is used, where real time in the simulated system is modelled by real time (suitably scaled) in the simulation (section 5.2.2).

5.2.1 Menu Driven Simulation

The formal semantics of AORTA is given in terms of a timed transition system defined by operational rules (see chapter 3). If an AORTA expression S_1 becomes S_2 after t units of time, this is written

$$S_1 \xrightarrow{(t)} S_2$$

and if a communication a takes place, this is written

$$S_1 \xrightarrow{a} S_2$$

where a can be a gate name (for external communication) or the distinguished action τ (for internal communication). These transitions can be seen by using the AORTA simulator, which takes an AORTA description of a system (such as those described in chapter 4) and allows the behaviour to be stepped through using a simple menu-driven system. This much is similar to other simulators, such as can be found in the concurrency workbench [27] or other tools, but there is also a facility for showing graphically which communications, internal and external, are available. Figure 5.1 shows a screen shot of the simulator being used on the alternating bit protocol design, with an internal action available — the corresponding connection is shown as a dashed line.

After each transition (time or action), a menu of possible further transitions is offered, with any possible action transitions displayed on the system layout diagram. If an action transition is chosen then the corresponding connection is flashed on the diagram. If a time transition is chosen then a further prompt asks for a time value, or one of the commands `NEXTCRUCIAL` and `NEXTCOMM`. The two commands progress time up to the next crucial point (the end of a delay or time-out), and the next possible internal communication respectively. If a time value is given the system will be aged by that amount, provided it does not pass through a state where an internal communication is possible.

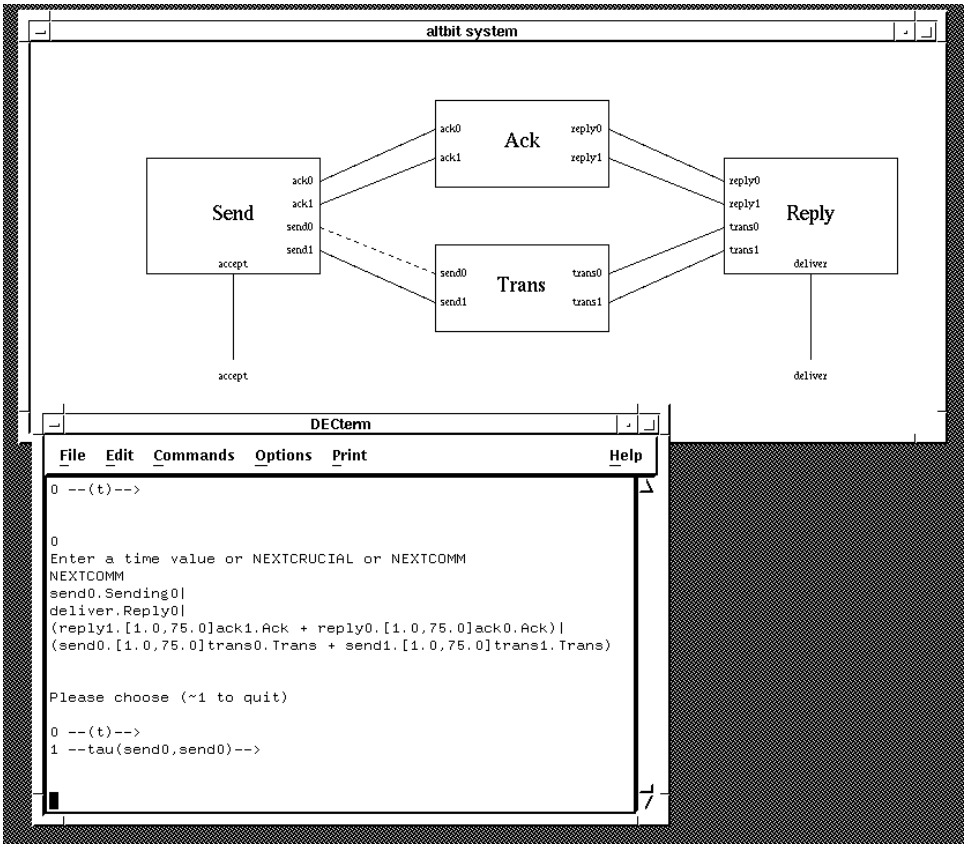

Figure 5.1: Screen Dump of a Simulation of the Alternating Bit Protocol

The non-determinism in the system, in the time bounds and non-deterministic choice, has to be resolved by the simulator somehow. There is a variety of tactics available, which are prompted for when the simulator is started up. Resolution of time bounds can be done by always choosing the minimum value, always choosing the maximum value, choosing a random value, or always prompting the user for a value. Resolution of non-deterministic choice is always achieved by prompting the user.

There is a technical problem associated with ensuring that the semantics of AORTA are accurately followed in the simulation. In the formal definition of AORTA, it was proved that there was a direct correspondence between the *Age* function, which is used to find the state a process reaches after a given amount of time has passed, and the time transition semantics of sequential expressions (theorem 1). This was necessary because the semantics of system expressions depends on the *Age* function, and this function is assumed to represent the aging of processes, as used in the following rule:

$$\frac{\forall i \in I . S_i \xrightarrow{(t)} S_i'}{\prod_{i\in I} S_i < K > \xrightarrow{(t)} \prod_{i\in I} S_i' < K >} \; \forall t' < t . \prod_{i\in I} Age(S_i, t') < K > \stackrel{\tau}{\not\longrightarrow}$$

Although we can be reassured about the meaning of this rule, there does remain a problem in checking whether the side condition holds as there must be no possibility of communication in the time interval $[0, t')$. For a dense time domain this will always give an infinite number of time values to check, and although it is relatively easy to prove by hand that a process cannot communicate for a time interval, the proof could not easily be automated. For the simulation to work efficiently, and yet follow the semantics, it is necessary to reduce the size of the set from which t' must be taken. This can be done by considering only certain crucial points in the evolution of a sequential process; it is these crucial points which are referred to in the `NEXTCRUCIAL` transition of the simulator. The new side-condition is then written

$$\forall t' \in (\bigcup_{i\in I} crucial(S_i) \cap [0, t)) . \prod_{i\in I} Age(S_i, t') < K > \stackrel{\tau}{\not\longrightarrow}$$

but the important definition is of the set of crucial points of a (regular) sequential process:

$$\begin{aligned} crucial(\sum_{i\in I} a_i . S_i) &= \{0\} \\ crucial([t]S) &= \{0\} \cup \{t' + t \mid t' \in crucial(S)\} \\ crucial(\sum_{i\in I} a_i . S_i \rhd^t S) &= \{0\} \cup \{t' + t \mid t' \in crucial(S)\} \end{aligned}$$

The crucial points of a sequential process are those at which the actions available may change, and so no new communication will become available between crucial points. To show that this side condition is equivalent to the previously stated (and more intuitively obvious) one, we need the following result:

Theorem 6 *for all regular* S_i *and all* $t > 0$

$$(\forall t' < t . \prod_{i \in I} Age(S_i, t') < K > \stackrel{\tau}{\not\rightarrow}) \;\Leftrightarrow$$

$$(\forall t' \in (\bigcup_{i \in I} crucial(S_i) \cap [0, t)\,) . \prod_{i \in I} Age(S_i, t') < K > \stackrel{\tau}{\not\rightarrow})$$

The proof of this theorem can be found in appendix A; its application allows the simulator to check only a finite number of points for communication, whilst retaining semantic integrity.

This style of simulation allows very detailed investigation into the behaviour of a system, with precise control over how time progresses, and detailed 'inside information'. However, it may only be used by people who both understand the meaning of the transitions (i.e. have a basic grasp of the semantics), and how the design is meant to work in detail. This approach does not lend itself well to evaluation of a design by an inexpert user, so an alternative type of simulation is offered, which requires very little knowledge of what the semantics mean, or how the design works internally.

5.2.2 Event Driven Simulation

In this style of simulation, the user has less direct control on the system under simulation, but is better able to get an intuitive feel for how the system behaves. Passage of time in the system under simulation is modelled by passage of time during the simulation itself, and external actions of the system are modelled as screen and keyboard events. The simulation can almost be thought of as a prototype implementation; the main difference being that the external connections are implemented differently, and no use is made (at present) of code given in annotations.

Concurrent ML [92] is used to provide communication primitives, as communication channels, choices and time-outs can be implemented directly using CML language constructs. Each AORTA process is spawned as a separate CML process, and runs a copy of an interpreter for the behaviour of sequential expressions. Time progress can be altered by changing the function which maps from AORTA time to UNIX system time, and time-stamps are used to implement both time-outs and

computation delays. An assumption is made throughout the simulator that the amount of time taken to set up computations delays and time-outs is negligible, so that the real-time clock can be used to model the passage of AORTA time accurately. This assumption is very similar to the assumptions used in synchronous language implementations [8].

Although there is a prototype event driven simulator, it is not as well developed as the menu driven simulator, and there are several problems to overcome before it can be used seriously as a simulation tool. One of the major problems lies in the resolution of data dependent choice: the simulation cannot easily be stopped to ask the user how the choice should be resolved (this is against the spirit of this type of simulation anyway), so some automatic way of deciding must be given. This could be random, but that would be very confusing, particularly as data dependent choice is mainly used to check for conditions which might adversely affect the performance of the system (e.g. a dangerously high temperature, or the brake being applied). A more satisfactory choice is to build in a data element to the language, which can be used to make informed decisions automatically. Chapter 8 discusses how data can be included into the formal model of the system.

Simulation was earlier compared with high-level testing as a means for validating the correctness of a design. However, in chapter 2 it was argued that testing real-time systems is in many ways unsatisfactory, and that for a level of confidence of correctness, such as may be required for safety-critical systems, a more formal approach is required. One formal verification technique, which has the advantage that it can be fully automated, is model-checking; this is the subject of the next section.

5.3 Verification by Model-checking

Model-checking is a technique, first introduced by Clarke *et al* [25], to verify temporal properties of finite state systems automatically. Originally, the work was applied to untimed systems, and dealt only with ordering of events, but was later applied to timed systems (see section 2.4.1). Work by Alur *et al* [1] showed how to perform model-checking in a dense time model, by constructing a *region graph*. Within the region graph, nodes are associated with intervals of time over which the logical sentence does not vary. As the region graph is finite, automated procedures may be applied to exhaustively examine the system. Model-checking is widely studied, and it is beyond the scope of this thesis to contribute anything to this area. It is

possible, however, to translate AORTA expressions into *timed graphs* [76], a formalism used by the KRONOS [30, 105] model checker. Using this translation, timed model-checking can be performed on AORTA designs; verification of designs with respect to timed specifications can be performed in this way. A similar translation, from the process algebra ATP, to timed graphs, is given elsewhere [75]. The next section reports on the translation from AORTA designs to timed graphs, as given by Kendall [19].

5.3.1 Translation to Timed Graphs

A timed graph is an automaton which is extended with a finite set of *clocks* where a clock is a real-valued variable which records elapsed time. Clocks advance uniformly with time or can be reset to zero. For a finite set of clocks C and rationals $\mathbf{Q}$, the set of *clock formulae* $\mathcal{F}(C)$ is

$$\mathcal{F}(C) = \{c \geq r \mid c \in C, r \in \mathbf{Q}\}$$

A clock *valuation* $v \in \mathbf{R}^C$ is a function which assigns to each clock $c \in C$ a value $v(c) \in \mathbf{R}$. Times can be added to valuations: $v + t$ stands for the valuation v' such that $v'(c) = v(c) + t$ for all $c \in C$. Clocks may also be reset: $v[C' := 0]$ stands for the valuation v' such that $v'(c) = 0$ for $c \in C'$ and $v'(c) = v(c)$ otherwise. The evaluation of clock formula f given clock valuation v is written $f(v)$ and a valuation v is said to satisfy a clock formula $c \geq r$ if $v(c) \geq r$.

Definition 7 *A timed graph is a tuple,* $(N, n^0, C, E, \mathbf{tcp})$*, where*

- N *is a finite set of nodes*
- n^0 *is the initial node*
- C *is a finite set of clocks*
- $E \subseteq N \times Label \times \mathcal{F}(C) \times 2^C \times N$ *is a finite set of edges representing transitions. Each transition* $(n, l, f, C', n') \in E$ *consists of a source location* n *and a target location* $n' \in N$*, a label* l*, a clock formula* f *and a set of clocks* $C' \subseteq C$*.*
- $\mathbf{tcp}: N \to \mathbf{R}^C \to \mathbf{R} \to \mathbf{Bool}$ *is a predicate which determines for each location* n*, clock valuation* v *and time value* t *whether the system can remain at location* n *while time is allowed to progress by an amount* t*.*

A timed graph gives rise to a labelled timed transition system, $(S, s^0, \longrightarrow)$ where

- $S = N \times \mathbf{R}^C$ is the set of states
- $s^0 = (n^0, v[C := 0])$ is the initial state, and
- the transition relation $\longrightarrow$ is given by the rules

$$\boxed{\textbf{Action}}\ \frac{(n, a, f, C', n') \in E \wedge f(v)}{(n, v) \xrightarrow{a} (n', v[C' := 0])} \qquad \boxed{\textbf{Time}}\ \frac{\mathbf{tcp}(n)(v)(t)}{(n, v) \xrightarrow{(t)} (n, v + t)}$$

The graph is built up from the graphs of the individual sequential processes; the translation depends on the fact that every sequential process can be implemented using a single clock (with c_s written for the clock associated with process S, and the singleton $\{c_s\}$ abbreviated to c_s when the context is clear). The clock associated with a sequential process is reset on every transition and so simply records the time since the process last made a transition. For a sequential process S and its associated clock c_s, the translation to a timed graph is defined inductively on the structure of S. A new distinguished action ϵ is introduced, which is used to label time-out transitions and transitions arising from the resolution of nondeterministic choice. The translation is given by a function $\mathcal{G}[\![\_]\!]$ which takes an AORTA expression to its equivalent timed graph.

Summation In general, a sum has the form $\sum_{i \in I} a_i.S_i$. The translation for this expression covers the cases of the **0** process and also action prefixing, in addition to deterministic choice.

For a gate a, let the lower and upper bounds of the possible communication delay of a be written l_a and u_a, respectively. Then, if for $i \in I$ the graphs of $[l_{a_i}, u_{a_i}]S_i$ are $(N_i, n_i^0, c_s, E_i, \mathbf{tcp}_i)$, then

$$\mathcal{G}[\![\sum_{i \in I} a_i.S_i]\!] = (N \cup \{n^0\}, n^0, c_s, E, \mathbf{tcp})$$

where $N = \bigcup_{i \in I} N_i$, $n^0 \notin N$,

$$E = \bigcup_{i \in I} E_i \cup \{(n^0, a_i, \mathtt{true}, c_s, n_i^0) \mid i \in I\}$$

and $\mathbf{tcp}(n_i) = \mathbf{tcp}_i(n_i)$ for all locations $n_i \in N_i$ and $\mathbf{tcp}(n^0)(v)(t) = \mathtt{true}$ for any clock valuation v and time value t.

Time-out Let $\mathcal{G}[\![S_i]\!] = (N_i, n_i^0, c_s, E_i, \mathbf{tcp}_i)$ for $i \in \{1, 2\}$. Then

$$\mathcal{G}[\![S_1 \rhd_{t_1}^{t_2} S_2]\!] = (N_1 \cup N_2, n_1^0, c_s, E, \mathbf{tcp})$$

where

$$E = E_1 \cup E_2 \cup \{(n_1^0, \epsilon, c_s \geq t_1, c_s, n_2^0)\}$$

and $\mathbf{tcp}(n_i) = \mathbf{tcp}_i(n_i)$ for all locations $n_i \in N_i$ except that $\mathbf{tcp}(n_1^0)(v)(t)$ is $\mathbf{tcp}_1(n_1^0)(v)(t) \wedge v(c_s) + t \leq t_2$.

The case of non-deterministic time-out presented here subsumes deterministic time-out and computation delay in an obvious way.

Non-deterministic choice For $i \in I$ let $\mathcal{G}[\![S_i]\!] = (N_i, n_i^0, c_s, E_i, \mathbf{tcp}_i)$. Then

$$\mathcal{G}[\![\bigoplus_{i \in I} S_i]\!] = (N \cup \{n^0\}, n^0, c_s, E, \mathbf{tcp})$$

where $N = \bigcup_{i \in I} N_i$, $n^0 \notin N$,

$$E = \bigcup_{i \in I} E_i \cup \{(n^0, \epsilon, \mathtt{true}, c_s, n_i^0) \mid i \in I\}$$

and $\mathbf{tcp}(n_i) = \mathbf{tcp}_i(n_i)$ for all locations $n_i \in N_i$ and $\mathbf{tcp}(n^0)(v)(t) = \mathtt{false}$ for any clock valuation v and time value t. In other words the choice must be resolved before time can progress.

Recursion The syntactic restrictions on the use of recursion (by the *regular* function) allow its translation to proceed in a very straightforward manner. When a process name X is encountered in the translation of a sequential expression, its translation is simply the graph associated with X; such an association will exist if X has been encountered before but not otherwise. In the latter case, $\mathcal{G}[\![\mathbf{0}]\!]$ is associated with X, and X is added to a list of names whose graphs are yet to be constructed. Following the first pass of our translation, the graphs for all names in this list are constructed, by translating the right-hand side of the defining equation for the name. The initial node of each graph constructed in this way is identified with the initial node of the graph previously associated with the name. We continue in this way until graphs have been constructed for all names encountered.

Parallel composition In giving the translation for parallel composition the following notational abbreviations are used:

$\vec{N}$	for $N_1 \times N_2 \times \ldots \times N_{\lvert I \rvert}$
$\vec{n}$	for $(n_1, n_2, \ldots, n_{\lvert I \rvert})$
$\vec{n}_{ij}$	for $(n_1, n_2, \ldots n_i, \ldots n_j, \ldots n_{\lvert I \rvert})$
$\vec{n}_{i'j'}$	for $(n_1, n_2, \ldots n_i', \ldots n_j', \ldots n_{\lvert I \rvert})$

where I is some indexing set, with $\{i, j\} \subseteq I$, $i \neq j$.

The translation for an AORTA system expression is given by

$$\mathcal{G}[\![\prod_{i\in I} S_i < K >]\!] = (\vec{N}, \vec{n^0}, \{c_{S_i} \mid i \in I\}, E, \mathbf{tcp})$$

The set of transitions is $E = IC \cup EC \cup TO$, where

$$\begin{aligned} IC &= \{(\vec{n}_{ij}, \tau, \mathtt{true}, \{c_{S_i}, c_{S_j}\}, \vec{n}_{i'j'}) \mid \\ &\quad (n_i, a, \mathtt{true}, c_{S_i}, n_i') \in E_i, (n_j, b, \mathtt{true}, c_{S_j}, n_j') \in E_j, (a, b) \in K\} \quad (5.1) \\ EC &= \{(\vec{n}_i, a, \mathtt{true}, \{c_{S_i}\}, \vec{n}_{i'}) \mid \\ &\quad (n_i, a, \mathtt{true}, c_{S_i}, n_i') \in E_i, (a, \_) \notin K, (\vec{n}_i, \_, \_, \_, \_) \notin IC\} \quad (5.2) \\ TO &= \{(\vec{n}_i, \epsilon, \phi, \{c_{S_i}\}, \vec{n}_{i'}) \mid (n_i, a, \phi, c_{S_i}, n_i') \in E_i\} \quad (5.3) \end{aligned}$$

For any location $\vec{n} \in \vec{N}$, clock valuation v and time value t, $\mathbf{tcp}(\vec{n})(v)(t)$ is $\bigwedge_{i\in I} \mathbf{tcp}_i(n_i)(v)(t)$ except that for any location $\vec{n}$ such that $(\vec{n}, \_, \_, \_, \_) \in IC$ we require that $\mathbf{tcp}(\vec{n})(v)(t)$ is false; in other words, time can progress only when all processes allow it but even then, not at any location which has the capacity for internal communication, so enforcing the maximal progress principle.

The sets (5.1) – (5.3) give the transitions for internal communication, external communication and time-outs / nondeterministic choice, respectively. Notice that an external communication is allowed only in a state where no internal communication is possible. This simple priority mechanism ensures the desirable implementation property that a component cannot become swamped by communication with its environment.

The translation of computation delays is not described here, but is defined to be the same as that for time-outs with no actions:

$$\mathcal{G}[\![[t_1, t_2]S]\!] \triangleq \mathcal{G}[\![\mathbf{0} \triangleright_{t_1}^{t_2} S]\!]$$

Similarly, deterministic forms of time-out and delay are interpreted as special cases of the nondeterministic form of time-out:

$$\begin{aligned} \mathcal{G}[\![S_1 \triangleright^t S_2]\!] &\triangleq \mathcal{G}[\![S_1 \triangleright_t^t S_2]\!] \\ \mathcal{G}[\![[t]S]\!] &\triangleq \mathcal{G}[\![\mathbf{0} \triangleright_t^t S]\!] \end{aligned}$$

Figure 5.2 shows some simple examples and figure 5.3 shows the graph constructed for the `Send` process in the alternating bit protocol, which exhibits a pleasing symmetry, reflecting that found in the process description.

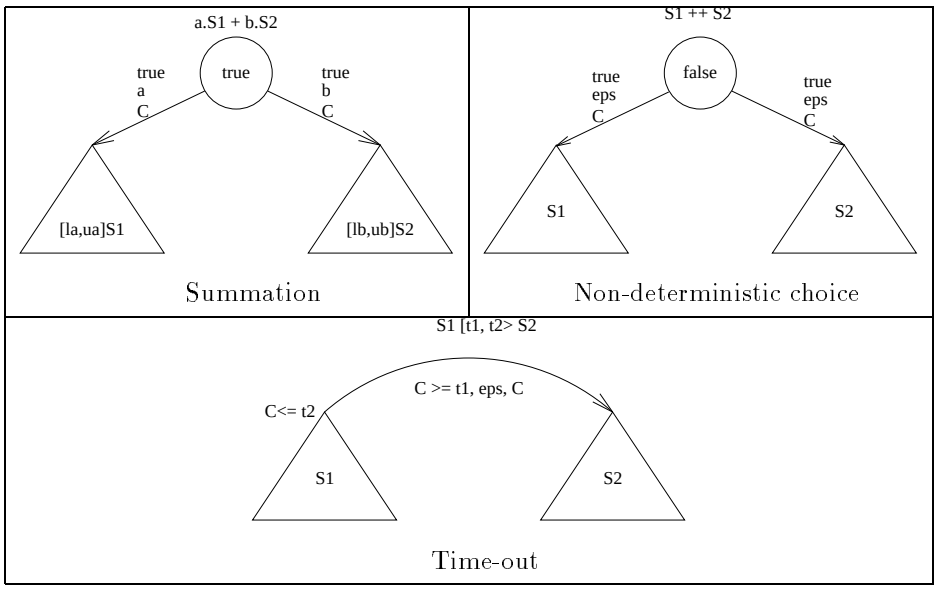

Figure 5.2: Simple timed graph constructions

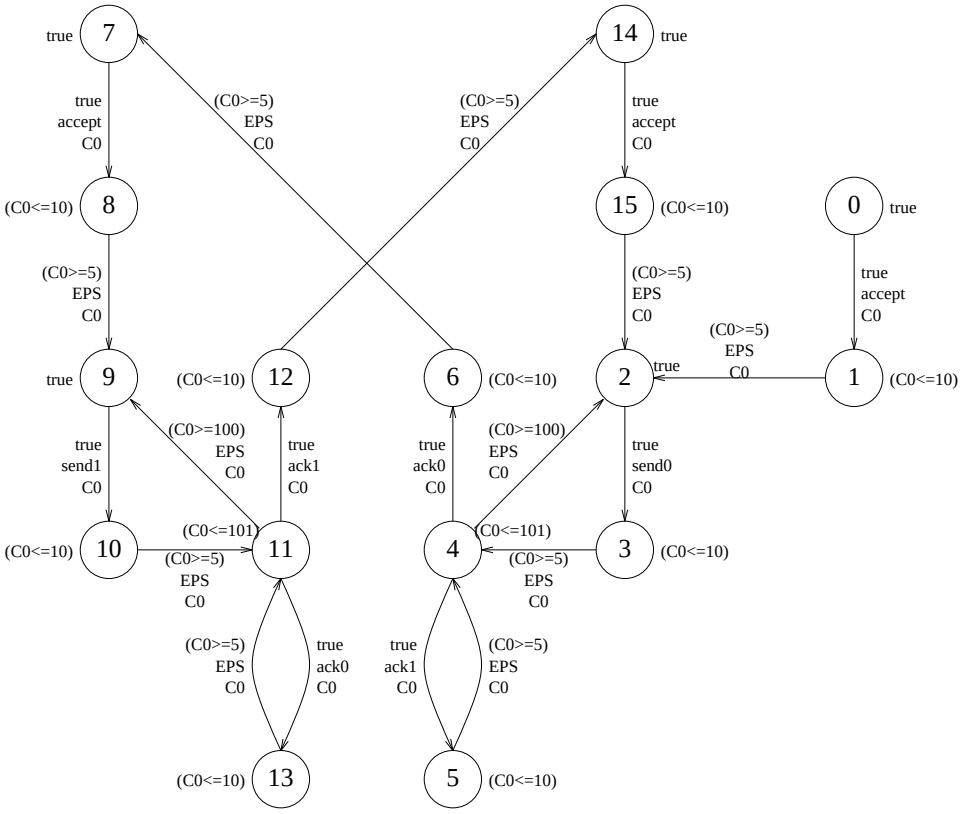

Figure 5.3: The timed graph of the `Send` process

5.3.2 Region Graphs and Model-checking

A timed graph state consists of a node $s \in S$ and a clock valuation for each $c \in C$. Unfortunately, this state space is uncountably infinite, so automatic model-checking cannot be done directly on the timed graph. In order to make the state space finite, an equivalence relation on states must be found, such that the quotient graph with respect to this relation is finite, but that behavioural properties are preserved by the relation. One cause of the infinity of states is that clock valuations can be arbitrarily large, even for discrete time domains. This can be overcome by equating states which have very large valuations for a particular clock: if two values of the clock exceed the largest value with which they can be compared in the enabling function, then they will not give different behaviour. Using the relation, the state space becomes finite for discrete time domains, but the state space is still uncountably infinite for dense time domains. Secondly, in a dense time domain, even a bounded interval contains uncountably infinitely many points, so more work needs to be done to make the state space of a timed graph with a dense time domain finite. The approach suggested by Alur *et al* [1] is based on the use of a logic (TCTL) in which time constraints can only take integer form. The formulae of TCTL, based on a finite set of atomic propositions P, are given by the syntax

$$\phi ::= p \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \phi_1 \exists\mathcal{U}_{\#n} \phi_2 \mid \phi_1 \forall\mathcal{U}_{\#n} \phi_2$$

where $p \in P$, n is a natural number and $\#$ is one of the relational operators $<$, $\leq$, $=$, $\geq$, or $>$. Intuitively, $\phi_1 \exists\mathcal{U}_{\#n} \phi_2$ means that there exists a sequence with a finite prefix such that ϕ_2 is satisfied by the last state at time t where $t \# n$ and ϕ_1 is satisfied continuously until then. $\phi_1 \forall\mathcal{U}_{\#n} \phi_2$ means that for every sequence this property holds. A number of abbreviations are commonly used: $\forall\Diamond_{\#n}\phi$ for $\texttt{true}\forall\mathcal{U}_{\#n}\phi$, $\exists\Diamond_{\#n}\phi$ for $\texttt{true}\exists\mathcal{U}_{\#n}\phi$, $\exists\Box_{\#n}\phi$ for $\neg\forall\Diamond_{\#n}\neg\phi$, and $\forall\Box_{\#n}\phi$ for $\neg\exists\Diamond_{\#n}\neg\phi$. The formal semantics of interpretation of TCTL formulae over sequences of states of timed graphs is given in [1].

The way in which the size of the state space is reduced to make model-checking with TCTL formulae feasible, is to construct *regions* of clock valuations. Within each region, all valuations agree on the integer parts of all clocks, and also on the ordering of the fractional parts of the clocks. As formulae may only use integer constraints, all states within a region agree on all clock comparisons, and the conditions for leaving the region are affected only by the ordering of the clocks. Hence all TCTL formulae agree on all states of the region, so the quotient space (known as the *region graph*) preserves TCTL properties, and can be checked in finite time. TCTL

is expressive enough to express most system properties of interest. For example, a bounded response property can be easily stated,

$$\forall\Box(\texttt{stimulus} \Rightarrow \forall\Diamond_{\leq 5}\text{response})$$

which captures the requirement that after any occurrence of `stimulus`, a `response` will always happen within 5 time units. Other useful properties such as bounded invariance, bounded inevitability, and self-stabilization can be expressed equally easily. Once the state space has been reduced to a finite size, the actual model-checking algorithm used is fairly simple: see the paper [1] for details. A more sophisticated model-checking technique is employed by the KRONOS tool [105], although timed graphs are still the basis for the formalism. The use of KRONOS to check properties of the alternating bit protocol described in AORTA, is described elsewhere [19].

5.4 Conclusion

Simulation and model-checking can both be used for evaluating the correctness of AORTA designs. Simulation of designs for validation purposes allows inexpert users to assess the system with respect to their requirements, and expert users to investigate the behaviour in detail before attempting verification. For high levels of confidence of correctness, a formal verification may be required, and model-checking can be used to calculate automatically whether a design satisfies a formal specification expressed in a timed temporal logic.

There is much scope for more work on the simulators, particularly on the event based simulator. The menu driven simulator has, however, proved to be a useful tool in investigating and debugging AORTA designs before verification of the design, or implementation. A proof of correctness of the translation of AORTA designs into timed automata is the subject of current research. Model-checking of timed process algebras, however is not particularly novel [75, 30], whereas implementation of timed process algebras is much less widely studied. This production of implementations from AORTA designs, and the corresponding timing analysis, are the subjects of chapters 6 and 7.

Chapter 6

Implementation Techniques

6.1 Introduction

The point of the last two chapters was to demonstrate that AORTA is expressive enough to design real systems, and that these designs can be validated and verified; the point of this chapter is to show that the restrictions which AORTA imposes on the kind of designs which can be written are enough to allow direct implementation. One of the advantages of the process abstraction used in process algebras is that the same notion (i.e. a process) can be used to represent software processes which share a single processor, distributed software processes, hardware processes, and environmental behaviour. For the methods described in this chapter only software processes sharing a single processor as considered, but the use of distributed processing and direct hardware implementation are areas for future work. Possible environmental behaviour does not need to be implemented for the final system, although it may need to be simulated (see chapter 5).

AORTA can be thought of as a virtual machine for real-time systems, allowing standardised, verified implementation techniques to be used without too much detail having to be known to the designer. Clearly it is important that such standard techniques are correct, or errors may be built into every system that is constructed. Chapter 7 deals with the analysis required to justify the correctness of the implementation techniques presented here. As previously mentioned, data is not included in the standard semantic model for AORTA, so aspects such as computation and data communication are not formally defined. However, computation and data communication are important to the implementation (even if they cannot necessarily be verified formally), so the information required is included within the AORTA

designs as *annotations*, which are essentially comments as far as the formal semantics is concerned. A complete working system can be generated automatically from an annotated AORTA design.

Within this chapter, different aspects of the implementation are considered in separate sections: section 6.2 describes the generation of code for each of the processes, section 6.3 deals with process scheduling on a single processor (multitasking), and section 6.4 explains how inter-process communication is implemented. As usual, the last section of the chapter (section 6.5) reviews the material and presents some conclusions. Some of the work on code generation has been published [19], whilst the use of the kernel and the general approach to implementation is described in several papers [13, 14, 16].

6.2 Implementing Processes: Code Generation and Annotations

In order to implement a parallel composition of processes, each process is implemented as a program with a single flow of control, and these programs are multitasked on a single processor. This section deals with the generation of the code for a process from an annotated AORTA description of the process. Currently the target language is C, largely for pragmatic reasons such as the availability of cross-compilers and code timing tools [81, 80]. However, any standard procedural language which is amenable to timing analysis could be used; C is used as a portable assembler. The first part of this section deals with the C code generation for the skeleton of the process, and then details of how this is fleshed out are given by the second part, which describes in detail the use of annotations.

6.2.1 Process skeleton generation

At the heart of the process code generation is a graph of the possible execution paths through the process. The graph is formed from the abstract syntax tree, with a node in the code graph for each node of the syntax tree. Each equational definition associated with the process gives rise to its own tree. Within the generated C, each node gives rise to a labelled segment of code, with edges in the graph (which correspond to transitions of the operational semantics) being implemented as `goto` statements. Some transitions, such as those for the completion of a piece of computation or for recursion, are unconditional, so the `goto` statement is placed directly

after the relevant piece of code. Other transitions, such as those for communication choice, time-out and data-dependent choice, may depend on the previous behaviour, so branching may be required.

For communication choice, the branch taken depends on which communication occurs first; this information is only available from the communication handler (in this case the kernel), and is accessed via a kernel call. The same call is used to inform the communication handler of the gates which wish to communicate. Both rôles are carried out by the `communicate` function, which takes as its arguments the process identifier (an integer), an array containing identifiers of the gates to be offered in choice, terminated with a zero, and an array which is used for passing values in and out during communication (see section 6.2.2). This function sets up the communication and waits for a communication event to occur. Once the event has occurred, `communicate` returns an index corresponding to the gate on which communication has taken place: 1 for the first element of the array, 2 for the second and so on. Based on this, the generated code for a communication choice has a `switch` statement relying on the returned value of a call to `communicate`. The code for the choice

```
Computer = one.[0.4,0.5]Computer + two.[1.2,1.4]Computer
```

used in the mouse button example 3.3.1 is

```
Computer_1:
gatenames[0] = GATEone;
gatenames[1] = GATEtwo;
gatenames[2] = 0;

switch (communicate(PROCComputer,gatenames,gatevalues)){
case 1: goto Computer_2;
case 2: goto Computer_4;
}
```

Here the values `GATEone`, `GATEtwo` and `PROCComputer` are replaced with integers at compile time by using a `#define` directive.

Time-outs are an extension of communication choice, and are implemented as such: a kernel call `communicatet` is provided, which has the same arguments as `communicate` plus a time-out value. Again, an index is returned corresponding to which event has occurred, with a 0 value returned in the case of a time-out event. The generated code is also very similar, so that the code for

```
(click.double.Mouse)[0.245,0.255>single.Mouse
```

is

```
Mouse_2:
gatenames[0] = GATEclick;
gatenames[1] = 0;

switch (communicatet(PROCMouse,245,gatenames,gatevalues)){
case 0: goto Mouse_5;
case 1: goto Mouse_3;
}
```

Data-dependent choices are generated simply as `switch` statements over the annotated conditions, and are described in more detail in section 6.2.2.

Code for each node of the syntax tree is generated in this way, with `goto` commands to implement the transitions. The transitions required for recursion are again simple `goto` commands to the top node of the appropriate syntax tree (which is labelled with a `_1` suffix). This will lead to a piece of generated code like this:

```
Mouse_6:
goto Mouse_1;
```

which will itself have been reached by a `goto` command, so that the code will contain various paths including 'goto a goto', which would be considered very poor programming if produced by a human. There are two points to be made here, firstly that style considerations and ease of debugging are not relevant to generated code, and secondly that there is unlikely to be any loss of efficiency because most compilers will optimise out any such situations. It would be possible (and indeed quite easy) to remove these 'goto goto' paths during the code generation, but for the reasons just given it is considered unnecessary at the moment. To complete the skeleton of the code, the initial state must be noted (the generated code for the nodes could appear in any order), and an initial `goto` command added.

6.2.2 Defining annotations

As far as the formal semantics of AORTA is concerned, annotations are merely comments, but in practice they are very important in defining how the system is to be implemented in various ways:

- in computation delays, annotations are used to define what computation should take place by giving a piece of code to be executed
- in communication, annotations are used to define what values are to be passed, and where passed values are to be stored
- in nondeterministic choices, annotations are used to define the basis on which a choice between two branches is to be made
- in the definition of each process, annotations are used to declare variables that are to be used within the computations, and to define the functions which may be used for external I/O
- in external I/O, annotations are used to define how the I/O should be handled, by giving a function to act as an I/O driver

All annotations are introduced into the text in the same way, by including the text within `@ ... @`, but the meaning of the annotation changes with its context, in order to give the different kinds of definitions just described. The last item (external I/O) is a system level concern, and is dealt with in section 6.4, but the others are used to put flesh on the skeleton of code which is automatically generated, and in considered in turn in the following subsections.

Computation delays

The most obvious and straightforward use of annotations is for computation delays. Pure AORTA only defines bounds on how long the computation will take, but in order to generate an implementation, the actual code must be supplied. This is achieved by giving an annotation immediately after the time bounds of the delay, but before the square bracket is closed. For example, a process which is to wait for communication on gate `a` before printing 'Wow', would be defined like this:

```
Wowee = a.[0.1,0,2 @printf("Wow");@] Wowee
```

Any fragment of C code may be included here: it may take up more than one line, and may use preprocessor commands such as `#include <filename>`. It is envisaged that this will be the usual way of including larger pieces of code. The derivation of the time bounds for the computation depends on the time the code requires to complete and on the scheduling mechanism used, and is discussed in more detail in chapter 7. Any code which is given in an annotation is included directly in the generated code at the correct point in the graph.

Value-passing

Communication within AORTA is modelled as pure synchronisation, but value-passing at synchronisation is handled by the implementation. The details of what data is to be passed out (if any), and what is to be done with incoming data (if any), are given by annotations, following the style of CSP [50] and LOTOS [77]. Data values to be passed out are given by annotating the gate name with an exclamation mark and a C integer expression. Input values are stored in the variable whose name is given in an annotation starting with a question mark. For example, a process which reads in a value from gate `a` and then prints it would by given by

```
GetItAndPrintIt = a@?x@ .
       [0.1,0.2 @printf("%d",x);@] GetItAndPrintIt
```

A process which sends out values on gate `out`, starting at 0 and increasing by one could be defined

```
CountInit = [0.01 @x=0;@] Count
Count = out@!x@ . [0.1,0.3 @x++;@] Count
```

Values are handled in the implementation by using the `gatevalues` argument to the `communicate` function. Any data which is to be passed out of a process is put in the appropriate position is the `gatevalues` array, and any data which is input to a process is passed back via the `gatevalues` array. For example, the code generated for the communication of the `GetItAndPrintIt` process is

```
GetItAndPrintIt_1:
gatenames[0] = GATEa;
gatenames[1] = 0;

switch (communicate(PROCGetItAndPrintIt,gatenames,gatevalues)){
case 1: x = gatevalues[0];
       goto GetItAndPrintIt_2;
}
```

and the code for the communication in `Count` is

```
Count_1:
gatenames[0] = GATEout;
gatenames[1] = 0;
```

```
gatevalues[0] = x;
switch (communicate(PROCCountInit,gatenames,gatevalues)){
case 1: goto Count_2;
}
```

Similar code is generated for a time-out communication with data.

Data-dependent choice

Nondeterministic choice (`++`) is used in AORTA to represent data dependent choices, but as no model of data is included in pure AORTA, the way of deciding which branch to take must be provided by an annotation. The annotation must occur immediately after the `++` symbol, and must give a C logical expression (evaluating to either 0 or 1), such as `x==3`. If the expression evaluates to 0 (false) then the branch before the `++` is taken, and if it evaluates to 1 (true) the branch after the `++` is taken. For example, if the the process `P1` is defined as

```
P1 = a@?x@.
      (P11
       ++@x==3@
       [0.1,0.2 @printf("Wow");@] P11)
```

data is read on gate `a`, and if it is not equal to 3 then control flow passes directly to process `P11`; if it is equal to three then 'Wow' is printed before passing on to `P11`. The code generated for the choice is

```
P1_2:
switch (x==3) {
case 0: goto P1_3;
case 1: goto P1_4;
}
```

Variable declaration and function definition

In order that the computation delay code can be compiled correctly, all variables that are used within the computation must be declared. However, as different delays may operate on the same variables, and some variables may be needed to store values for communication, these variables need to be associated with a particular process rather than a particular computation. To allow for declaration of variables, each process may be annotated (within the system definition) with the variable

declarations for that process. For the process `P1` of the previous subsection, the variable `x` needs to be declared, and this is achieved by

```
(P1
@int x;
@ |
P2 | ... )
< ... >
```

Note that each process has its own set of variables, so the same variable name can be used independently in different processes.

Similarly, the I/O functions used by external connections need to be defined somewhere — in this case the whole system may be annotated by a set of function definitions in between the list of processes to be composed and the connection set. Functions to be called from computations may also be defined here.

Giving a fully annotated AORTA design gives rise to a set of C functions associated with the processes of the system. These functions can be compiled separately, but in order to execute them in parallel on a single processor, a multitasking mechanism must be found, which is the topic of the next section.

6.3 Implementing Parallelism: Multitasking

There are three main reasons for wishing to introduce concurrency into a computer system:

1. The performance required is better than that which can be achieved economically on a single processor, so more than one processor must be used

2. The computer system has to be distributed geographically (e.g. in a telecommunication system), so more than one processor must be used

3. The design of the program is too complex if expressed linearly, so more than one process must be used

Any combination of these reasons for wishing to introduce concurrency may be true of a given system, but for the purposes of this section we consider only the third reason. In this case, parallelism can be provided by multitasking, where the resources of a single processor can be shared by more than one process; clearly this solution is not appropriate if either of the first two conditions apply.

Process scheduling has been (and still is) the subject of much research, with many different ways of distributing the resources being suggested. A review of scheduling techniques can be found elsewhere [21]. The building block of all scheduling mechanisms is the *task switch*, which stops one process from executing, saves the context of that process, loads in a new context and starts a new process running. Aside from the issue of what is meant by 'context' (this may change depending on the nature of the switch), which is not of great relevance to this discussion, the important features by which a scheduling mechanism can be classified are

- the times at which task switches take place
- the method for choosing which process is to have the processing resource next

If there are no crucial time constraints, it is often simplest to use a *voluntary task switch* (or *non preemptive task switch*) mechanism, where a process releases the processor to other processes at a point which is convenient to itself. However, this may mean that one process has control of the processor for a long period of time when another (more critical) process is in need of the resource. In such a case it is often necessary to introduce a *preemptive* task switch, which forces a process to release the processor, to guarantee a better distribution of the processor's time. The length of time between preemptive task switches is often referred to as the *time-slice*; the implementations described here use a fixed time-slice preemptive scheduling mechanism, so task switches occur at a constant rate.

There are various techniques available for the selection of the process which is to follow after the task switch. One of the simplest techniques is known as *round robin* or *cyclic executive* scheduling, where the processes repeatedly follow one after another in a predefined order. This is a simple form of *static schedule*, where the ordering of processes is calculated off-line, and is not affected by conditions at run time. Other, more complex schedules may be calculated to satisfy various deadlines and precedence constraints [60, 97]. *Dynamic scheduling* allows run time information to affect the choice of process to be executed. Perhaps the simplest form of dynamic scheduling is *fixed priority scheduling*, which has a set of priorities assigned to tasks, and the next task is chosen as the currently runnable process which has the highest priority. A well established body of theory, known as *rate-monotonic analysis* [4] provides techniques for demonstrating that deadlines will be met within a fixed priority preemptive scheduling environment. More advanced dynamic scheduling techniques use a dynamic priority system, which can be based on the *earliest deadline first* principle, or on *least laxity first*.

A variety of techniques can be used for scheduling AORTA implementations, but the easiest and most developed is preemptive fixed time-slice round-robin scheduling. A dedicated kernel has been written to support this method [13] and the analysis techniques, described in chapter 7, are fairly simple, and not computationally intensive. Priority based scheduling can be used, but this relies at the moment on the use of VxWorks, a proprietary real-time kernel which allows the use of fixed priorities. Analysis techniques for priority based scheduling are also presented in chapter 7, but these use state space enumeration methods which are computationally expensive.

6.4 Implementing Communication: I/O and the Kernel

Apart from providing a platform for execution of the processes, a kernel must provide facilities for inter-process communication, and for communication with the environment. To implement communication within AORTA in accordance with the semantics, when a pair of gates in the given connection set is ready to communicate it must do so within a certain amount of time (the communication delay). If any other gates are offered in choice, they must be disabled so that only one gate in the choice may communicate. As any process may be preempted at any point, careful attention has to be paid to keeping a consistent and complete set of information available to the kernel and the processes, to ensure that the mechanism is sound. To this must also be added an implementation of time-outs, a mechanism for communicating with the environment, and a way of passing values during communication.

To describe the algorithm used to implement communication, it is necessary to introduce all of the various flags and pointers associated with each process and gate of the system. Given an indexing set I for the processes, and a set J for the gates, i and j range over I and J respectively. For each gate j there are two flags: $gate_j.ready$ and $gate_j.has\_comm$, which correspond to the readiness of the gate to communicate (i.e. the process to which it belongs has offered it by itself or in a choice), and the successful completion of communication on that gate. There is also an entry $choice\_addr_j$ which contains a pointer to a list of the gates in choice with j at the current time, and a choice list $choice\_list_i$ for each process. Finally, each process has a flag set_up_i which is used to indicate when that process wishes to

variable	*set*		*reset*	
name	*what*	*when*	*what*	*when*
set_up_i	`communicate`	wishes to comm	kernel	done set up
$choice\_list_i$	`communicate`	before set_up_i	N/A	N/A
$gate_j.ready$	kernel	in set up	kernel	after comm
$gate_j.has\_comm$	kernel	after comm	`communicate`	after noting comm

Table 6.1: Access to Communication Variables in the Kernel

offer a communication. Figure 6.1 shows how these flags are changed when a simple choice between two gates is offered.

There are two parts to the algorithm for communication, which effectively run concurrently: the code executed by the kernel on every task switch, and the code executed by a process when it calls the `communicate` function from the generated code. Firstly, the `communicate` function stores the gate identifiers in $choice\_list_i$ (for process i), and then sets the set_up_i flag. This corresponds to figure 6.1a. During its task switch, the kernel checks the set_up_i flag for every process, and if it is set, it goes through the corresponding choice list, setting the $gate_j.ready$ flag for every gate it finds there, as a signal to itself that gate j wishes to communicate. This is illustrated by figure 6.1b. After having checked for setups, the kernel looks through the connection set (i.e. the set of pairs of gates that are connected in the design). If it finds a pair in which both gates are ready to communicate (have their $gate_j.ready$ flags set), it resets all of the $gate_j.ready$ flags in the corresponding choice lists (as pointed to by $gate_j.choice\_addr$), and sets the $gate_j.has\_comm$ flag for each of the communicating gates, as illustrated by figure 6.1c. The setting of the $gate_j.has\_comm$ flags is the signal back to the processes which initiated the communication that it has been successfully completed, and where there is a choice, which of the gates was successful, and it resets the $gate_j.has\_comm$ flag before continuing. The `communicate` function checks the has_comm flags, and when it finds that one is set, it resets it and returns the index of the gate that has communicated. Table 6.1 summarises how the various variables are accessed, and under what conditions they are set and reset, while figures 6.2 and 6.3 give the pseudo-code forms of the algorithms used by the `communicate` function and the kernel task switch respectively. Figure 6.4 gives an example of how the kernel works, by showing the order of flags being set and reset for a communication between the `Datalogger` process (see section 4.2), which is offering a choice between `download` and `speed`, and the `Convert` process, which is offering the gate `changespeed`. As `Convert.changespeed` and `Datalogger.speed` and linked in the connection set,

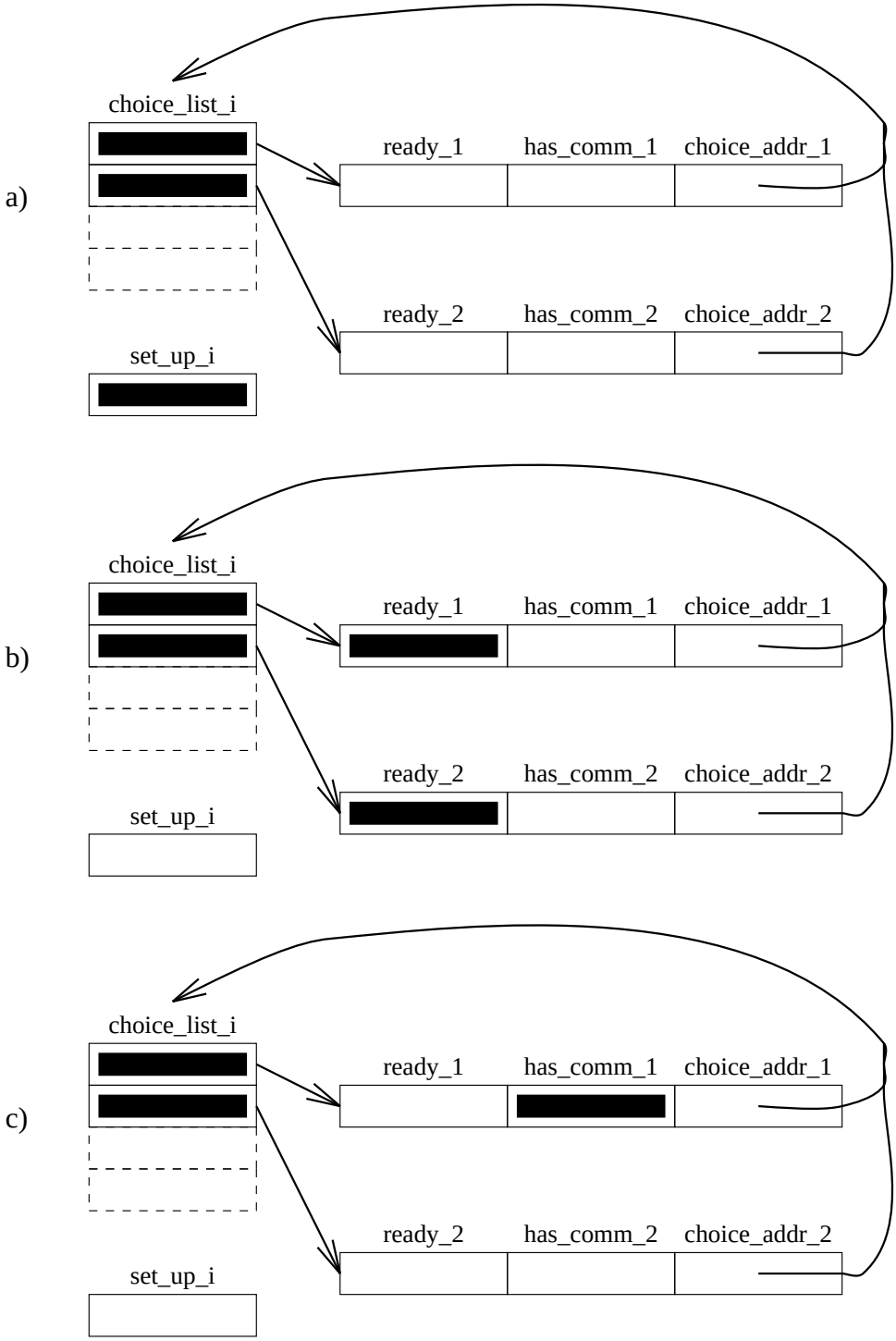

Figure 6.1: An Illustration of the Communication Flags

```
function communicate(i,choices)
for all j in choices do
  copy j into choice_list_i
end do
set_up_i := true
got_comm := false
do while (not got_comm)
  for all j in choice_list_i do
    if gate.has_comm_j = true then
      j_comm := j
      gate.has_comm_j := false
      got_comm := true
    end if
  end do
end do
return j_comm
```

Figure 6.2: Communication Function for Process i

```
for i = 1 to number of processes do
  if set_up_i = true then
    set_up_i := false
    for all j in choice_list_i do
      gate_j.ready := true
    end do
  end if
end do
for all (j_1, j_2) in connection set (* i.e./ j_1 and j_2 are connected *) do
  if gate_j1.ready = true and gate_j2.ready = true then
    for all j in list pointed to by choice_address_j1 do
      gate_j.ready := false
    end do
    for all j in list pointed to by choice_address_j2 do
      gate_j.ready := false
    end do
    gate_j1.has_comm := true
    gate_j2.has_comm := true
  end if
end do
```

Figure 6.3: The Kernel's Communication Test Algorithm

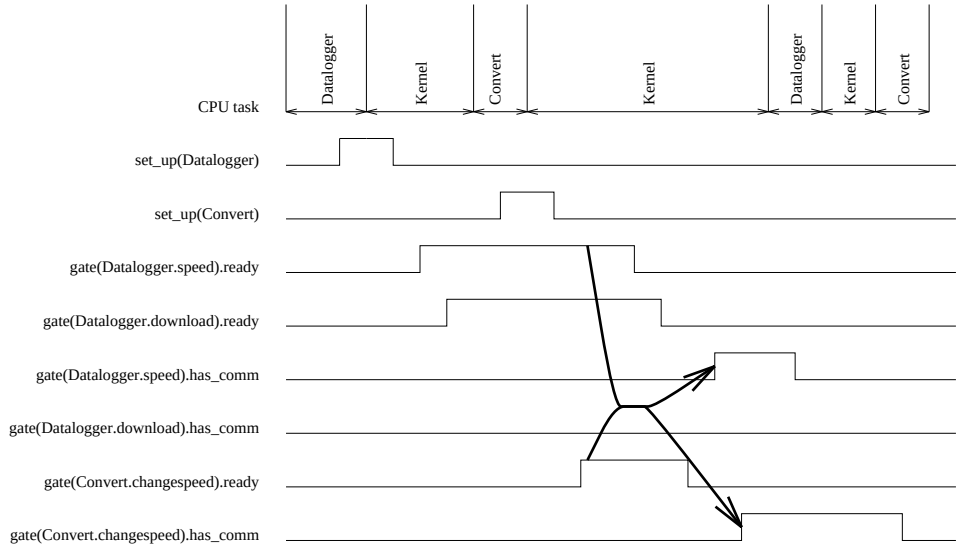

Figure 6.4: Communication Between the *Convert* and *Datalogger* processes (timings not to scale)

they trigger a communication when both are ready, as indicated on the diagram.

The algorithms as presented in figures 6.2 and 6.3 do not describe the way that timeouts and value-passing (as opposed to pure synchronisation) are handled, but they require only a little modification to deal with these features. Timeouts can be considered as choices with an extra possible branch, where instead of a communication event triggering the branch, a timeout event is used instead. By including a timeout in the choice list, the relevant communications are disabled if timeout occurs, and the timeout is disabled if a communication event occurs. A timeout has associated with it a *ready* flag, a *has_comm* flag, and the time at which it should occur. Instead of the kernel checking for a pair of gates to be ready to communicate, it checks to see whether the occurrence time has been passed, and if it has, the choice list is reset and the *has_comm* flag set, as for internal communications.

A value-passing mechanism is provided by associating with each gate storage for a data-in value and a data-out value. The value passed could be the data itself, or a reference to the data. When the kernel finds a pair of gates ready to communicate, it copies the data-out of one process to the data-in of the other and vice versa. As far as the calling routine for the process is concerned, the `gatevalues` parameter values are copied to the relevant data-out slots before the *set_up* flag is set. After communication has occurred the data-in from the gate which has communicated is copied into the relevant entry in the array of values.

The actual implementation of the kernel and communication primitives (i.e. functions for communication and timeout) was written in 68000 assembly language, with the communication primitives callable from C. The code for the whole kernel, including the communication primitives, only occupies about 3 kilobytes of memory, with storage for the variables, flags, and stacks of a small system with four processes and ten connections occupying another 3 kilobytes.

6.4.1 External I/O

In order to allow external I/O, information has to be given as to how the external devices are to accessed. This is done by annotating an `EXTERNAL` connection with the name of a driver function for that device, after the time bounds for that connection. The named function should accept one parameter, pointing to data to be output, and return an integer whose lower byte contains `0xff` if communication has taken place, and `0x00` if not. Input data will be stored in the place pointed to by the function parameter on return if communication has taken place. This function will be called during the task switch if the gate to which it is connected is ready to communicate. Because the time spent in a task switch must be kept low, this does not allow for complicated I/O, such as may be required to perform analogue to digital conversion, or to send long packets of data. To handle such cases, the code generator has been modified so that if an externally connected gate appears in a communication without choice, then the I/O function is called directly, rather than via the kernel's communication mechanism. Apart from this case, the method of communication is transparent to the process, and is resolved only at the system level, when gates are connected either to other gates, or to external I/O functions. Also, because all communication testing takes place during task switching, the same communication mechanism can be used for any fixed time-slice preemptive scheduling mechanism, whether it uses round-robin static scheduling, or a dynamic, priority based scheduling mechanism.

6.5 Conclusion

This chapter has described in some detail how AORTA designs can be implemented. The basic idea is shown in figure 6.5. Code is generated for each of the processes, including calls to kernel routines which deal with communication and time-outs. The kernel also allows the processes to be executed concurrently on a single processor by using multitasking; it needs only to be told how often preemption is to occur,

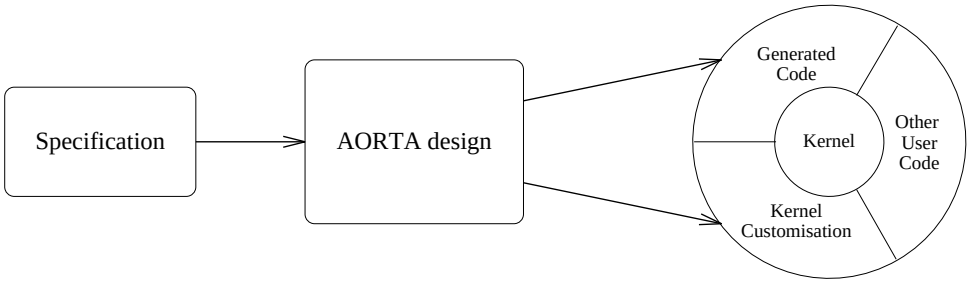

Figure 6.5: Implementation of an AORTA design using code generation and the kernel

the order of execution of processes, and how the gates are to connected (internally and externally). An analysis of times for computation, communication delay and time-outs is given in chapter 7.

The motivation behind this chapter was to demonstrate that AORTA is restrictive enough to allow for direct implementation; I hope that the reader is convinced that this is the case. However, only single processor solutions have been presented, and as mentioned above, concurrency is sometimes introduced specifically because a multiprocessor solution is required. In order to extend possible implementations to include multiprocessor systems, only the communication mechanisms described in section 6.4 need be changed, as the same generated code would be used, and each processor could still execute processes concurrently. Work has also been undertaken on the use of VxWorks, a proprietary real-time kernel, as a platform for execution [31]; 'hooks' can be added to the task switching mechanism to allow communications and time-outs to be monitored, and the `communicate` and `communicatet` functions remain the same.

Chapter 7

Analysis and Verification of Implementations

7.1 Introduction

Having examined the problem of implementing AORTA designs in chapter 6, we now turn our attention to the analysis of these implementations. The main problem which this chapter addresses is that of timing: in an AORTA design, time bounds are placed on computation delay, communication delay and time-out, all of which must be verified. Any analysis of timing in a multitasking system depends crucially on the scheduling mechanism employed, as is demonstrated by the wealth of literature (see [21] for a review). The implementation techniques described in chapter 6 use fixed time-slice preemptive scheduling mechanisms. This kind of scheduling was chosen because it makes the task of predicting communication and time-out delays much easier, and if round-robin scheduling is used then computation delays are also easy to predict. Section 7.2 presents a relatively simple technique for calculating maximum and minimum elapsed times for computation using round-robin scheduling, along with communication and time-out delays. A small example of how to apply the simpler round-robin analysis techniques is given in section 7.3. A more complex technique is required for priority based scheduling — there is a trade-off between the efficiency of the scheduling mechanism and the difficulty of the analysis. An approach which uses the timed graphs of section 5.3.1 to explore the whole state space is outlined in section 7.4. There follows a discussion about other verification issues in section 7.5, and conclusions in section 7.6.

7.2 Timing Analysis of Round-robin Scheduling

Predictability is at a premium in a system which is to be formally verified, and the round-robin scheduler is very predictable, if not the most efficient under light loads. It should be noted, though, that it is under heavy processing loads that prediction becomes most important, and this scheduler has the pleasing property that the more processing required the more efficient it becomes.

There are three aspects of the timing behaviour of an implementation which need to be verified: the amount of processing time each process gets, the time delay in communication, and the bounds which can be placed on time-out occurrence; these shall be dealt with one at a time. Figure 7.1 gives a scheduling diagram for a system with three processes, and introduces the variables used in the analyses. The fundamental variable in the system is the interrupt or scheduling period, which we shall call p, and after each p time units the kernel is activated to a task switch. Each time the kernel is activated (via an interrupt), a process is descheduled; once the kernel has finished its activity the next process is rescheduled. Depending on how much setting up and communication the kernel has to do, the amount of time it takes to perform all its tasks will vary, but can be bounded. We define the upper and lower bounds on time spent in the kernel at each activation to be k_l and k_u respectively, (actual figures for this implementation are given later) and the value for a particular activation of the kernel is denoted by k. The point at which each process starts will depend on how long the scheduler took, but the time between deschedules is constant, and is represented by d_i for the process i. For a system with the processes being scheduled one after another, the values d_i will be the same for all processes, but where one process occurs twice as frequently as others (e.g. 12131213... as in figure 7.1) the d_i may be different, although they will all be integer multiples of p. We shall also refer to the process schedule time, which is the amount of time a process remains scheduled. This depends on how long the kernel takes to execute immediately preceding the process being scheduled, but is bounded by $[p\text{-}k_u, p\text{-}k_l]$.

The easiest way to calculate the amount of real time required for a certain amount of processing time on a process is to consider the amount of time that the process will be in the middle of the processing, but not scheduled. For the minimum execution time we have to consider the best situation, which is where the processing starts at the beginning of a schedule block. In this case, if the amount of processing time required is less than the process schedule time, then the real time

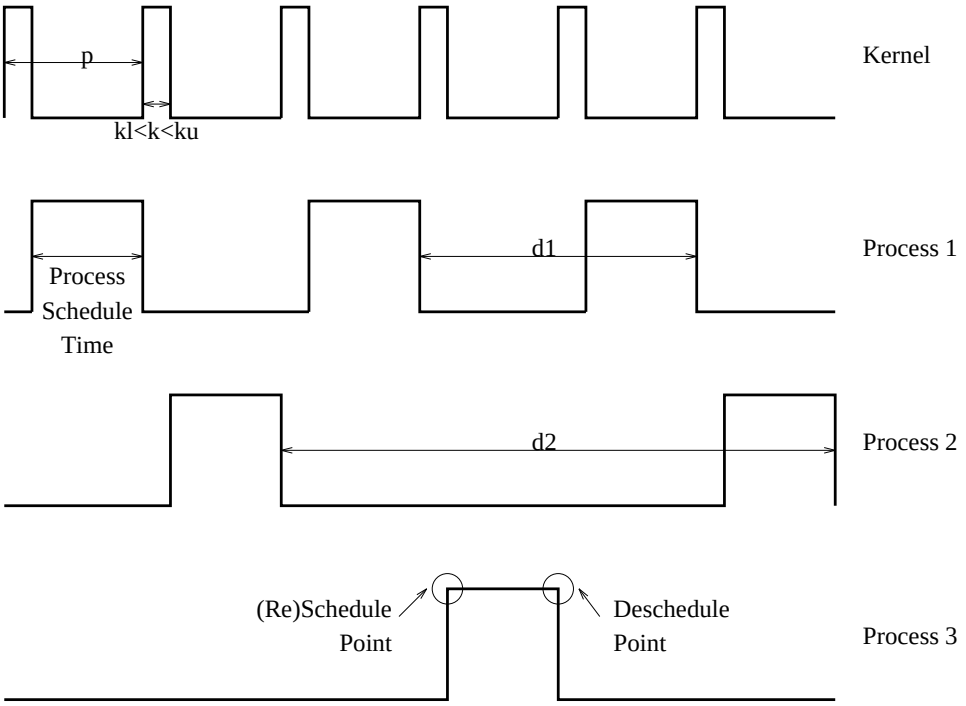

Figure 7.1: Scheduling diagram for three processes

elapsed will be equal to the processing time required, as the processor will have been solely devoted to that processing for its complete duration. If the processing time is greater than the process schedule time then the number of blocks of unscheduled time will be given by $\lfloor r/(p\text{-}k_l) \rfloor$, where r_i is the required processing time and $\lfloor x \rfloor$ is the lower integer part of x. This comes from the assumption that this is the best case, so all kernel execution times will be minimal (k_l), giving a process schedule time of $p\text{-}k_l$. Having calculated the number of blocks of unscheduled time we get the amount of unscheduled time to be

$$\lfloor \tfrac{r_i}{p\text{-}k_l} \rfloor \times (d_i + k_l\text{-}p)$$

as each unscheduled block will last for $(d_i\text{-}(p\text{-}k_l))$. This gives a total elapsed time of

$$r_i + \lfloor \tfrac{r_i}{p\text{-}k_l} \rfloor \times (d_i\text{-}(p\text{-}k_l))$$

as the minimum for a required amount of processing r_i. The worst case (which is usually of more interest) is calculated similarly, but this time the computation starts just as the process is being descheduled, and all of the kernel execution times are maximal, giving an elapsed execution time of

$$r_i + \lceil \tfrac{r_i}{p\text{-}k_u} \rceil \times (d_i + k_u\text{-}p)$$

where $\lceil x \rceil$ is the upper integer part of x.

Communication delays are measured from the point at which the second (i.e later) process starts the communication procedure, to the point at which the process has noted the communication and carries on. This delay will be different for the two sides of a communication, but both sides will have the same upper and lower bounds on the delay. There are three phases to the communication, each of which contributes to the overall delay. Firstly, some processing is required to set up the choice list before the set_up_i flag is set. We define the upper and lower bounds on this time to be pre_comm_l and pre_comm_u respectively. Secondly, once set_up_i has been set, there is a delay during which the kernel will set up the communication, complete it, and eventually reschedule the process. The best case for this delay occurs when the process is descheduled just as it completes its precommunication computation, giving a delay of d_i-(p-k_l). In the worst case, the process only just fails to complete before being descheduled, giving a delay of $2 \times d_i$-(p-k_u). Thirdly there is a delay from the process next being scheduled, to the final completion of the processing required to reset the has_comm_j flag and continue to the correct branch of the choice. If we write the bounds on this time as $post\_comm_l$ and $post\_comm_u$, the minimum overall communication delay will be

$$pre\_comm_l + (d_i\text{-}(p\text{-}k_l)) + post\_comm_l$$

and the maximum will be

$$pre\_comm_u + (2 \times d_i\text{-}(p\text{-}k_u)) + post\_comm_u$$

Both of these figures assume that the very small amount of post-communication processing will be completed in one schedule block, and the maximum time can be reduced to

$$d_i + (p\text{-}k_l)\text{-}(p\text{-}k_u) + post\_comm_u = d_i + k_u\text{-}k_l + post\_comm_u$$

if it can be shown that the pre-communication processing will be completed in one schedule block (as would probably be the case if the communication immediately followed from another).

A time-out of time t, under a similar analysis, yields a minimum occurrence time of

$$pre\_comm_l + (\lfloor \tfrac{t+p}{d_i} \rfloor + 1) \times d_i\text{-}(p\text{-}k_l)$$

and a maximum of

$$pre\_comm_u + (\lceil \tfrac{t+p}{d_i} \rceil + 1) \times d_i\text{-}(p\text{-}k_u)$$

which can be reduced to

$$\lceil \frac{t+p}{d_i} \rceil \times d_i + k_u\text{-}k_l + post\_comm_u$$

if the pre-communication processing can be guaranteed to complete within one schedule block. The next section gives an example of how these calculations are done in practice.

7.3 An Example Analysis

In the above analyses, p, the scheduling period and d_i, the time between schedules for process i, may be adjusted by the implementor, but the figures k_l and k_u are fixed by the kernel. The actual figures depend on the number of processes in the system, the number of connections, and the maximum number of gates in any choice. The minimum time is of the approximate form

$$k_l = 1222 + 98 \times no\_processes + 88 \times no\_connections$$

and the upper bound of the form

$$k_u = 2222 + 268 \times no\_processes + 88 \times no\_connections + 212 \times max\_no\_in\_choice$$

where the figures given are for clock cycles on a 68000 microprocessor. For the cruise control system, which has 4 processes, 19 connections (10 internal and 9 external) and a maximum of three gates in choice, these figures work out as

$$k_l = 1222 + 98 \times 4 + 88 \times (10 + 9) = 3286\text{cycles}$$

and

$$k_u = 2222 + 268 \times 4 + 88 \times (10 + 9) + 212 \times 3 = 5602\text{cycles}$$

which gives, on an 8MHz processor, time bounds of $[0.41, 0.70]$ milliseconds.

The scheduling period chosen must be greater than the maximum time spent by the kernel, so $p > 0.7$ms. For the sake of this analysis, we will choose $p = 3$ms. The order of the processes must also be decided, and the simplest choice is to have the processes scheduled in turn (12341234...). This gives

$$d_1 = d_2 = d_3 = d_4 = 12\text{ms}$$

Having calculated k_l and k_u, and fixed p and the d_i, computation times may be calculated, based on required processing times. The only piece of computation in the car cruise-controller example is found in the `Speedo` process, and is concerned

with calculating a value for acceleration from the most recent values for the speed. If the processing time required for this computation were bounded by $[50, 55]$ms, then the lower and upper bounds on the elapsed times would be calculated as follows:

$$\begin{aligned} elapsed_l &= r_i + \lfloor \tfrac{r_i}{p\text{-}k_l} \rfloor \times (d_i\text{-}(p\text{-}k_l)) \\ &= 50 + \lfloor \tfrac{50}{3\text{-}0.41} \rfloor \times (12\text{-}(3\text{-}0.41)) \\ &= 50 + 19 \times 9.41 \\ &= 229 \text{ ms} \end{aligned}$$

and the upper bound is given by

$$\begin{aligned} elapsed_u &= r_i + \lceil \tfrac{r_i}{p\text{-}k_u} \rceil \times (d_i\text{-}(p\text{-}k_u)) \\ &= 55 + \lceil \tfrac{55}{3\text{-}0.7} \rceil \times (12\text{-}(3\text{-}0.7)) \\ &= 55 + 24 \times 9.7 \\ &= 288 \text{ ms} \end{aligned}$$

The calculated elapsed times are then bounded by $[0.229, 0.288]$ seconds, so they lie within the bounds $[0.2, 0.3]$ expressed in the design; in other words the computation falls within its design timing constraints.

Communication delays also need to be verified, for which bounds are required on the pre- and post-communication computation requirement. This computation is typically very small, and reasonable figures are

$$\begin{aligned} pre\_comm_l &= 0.01 \text{ ms} \\ pre\_comm_u &= 0.07 \text{ ms} \\ post\_comm_l &= 0.02 \text{ ms} \\ post\_comm_u &= 0.03 \text{ ms} \end{aligned}$$

Using these figures we can calculate the following bounds on communication time:

$$\begin{aligned} comm\_delay_l &= pre\_comm_l + (d_i\text{-}(p\text{-}k_l)) + post\_comm_l \\ &= 0.01 + (12\text{-}(3\text{-}0.41)) + 0.02 \\ &= 9.45 \text{ ms} \end{aligned}$$

and

$$\begin{aligned} comm\_delay_u &= pre\_comm_u + (2 \times d_i\text{-}(p\text{-}k_u)) + post\_comm_u \\ &= 0.07 + 2 \times 12\text{-}(3\text{-}0.7) + 0.03 \\ &= 22.0 \text{ ms} \end{aligned}$$

These bounds for communication delays of $[0.009, 0.022]$ lie well outside the interval $[0.001, 0.004]$ given in the design, so some more work must be done. Either the new figures must be fed back into the design, for revalidation and reverification, or some way must be found of changing the implementation to meet the bounds given in the design.

Time-out values can also be calculated from the data we already have. Considering a time-out with bounds $[0.4, 0.5]$s on its occurrence, based on an activation time of 400ms, the bounds on the behaviour of the implementation are given by

$$\begin{aligned} timeout\_occurrence_l &= pre\_comm_l + (\lfloor \tfrac{t+p}{d_i} \rfloor + 1) \times d_i\text{-}(p\text{-}k_l) \\ &= 0.01 + (\lfloor \tfrac{400+3}{12} \rfloor + 1) \times 12\text{-}(3\text{-}0.41) \\ &= 0.01 + (33 + 1) \times 12\text{-}2.59 \\ &= 405 \text{ ms} \end{aligned}$$

and

$$\begin{aligned} timeout\_occurrence_u &= pre\_comm_u + (\lceil \tfrac{t+p}{d_i} \rceil + 1) \times d_i\text{-}(p\text{-}k_u) \\ &= 0.07 + (\lceil \tfrac{400+3}{12} \rceil + 1) \times 12\text{-}(3\text{-}0.7) \\ &= 0.01 + (34 + 1) \times 12\text{-}2.3 \\ &= 418 \text{ ms} \end{aligned}$$

The bounds on the time-out occurrence in the implementation are then given by $[0.405, 0.418]$, which lie within the bounds given in the design.

There are some points to note on how the various parameters (p, d_i, pre_comm_l etc.) affect the various calculations

- The sizes of the pre- and post-communication delays are relatively insignificant.
- A lower value for the scheduling period p will reduce the upper and lower bounds on the communication delay.
- A lower value for p may increase computation delays, due to an increased amount of time spent by the kernel, and widens the upper and lower bounds on computation times.
- The difference between the lower and upper bounds on occurrence of time-outs is of the same order of magnitude as the value for d_i.

As lowering the value for p may tend to decrease communication delays and increase computation delays, some optimisation of p may be required. However, in some

situations there may be no value for p to satisfy all the time bounds requirements. Several options are available in this case, but they are discussed as part of a possible design method in section 9.4.

7.4 Timing Analysis of Priority Based Scheduling

Verifying time bounds on computations with the round-robin scheduling policy is easy, as each process has a fixed proportion of the processing time (with an allowance for jitter). To verify that each piece of computation is always completed within its time bounds for a priority-based mechanism requires a much more involved calculation. Each process may include several separate pieces of computation (which shall be referred to as *tasks*), each of which needs to have its time bounds verified, and each of which may be competing for resources with other tasks from other processes. The consideration of every possible execution path through the system to check that each task gets enough resource time is closely related to the problem of real-time model-checking, where temporal properties such as reaction time have to be verified. The approach taken to the verification of temporal properties is to calculate the state space of an AORTA design as a *region graph* of a timed graph [1], and then to apply duration model-checking techniques to this region graph [2].

Some extra information must be added to the timed graph, as propositional labels on states. The algorithm is concerned with determining whether all tasks are allocated enough processor time to complete, so information about which tasks are currently runnable must be included in the graph. Each node of the timed graph for a process is marked with a proposition which identifies whether the process is runnable or blocked, with states waiting for communication and/or time-out being blocked, and all others being runnable (including communication delay states, where a process must receive computation time before it can proceed). The propositional node labels of the process graphs (*run* or *block*) are then used, along with the priority policy, to determine which process is running — only runnable processes are eligible, but the choice from the runnable processes will depend on the priority policy. If process p is blocked, the state is labelled with $block_p$, if it is waiting (i.e. runnable but not running) it is labelled with $wait_p$, and if it is running it is labelled with run_p. The constructed system graph models the system accurately because all events which might change the process which is currently running (i.e. communication event, time-out event, or end of computation) appear as transitions within the timed graph of the system state space. This means that within a single

state in the timed graph, only one process will ever be running, so the derived 'process running' proposition is a faithful representation of the priority mechanism.

Checking that computation bounds are satisfied can now be solved by verifying accumulated durations of states in which certain propositions are true. This type of problem has been studied, and solved, using a technique which extends the region graphs used for model-checking with TCTL by adding *bound expressions* which are used to give upper and lower bounds on durations [2]. This forms the so-called *bounds graph*, which is larger than the region graph, but is finite, and is fine enough to distinguish between logical sentences which include duration bounds. Using this technique to verify upper computation bounds, the upper bound t_u of the processing time required for the task must be found, using code analysis techniques described elsewhere [80, 81, 88], and adjusted to take account of the amount of time the kernel will require for checking communications. Then all paths from the set of system states S_{start} in which the task is started must be identified — this can most easily be done when the graph is being built — and the duration bound checker [2] used to ensure that the run_p proposition is true for at least the required time. Paths are terminated by the occurrence of a ϵ_p action, which occurs at the end of the computation time allowed in the design. To verify that lower bounds are met a very similar process is adopted: lower bounds on processing time are calculated, and lower bounds on the duration are checked. Note that the timed graph of the state space only corresponds to the implementation when all bounds are met, as 'end of computation' transitions occur at latest when the upper bound is reached, rather than when enough processing time has been allocated. This does not affect the correctness of the algorithm, because the result will only be true if all bounds are attained, in which case the graph is an accurate model of the implementation.

Communication delays in AORTA model not only the time taken for communications to be noticed by the kernel, but also the time until the process can proceed with the next stage of its behaviour. If we consider the very simple process

```
A = a.b.A
```

then the communication delay for the `a` gate is the time from when a communicating partner is ready (which may be immediately), to the time when the `b` gate is ready. As each process manages its own control flow, a small amount of processing time is required to get to the next communication or computation (which may depend on choice), and this is included in the communication delay. Verification of communication delays is then very similar to verification of computation delays, except that

the processing time required is the time taken to proceed to the next part of the behaviour (typically very small), and the elapsed time must be decreased by the time for the kernel to notice the communication once it has become possible.

Having applied these verification techniques all may be well — the computation and communication bounds may be achievable with the given scheduling policy — but if there is a problem the design or implementation of the system will have to be changed so that the bounds are satisfied. As a re-design of the system may require re-verification, and will require the state space graph to be rebuilt, the implementation should be modified first. One possible modification is to change the priority policy, which will require the state space to be re-marked to indicate which processes will be running. Information can be extracted from the previous (failed) attempt at verification to indicate which computations or communications are exceeding their bounds, and this may help to inform the choice of priority. Alternatively, if there are one or two computations that are proving difficult, a re-implementation of the code, or more detailed analysis (such as [80]) may yield a tighter upper bound on processing time. In this case the graph and marking remains exactly as before, and the algorithm is re-executed using the new processing time bounds.

Whilst priority-based scheduling mechanisms offer flexible yet simple implementations of real-time systems, the verification of even simple properties of cooperating interdependent parallel processes, scheduled using a priority-based mechanism, is a difficult task. The algorithm described here does not attempt to find a feasible schedule for a set of tasks generated by parallel processes (this problem is NP-complete in general [103]), but solves the slightly easier problem of verifying that a particular scheduling policy will satisfy all its constraints. By reasoning about communication blocking at a higher level (i.e. the level of model-checking) there is no mutual blocking between the tasks (i.e. pieces of computation described in AORTA using `[...]`), and extra information is available about which tasks may be competing for resources. There are, however, no strictly periodic tasks (which are the kind most easily handled by rate monotonic reasoning [4]), and computation delays associated with communication also have to be verified.

7.5 Verification of Implementations

Apart from verifying the timings for computation and communication delays, there are other aspects of the implementation which should be verified for high levels of

assurance. Firstly, there are some aspects of timing which still have to be considered. Data dependent choice is resolved in the implementation with a branch on a given condition; the evaluation of this condition, and the branch itself, will take up processing time. For this time to be properly accounted for, it should be added to the required processing time of the computation before the choice. Secondly, when external communications are offered without choice, they are implemented as calls direct to the external function, rather than via the kernel. This is to allow longer communications (such as those involving a large amount of data transfer) to be implemented without adversely affecting the time spent by the kernel. Such communication delays must be verified separately, using the same analysis as for computations.

The behaviour of the code generator and the kernel, and the interaction between the two, is of course crucial to the correct behaviour of any implemented system. This does not require specific verification for each design, but rather a general verification that the code generator and the kernel are correct. There are currently no proofs that this is the case (although they have been reasonably well tested). The main justification that the code generator is correct, is that the code for each process is generated from the same graph as that used in the generation of timed graphs for model-checking. As the graph of the generated code and the timed graph are topologically equivalent, all that remains to check (for the individual processes) is that the labels on the edges are the same. As the labels on the edges are concerned mainly with the times of transitions, this is equivalent to verifying the timings of computations, communications and time-outs, as described in the preceding sections. The verification of the behaviour of the whole system, however, depends on the correctness of the algorithms which handle communication and time-out, given in section 6.4. At the moment, these algorithms have not been verified: this will the subject of future work.

7.6 Conclusion

The timing analysis of the implementation is one of the last stages in the development of an AORTA system (see chapter 9), but one of the most crucial. This chapter has described two ways of verifying time bounds, the first based on round-robin scheduling, the second using priority-based scheduling. Timing verification of round-robin systems is relatively straightforward, but that kind of implementation is not the most efficient. A simple tool to calculate time bounds for round-robin

scheduling has demonstrated the simplicity of the approach. Priority-based implementation is more efficient, but requires much more work to verify. This verification process has not yet been implemented as it relies on the duration bound checker [2] which is not currently available.

Both of these verification processes would be much more complicated if the restriction of static parallel composition was not made, and neither could work without nondeterministic time bounds. Similarly, communication times would be more difficult to analyse without the static connection set. The fact that realistic systems can be designed, implemented and verified justifies these restrictions that were placed on the language in chapter 3. One other restriction that was made concerned the pure synchronisation character of communication, and the representation of computations solely as time bounds. This restriction, however, is lifted in chapter 8, which allows data properties of the design to be specified.

Chapter 8

Reasoning About Data

8.1 Introduction

One of the major limitations of AORTA is that data values are not included in the formal model. This means that the functional behaviour of computations cannot be specified, that no reasoning can be done about data values that are passed during communication, and that any behaviour branches which depend on data values have to be modelled nondeterministically. The main reason for excluding data from the model is that many real-time systems do not have to deal with large amounts of data, and that the crucial properties (i.e. those which may require formal treatment) can be handled in the simpler abstraction which ignores data. However, there are systems which have critical properties relating to both timing and data, which deserve formal treatment of both aspects. For example, in the chemical plant controller example of section 4.2, one of the requirements is that an alarm is sounded within three seconds of a dangerous temperature condition occurring. This kind of requirement cannot easily be formulated as separate requirements on the timing and data parts of the system, so there are instances when a unified formal framework is required, rather than two 'orthogonal' frameworks for timing and data. Such a formal framework is presented in this chapter. Alternatively this work can be thought of as a way of introducing ordering and concurrency into model-based languages, by using a process algebra (AORTA) to specify the ordering of and cooperation between data operations.

This chapter explains how model-based data specification can be included into AORTA, to deal with computations, communications, data-dependent choice, and also real-time clock readings. Section 8.2 lists the assumptions to be made about the

modelling of data within AORTA systems, and section 8.3 describes an extended abstract syntax for AORTA with data information added. A formal semantics for the data enriched language is given in section 8.4, which also allows updates to a real-time clock variable, which can be used as data within the system. In section 8.5, VDM is developed as the data specification language, and the chemical plant controller example is reworked using the new, data-enriched language. Section 8.5 discusses some implementation and verification issues, and presents some conclusions.

8.2 Data Model Assumptions

There are two main types of functional specification languages: model-based (such as Z [87] and VDM-SL [56]) and algebraic (such as ACT ONE [77] and OBJ [41]). In a model-based language, an abstract formal model of the data in the system is built, and operations are specified and described as transformations on that model. An algebraic approach does not require a complete model to be built, and operations are specified only in terms of each other. These two approaches are not entirely incompatible, as model-based specifications can be written in an algebraic style, and models can be built into an algebraic specification, but for the purposes of this chapter the distinction is important.

One of the areas which existing functional specification techniques do not deal with satisfactorily is the issue of *when* operations are to take place, as opposed to what they do. The inclusion of operations into a process algebra is then quite natural, as the ordering of events (or operations) is the primary concern in a process algebra. However, as operations are to be interspersed with communications of the process algebra, some notion of state must be carried between operations. For this reason, a model-based approach, with a more obvious notion of state, is preferred here to an algebraic approach. There are no pressing reasons for choosing one model-based language over another, and in particular this work should be equally applicable to Z and VDM. Rather than choosing one of these languages arbitrarily, a general presentation is given here. The specifics of how VDM can be used with AORTA are given in section 8.5.

The basic assumption is that each process has a set of possible states, $States$, over which the variable Φ may range. The state Φ includes evaluations for a set of state variables. Each variable A has a set of values over which it may range, given by $values(A)$. Variables can be read using a projection $\Phi.A$, and may be

updated using the standard notation $\Phi[A = v]$ where A is a variable name and $v \in values(A)$. Operations are represented as state-valued functions of states, so an operation Δ which acts on state Φ to give state Φ' is written

$$\Delta(\Phi) = \Phi'$$

The operation which changes nothing is then the identity function on states Ξ, where

$$\Xi(\Phi) = \Phi$$

As well as accessing individual variables and performing operations on states, decisions have to be made based on the data state, which requires the definition of predicates on states, written $p(\Phi)$. Finally, two distinguished state variables are needed: $\mathcal{A}$, with $values(\mathcal{A}) = None = \{\perp\}$, and $\mathcal{T}$, with $values(\mathcal{T})$ as the time domain in use.

8.3 Extension of Syntax

According to section 3.4.1, the abstract syntax for AORTA sequential expressions is

$$S ::= \sum_{i \in I} a_i.S_i \;\mid\; [t]S \;\mid\; \sum_{i \in I} a_i.S_i \rhd^t S \;\mid\; [t_1, t_2]S \;\mid\; \sum_{i \in I} a_i.S_i \rhd_{t_1}^{t_2} S \;\mid\; \bigoplus_{i \in I} S_i \;\mid\; X$$

where t, t_1 and t_2 $(t_1 < t_2)$ are time values taken from the time domain, a_i are gate names, and X is taken from a set of process names used for recursion. A system expression is written as a product of processes with a connection set K

$$P = \prod_{i \in I} S_i < K >$$

The syntax is extended for each of these constructs apart from recursion, so rather than give the whole new syntax at once, the extensions are dealt with in turn.

8.3.1 Communication

In the original abstract syntax, communication (and its extensions to choice and time-out) uses only gate names, reflecting the pure synchronisation model of the semantics. Extending communication to include value-passing can be achieved by associating a different gate name with each data value to be offered or received (á la CCS [70]). While attractive from a theoretical point of view, as this requires only a little syntactic sugaring, it does raise some practical difficulties in implementation.

Also, the abstract specification of data state transformation via computation is difficult to incorporate into this model.

The approach adopted here is more akin to that adopted by LOTOS, with its inclusion of the ACT ONE data language for value-passing [77]. Variable names can be attached to communications as input or output parameters, using a question mark for input and an exclamation mark for output. If a value is to be read from gate a into variable A, this is written $a?A.S$, and if the value held in the variable B is to be output on gate a, this is written $a!B.S$. In the general case a gate may have input and output, written $a?A!B.S$, so the abstract syntax form for choice is

$$\sum_{i \in I} a_i?A_i!B_i.S_i$$

If no data is associated with a communication then the input and output variables are both given as the distinguished variable $\mathcal{A}$ (which always has value $\perp$), so that $a.S$ is an abbreviation for $a?\mathcal{A}!\mathcal{A}.S$. Similarly, $a?B.S$ is an abbreviation for $a?B!\mathcal{A}.S$ and $a!B.S$ is an abbreviation for $a?\mathcal{A}!B.S$. The variable $\mathcal{T}$ is used to represent a perfect local clock, and so cannot be used as a communication variable, because only approximations to the actual clock value will be available. Communications within a time-out are adapted in exactly the same way as for choice, giving the abstract syntax forms

$$\sum_{i \in I} a_i?A_i!B_i.S_i \rhd_{t_1}^{t_2} S$$

and

$$\sum_{i \in I} a_i?A_i!B_i.S_i \rhd^{t} S$$

8.3.2 Computation

Within AORTA, computations are represented only by a time delay, but during such delays a change of data state will usually take place. Operations which change state are represented by transformation functions Δ, which are attached to the time delay construct using braces. If an operation Δ takes between t_1 and t_2 time units to complete, this is represented by the abstract syntax form

$$[t_1, t_2\{\Delta\}]S$$

which has the corresponding deterministic form

$$[t\{\Delta\}]S$$

Some computations will require access to a real time clock, for time stamping or time averaging, so a special state variable $\mathcal{T}$ is used to represent a perfect local clock.

In practice, a physical clock will not be perfect, as it may run at the wrong speed, and may have its values discretised. This is modelled by defining a physical clock function on the perfect clock, which gives a set of values related to the perfect clock within some level of accuracy. During computations, time can only be accessed via the physical clock function.

8.3.3 Data dependent choice

Data dependent choice is represented as nondeterministic choice in AORTA, using the $\bigoplus_{i \in I} S_i$ notation. In order to give the conditions under which each branch of the choice is to be taken, a predicate on the state is attached to each, again using braces

$$\bigoplus_{i \in I} S_i\{p_i\}$$

Sometimes a degree of nondeterminism is helpful, so the predicates are allowed to overlap (i.e. there can be j and k such that $p_j^{-1}(true) \cap p_k^{-1}(true)$ is nonempty). There must, however, always be one predicate which is true (i.e. $\forall \Phi. \bigvee_{i \in I} p_i(\Phi)$), to ensure that some branch will be taken up.

Combining the extensions for communication, computation and data dependent choice gives the full abstract syntax for AORTA terms with data information

$$\begin{array}{rcl} S & ::= & \sum_{i \in I} a_i?A_i!B_i.S_i \\ & | & [t\{\Delta\}]S \\ & | & \sum_{i \in I} a_i?A_i!B_i.S_i \rhd^t S \\ & | & [t_1, t_2\{\Delta\}]S \\ & | & \sum_{i \in I} a_i?A_i!B_i.S_i \rhd_{t_1}^{t_2} S \\ & | & \bigoplus_{i \in I} S_i\{p_i\} \\ & | & X \end{array}$$

where t, t_1 and t_2 ($t_1 < t_2$) are time values taken from the time domain, A_i and B_i are state variable names, Δ is a state transformation function, the p_i are predicates on the state, and X is taken from a set of process names used for recursion.

8.4 Enriched Semantics for AORTA

The semantics defined in section 3.4.2 gives a stratified set of operational transition rules for defining a transition relation between AORTA terms. A similar approach is

adopted here, except that the transition system is enriched with the data state. The same interleaving time semantic model is used, with time transitions represented by $\xrightarrow{(t)}$ and action transitions (i.e. communications) represented by $\xrightarrow{\alpha}$. Again, the transition rules are stratified [43], to allow negative premises attached to rules.

To define the first stratum, we have to reconsider the set of regular expressions, on which the semantics is defined; these expressions have no nondeterminism or recursion before the next action, and are characterised by the function defined in figure 3.2. In order to include data, a regular sequential expression is annotated with a data state Φ, written

$$S[\![\Phi]\!]$$

The first of the sequential expression transition rules (which are defined only on regular expressions) are the rules for a computation delay. To simplify the notation, we abbreviate the updating of the perfect clock using the following definition

$$\Phi_{+t} \triangleq \Phi[\mathcal{T} = \Phi.\mathcal{T} + t]$$

which changes the state only by adding t to the perfect clock variable. During a computation no action transitions are available, but time may pass. The change in data state occurs at the end of the time interval, by applying the transformation function to the data state.

$$\frac{}{[t\{\Delta\}]S[\![\Phi]\!] \xrightarrow{(t')} [t\text{-}t'\{\Delta\}]S[\![\Phi_{+t'}]\!]}\ t' < t$$

$$\frac{}{[t\{\Delta\}]S[\![\Phi]\!] \xrightarrow{(t)} S[\![\Delta(\Phi_{+t})]\!]}$$

The time delay rule for choice is very straightforward

$$\frac{}{\sum_{i\in I} a_i?A_i!B_i.S_i[\![\Phi]\!] \xrightarrow{(t)} \sum_{i\in I} a_i?A_i!B_i.S_i[\![\Phi_{+t}]\!]}$$

as time passing does not affect the choice. In the communication rule, each action has two values associated with it, one for input and one for output. The input value v is used to update the value of the corresponding variable, and the output value $\Phi.B_j$ is derived from the state.

$$\frac{}{\sum_{i\in I} a_i?A_i!B_i.S_i[\![\Phi]\!] \xrightarrow{a_j?v!\Phi.B_j} [t\{\Xi\}]S'_j[\![\Phi[A_j = v]]\!]}\ \begin{array}{l} j \in I \\ S'_j \in Poss(S_j, \Phi[A_j = v]) \\ t \in delays(a_j) \\ v \in values(A_j) \end{array}$$

Note that the communication time delay which is introduced is labelled with the operation Ξ, the identity function on states. A certain amount of type-checking is

introduced by the side condition $v \in values(A_j)$. If the variable A_j is $\mathcal{A}$ (because no data is wanted) then the only allowed input value v is $\perp$, which can only be matched by a gate that is not offering output. The rules for time-out are given in a similar way, and include features from the computation and communication rules.

$$\frac{}{\sum_{i\in I} a_i?A_i!B_i.S_i \rhd^t S[\![\Phi]\!] \xrightarrow{(t')} \sum_{i\in I} a_i?A_i!B_i.S_i \rhd^{t-t'} S[\![\Phi_{+t'}]\!]} \; t' < t$$

$$\frac{}{\sum_{i\in I} a_i?A_i!B_i.S_i \rhd^t S[\![\Phi]\!] \xrightarrow{(t)} S[\![\Phi_{+t}]\!]}$$

$$\frac{}{\sum_{i\in I} a_i?A_i!B_i.S_i \rhd^t S[\![\Phi]\!] \xrightarrow{a_j?v!\Phi.B_j} [t\{\Xi\}]S'_j[\![\Phi[A_j = v]]\!]} \quad \begin{array}{l} j \in I \\ S'_j \in Poss(S_j, \Phi[A_j = v]) \\ t \in delays(a_j) \\ v \in values(A_j) \end{array}$$

Added to these rules for computation, communication and time-out, there is a rule which enforces time additivity for all expressions

$$\frac{S[\![\Phi_1]\!] \xrightarrow{(t_1)} S'[\![\Phi_2]\!] \quad S'[\![\Phi_2]\!] \xrightarrow{(t_2)} S''[\![\Phi_3]\!]}{S[\![\Phi_1]\!] \xrightarrow{(t_1+t_2)} S''[\![\Phi_3]\!]}$$

The semantics of data dependent choice is not given by transition rules, but by the definition of the *Poss* function. Any AORTA term which starts with $\bigoplus_{i\in I} S_i$ is not regular, so has to be regularised when an action transition takes place. Without any data state information, the choice between branches is nondeterministic, but by attaching predicates to the branches, a data dependent choice can be made. The definition of the new *Poss* function is as follows

$$\begin{aligned}
Poss(\sum_{i\in I} a_i?A_i!B_i.S_i, \Phi) &= \{\sum_{i\in I} a_i?A_i!B_i.S_i\} \\
Poss([t\{\Delta\}]S, \Phi) &= \{[t\{\Delta\}]S' \mid S' \in Poss(S, \Delta(\Phi_{+t}))\} \\
Poss(\sum_{i\in I} a_i?A_i!B_i.S_i \rhd^t S, \Phi) &= \{\sum_{i\in I} a_i?A_i!B_i.S_i \rhd^t S' \mid S' \in Poss(S, \Phi_{+t})\} \\
Poss([t_1, t_2\{\Delta\}]S, \Phi) &= \{[t\{\Delta\}]S' \mid t \in [t_1, t_2], S' \in Poss(S, \Delta(\Phi_{+t}))\} \\
Poss(\sum_{i\in I} a_i?A_i!B_i.S_i \rhd^{t_2}_{t_1} S, \Phi) &= \{\sum_{i\in I} a_i?A_i!B_i.S_i \rhd^t S' \mid t \in [t_1, t_2], S' \in Poss(S, \Phi_{+t})\} \\
Poss(\bigoplus_{i\in I} S_i\{p_i\}, \Phi) &= \{S'_i \mid i \in I, p_i(\Phi), S'_i \in Poss(S_i, \Phi)\} \\
Poss(X, \Phi) &= Poss(S, \Phi) \quad \text{if } X \triangleq S
\end{aligned}$$

Note that when evaluating possible regularisations of a computation, the regularisations following the computation must take note of the change to the state effected

by the operation. Also, the condition on predicates $\forall \Phi. \bigvee_{i \in I} p_i(\Phi)$ ensures that $Poss(S)$ will always be nonempty. It would be possible to use knowledge of the data state to provide tighter bounds on execution times for computations, but this would require detailed knowledge of the implementation, and would only be of use in a data intensive system.

Having given the semantics for sequential expressions, we can now define the semantics of system expressions. There are still three rules, the first of which defines internal communication. In this rule the input and output values of communicating gates are matched, and only the communicating processes are affected

$$\frac{S_j[\![\Phi_j]\!] \xrightarrow{a?u!v} S'_j[\![\Phi'_j]\!] \quad S_k[\![\Phi_k]\!] \xrightarrow{b?v!u} S'_k[\![\Phi'_k]\!]}{\prod_{i \in I} S_i[\![\Phi_i]\!] < K > \xrightarrow{\tau} \prod_{i \in I} S'_i[\![\Phi'_i]\!] < K >} \quad \begin{array}{l} (j.a, k.b) \in K \\ S'_i = S_i \text{ if } i \neq j, k \\ \Phi'_i = \Phi_i \text{ if } i \neq j, k \end{array}$$

For external communication, a similar rule is applied, with only one process being affected

$$\frac{S_j[\![\Phi_j]\!] \xrightarrow{a?u!v} S'_j[\![\Phi'_j]\!]}{\prod_{i \in I} S_i[\![\Phi_i]\!] < K > \xrightarrow{a?u!v} \prod_{i \in I} S'_i[\![\Phi'_i]\!] < K >} \quad \begin{array}{l} j \in I \\ (j.a, \_) \notin K \\ S'_i = S_i \text{ if } i \neq j \\ \Phi'_i = \Phi_i \text{ if } i \neq j \\ \prod_{i \in I} S_i[\![\Phi_i]\!] < K > \stackrel{\tau}{\not\longrightarrow} \end{array}$$

Finally, the rule for system time transitions is given as before

$$\frac{\forall i \in I. S_i[\![\Phi_i]\!] \xrightarrow{(t)} S'_i[\![\Phi'_i]\!]}{\prod_{i \in I} S_i[\![\Phi_i]\!] < K > \xrightarrow{(t)} \prod_{i \in I} S'_i[\![\Phi'_i]\!] < K >} \; \forall t' < t. \prod_{i \in I} Age(S_i[\![\Phi_i]\!], t') < K > \stackrel{\tau}{\not\longrightarrow}$$

Here the Age function is formally defined by

$$Age(E, t) = E' \;\Leftrightarrow\; E \xrightarrow{(t)} E'$$

and a similar syntactic interpretation to that given in figure 3.6 can be given.

The stratification of these transition rules is exactly as before, yielding a timed transition system between system expressions with data state. Action transitions are now labelled not only with a gate name, but also with two data values, one for input and one for output. A different transition is available for each pair of values, as is required to distinguish between similar AORTA expressions which have different data states.

This semantics is clearly very closely related to the semantics for the language without data, given in section 3.4.2. Adding data specification to the language further restricts the possible behaviour, due to restriction of data dependent choice, and

the possible restriction of communication. The translation from the data enriched language to the pure language can be given inductively as follows (the translation is signified by an over-bar):

$$\begin{aligned}
\overline{\sum_{i\in I} a_i?A_i!B_i.S_i} &= \sum_{i\in I} a_i.\overline{S_i} \\
\overline{[t\{\Delta\}]S} &= [t]\overline{S} \\
\overline{\sum_{i\in I} a_i?A_i!B_i.S_i \rhd^t S} &= \sum_{i\in I} a_i.\overline{S_i} \rhd^t \overline{S} \\
\overline{[t_1,t_2\{\Delta\}]S} &= [t_1,t_2]\overline{S} \\
\overline{\sum_{i\in I} a_i?A_i!B_i.S_i \rhd_{t_1}^{t_2} S} &= \sum_{i\in I} a_i.\overline{S_i} \rhd_{t_1}^{t_2} \overline{S} \\
\overline{\bigoplus_{i\in I} S_i\{p_i\}} &= \bigoplus_{i\in I} \overline{S_i} \\
\overline{X} &= X
\end{aligned}$$

If action transition labels are also translated, so that

$$\overline{a_i?A_i!B_i} = a_i$$

then the relationship between the two languages can be stated as follows.

Theorem 7 *For all regular sequential expressions S_i and S_i' of the data enriched language, and all data states Φ_i and Φ_i'*

1.

$$\prod_{i\in I} S_i[\![\Phi_i]\!] < K > \xrightarrow{\tau}_{DATA} \prod_{i\in I} S_i'[\![\Phi_i']\!] < K > \;\Rightarrow$$

$$\prod_{i\in I} \overline{S_i} < K > \xrightarrow{\tau}_{PURE} \prod_{i\in I} \overline{S_i'} < K >$$

2.

$$\prod_{i\in I} S_i[\![\Phi_i]\!] < K > \xrightarrow{a}_{DATA} \prod_{i\in I} S_i'[\![\Phi_i']\!] < K > \;\wedge$$

$$\prod_{i\in I} \overline{S_i} < K > \not\xrightarrow{\tau}_{PURE} \;\Rightarrow$$

$$\prod_{i\in I} \overline{S_i} < K > \xrightarrow{\overline{a}}_{PURE} \prod_{i\in I} \overline{S_i'} < K >$$

3.

$$\prod_{i\in I} S_i[\![\Phi_i]\!] < K > \xrightarrow{(t)}_{DATA} \prod_{i\in I} S_i'[\![\Phi_i']\!] < K > \;\wedge$$

$$\forall t' < t. \prod_{i\in I} Age(\overline{S_i}, t') < K > \not\xrightarrow{\tau}_{PURE} \;\Rightarrow$$

$$\prod_{i\in I} \overline{S_i} < K > \xrightarrow{(t)}_{PURE} \prod_{i\in I} \overline{S_i'} < K >$$

where $\longrightarrow_{DATA}$ *is the transition relation for the language with data, and* $\longrightarrow_{PURE}$ *is the transition relation for the language without data.*

This theorem can be interpreted as follows. All internal communication allowed in the data enriched language, is also allowed in the pure language. All external communication allowed in the data enriched language, is also allowed in the pure language, provided no extra immediate internal communications are allowed by ignoring data. All time transitions allowed in the data enriched language are also allowed in the pure language, provided no extra internal communications over the time interval are allowed by ignoring data. The addition of data to the language restricts the allowed transitions, except where the restriction of internal communication enables external communication or time transitions.

8.5 Using VDM for Data Specification

The chemical plant controller example of chapter 4 is given here as an example of how data specifications can be built into AORTA. VDM is used as the specification language here, although Z can equally well be used. Addressing the data model assumptions given in section 8.2 in turn, we first have to consider how the set of possible states of a process can be defined. In VDM this can be done by defining a composite type, including fields for each of the state variables of the process (including A and T). Invariants on the datatype can be used to restrict the state space. The set of values for each state variable is defined by its type. Selectors are used to provide projections for individual variables, and the μ function gives an easy mechanism for updating:

$$\Phi[A = v] = \mu(\Phi, A \mapsto v)$$

Operations are simply VDM operations which take no argument and return no result, but have the process state as a writable external, and no (i.e **true**) precondition. The identity function on states Ξ is simply the operation

ID

ext wr s : *States*

post $s = \overleftarrow{s}$

Finally, predicates on states are defined simply as boolean valued functions on states (i.e. of type *States* $\rightarrow$ **B**).

To construct the set of (data) states for the `Convert` process, we use five state variables, including the perfect clock $\mathcal{T}$ and the dummy $\mathcal{A}$. There are two gates of the `Convert` process which carry data, namely `in` and `out`: the state variables associated with these gates are *input:Rawdata* and *output:Temp* respectively. A lookup table is used for the conversion, and this is stored in the state variable *table:Lookuptable*. With the time domain represented as the type *Time*, the composite type representing the state of `Convert` is given by

$$
\begin{aligned}
Convert :: \quad & input : Rawdata \\
& output : Temp \\
& table : Lookuptable \\
& \mathcal{T} : Time \\
& \mathcal{A} : None
\end{aligned}
$$

Within `Convert`, there are two computations: the first converts raw data to a temperature, using a lookup table, and the second recalculates the lookup table for a different conversion mode. Assuming that we have the function

$$evaluate : Rawdata \times Lookuptable \rightarrow Temp$$

$$evaluate(x, l) \triangleq \text{value for } x \text{ in the table } l$$

then the conversion operation is defined as

DOCONVERSION

ext wr $conv : Convert$

post $conv = \mu(\overleftarrow{conv}, output \mapsto evaluate(\overleftarrow{input}, \overleftarrow{table}))$

Changing conversion mode depends on a function which recalculates the lookup table:

$$newtable : Lookuptable \rightarrow Lookuptable$$

$$newtable(l) \triangleq \text{new lookup table based on old value } l$$

The operation for changing mode is then defined as

CHANGEMODE

ext wr $conv : Convert$

post $conv = \mu(\overleftarrow{conv}, table \mapsto newtable(\overleftarrow{table}))$

To specify the behaviour of nondeterministic choice, a predicate on the state must be attached to each branch of the choice. In the `Convert` process, the behaviour depends on whether the raw data value exceeds a threshold value; if so a warning signal must be sent. The predicates which we are interested in are

$$convertdatahigh : Convert \to \mathbf{B}$$

$$convertdatahigh(conv) \triangleq input(conv) > threshold$$

and

$$convertdataok : Convert \to \mathbf{B}$$

$$convertdataok(conv) \triangleq input(conv) \leq threshold$$

which assumes that we have defined a total order $>$ on *Rawdata* and that the value *threshold:Rawdata* is defined. Attaching these new data constructs to the `Convert` process gives the definition

```
Convert = in?input.
            (Convert2 {convertdataok} ++
             warning.Convert2 {convertdatahigh})
          +
          mode.(changespeed.
                [0.3,0.4 {CHANGEMODE}]Convert)[1.5,1.505>Convert
Convert2 = [0.001,0.004 {DOCONVERSION}]
            (out!output.Convert)[1.5,1.505>Convert
```

The `Datalogger` process has its own set of states, defined by the composite type

$$\begin{aligned} Datalogger :: \quad input &: Temp \\ packet &: Loggerpacket \\ history &: (Temp \times Time)^* \\ T &: Time \\ \mathcal{T} &: Time \\ \mathcal{A} &: None \end{aligned}$$

Two of the variables, *input* and *packet* are used to carry data for communication on gates `getdata` and `senddata`, while *history* is used to record data with time stamps. The variable T is used for the physical clock, as well as the usual $\mathcal{T}$ and $\mathcal{A}$ variables. Two computations are associated with `Datalogger`, which correspond to adding a data item (with time stamp) to the store, and making up a data packet for down-loading. To get the time stamp value from the clock, we require the function

$possclocks : Time \rightarrow Time\text{-}\mathbf{set}$

$possclocks(t) \triangleq$ possible physical clock values at time t

The data which is input from the `getdata` port is added to *history* with the operation

ADDDATA

ext wr $mk\text{-}Datalogger(h, i, p, t1, t2, a) : Datalogger$

post $t1 \in possclocks(t2) \wedge h = cons((t1, i), \overleftarrow{h}) \wedge t2 = \overleftarrow{t2}$

Finally, using the function

$makepacket : (Time \times Temp)^* \rightarrow Loggerpacket$

$makepacket(h) \triangleq$ formatted text version of h

we can define the operation

MAKEPACKET

ext wr $mk\text{-}Datalogger(i, p, h, t1, t2, a) : Datalogger$

post $p = makepacket(\overleftarrow{h}) \wedge h = [] \wedge t2 = \overleftarrow{t2}$

There are no nondeterministic choices in the `Datalogger` process, so the full version of the process, including data information, is

```
Datalogger = getdata?input.[0.01,0.015 {ADDDATA}]
                (speed.Datalogger2
                 +
                 download.[0.5,1.0 {MAKEPACKET}]
                         senddata!packet.Datalogger)
                  [1.00,1.005>Datalogger
Datalogger2 = getdata?input.[0.01,0.015 {ADDDATA}]
                (speed.Datalogger
                 +
                 download.[0.5,1.0 {MAKEPACKET}]
                         senddata!packet.Datalogger2)
                 [0.25,0.255>Datalogger2
```

Having defined the individual processes, the system composition is given as before, using the | operator and a connection set, but with the addition of initial data states for each of the processes within the parallel composition.

8.6 Conclusion

Relatively little work has been done on combining timed process algebras with model-based specification languages. The most notable piece of work in this area is the MOSCA language [99], which allows process algebra constructs to be included within a VDM specification, and also allows access to a real-time clock. The approach is somewhat different to that described in this chapter, with processes forming part of the VDM specification, rather than data states being attached to processes with the process algebra. Also, no implementations techniques for MOSCA are considered.

The annotations described in this chapter are used to give a formal language in which to express data specifications for computation, communication and branching. As such they are very similar to annotations described in chapter 6 which are used for implementation. This gives rise to a natural refinement process: properties which are specified using annotations are implemented using annotations. The formal state variables are mapped onto variable declarations, computation operations are mapped onto code annotations, and values to be passed during communication are translated directly to the corresponding implementation annotations. The other important aspect of implementation is the use of the real-time clock to give data values for time stamping and so on. As discussed earlier, an implementation of a clock will not be perfect, as it will be discretised and may run at the wrong speed. However, if a real-time clock access implementation is available, it should be possible to reason about its accuracy (perhaps using an analysis similar to that of section 7.2), and give a *possclocks* function which can be used in the formal specification.

Validation and verification of designs with data specification is an area for future work, but there are several interesting possibilities. Firstly, for reactive simulation (section 5.2.2) it would be very helpful to include a data model, so that data dependent choices can be resolved without user intervention, and that simple data requirements can be validated. Secondly, although including data will often lead to infinite state spaces, it may still be possible to verify properties by model-checking, as recent work on data abstraction has shown [26]. It may prove, however, to be too large a problem to tackle with algorithmic techniques, in which case other proof techniques will be required, perhaps to be used in combination with model-checking.

Chapter 9

Evaluation of AORTA

9.1 Introduction

Over the preceding chapters AORTA has been introduced, and techniques given for simulation, verification, implementation, timing analysis, and reasoning about data. In this chapter, an evaluation of AORTA, with its associated techniques, is made. The overall aim has been to provide a practical technique for designing and implementing real-time systems in a verifiable way. In order to do this a design language has been introduced which, it is claimed, is both expressive enough to allow meaningful and useful systems to be designed, and yet suitably restrictive to allow implementation. The first of these issues, expressivity, is discussed in section 9.2, where AORTA is compared with other real-time formalisms, and possible extensions to the language are outlined. Section 9.3 examines the implementability issue, and looks at how the implementation relates to the semantics, and discusses other possible approaches to implementation. The overall practicality of the approach is the subject of section 9.4. An outline design method is proposed, and each of the stages of the design life cycle are discussed. Finally, some conclusions are presented in section 9.5.

9.2 Expressivity

For a design language to be useful, it must be expressive enough to define solutions for the range of problems for which it is likely to be used. It is unlikely that any one language will be suited to all problems (AORTA is certainly not a general purpose language), so the answer to the expressivity question depends crucially on

the type of problems which the language is expected to solve. Put another way, the question to be asked is 'for what range of problems is the language suitable'. The most important feature of AORTA, as opposed to other real-time languages, is that it has a formal semantics, and so systems can, in principle at least, be formally verified. Formal verification is most often thought of as necessary in safety-critical systems, so this is an obvious general area of application for AORTA. Similarly, the most important feature of AORTA, as opposed to other formally defined languages, is that it includes a notion of time, and has techniques for verifiable implementation. Clearly then, AORTA is most suitable for real-time applications. This is, of course, nothing more than a tautology, as AORTA was motivated by the need to provide a formal design technique for real-time safety-critical systems.

There can be no better way to establish the expressivity of a language than to exercise it on some examples. Several complete examples are given in chapter 4, showing how AORTA can be used to design systems which have a fixed set of possible input and output signals, and which have timing requirements which can be stated as bounds. Control systems often fall into this category, and indeed many motivating examples come from real-time controllers which operate in a safety-critical environment. There are, however, some types of control systems which are not easily designed by AORTA, largely because of the restriction to a fixed number of processes. In particular, an air-traffic controller, dealing with many aircraft, causes a problem. A natural solution to this problem would have a separate parallel process to deal with each aircraft, and other processes to arbitrate landing slots, air space and so on. This kind of solution would require a new process to be created for each aircraft that arrived, which is not possible in AORTA, as all parallel composition and communication links are statically defined. It would be possible, but clumsy, to obtain an estimate of the maximum number of aircraft, and to start the system with a large enough number of processes, many of which were dormant. This solution, however, would not be at all flexible, extensible, or able to degrade gracefully in the presence of transient overload. Overall, the restriction to static parallel composition and communication can cause problems, but many (and I would argue, most) real-time controllers could be designed with a fixed number of processes. The benefits, in terms of ease of analysis of implementation, ease of building timed graphs, and ease of including data into the language, outweigh the disadvantages of this particular restriction.

There are other types of problem which cannot easily be solved using AORTA. One such example is the tick-tock protocol [30], which requires a certain action to

occur regularly, with a constant time between occurrences. From an implementation point of view, it is difficult to guarantee exact timings using the techniques described in chapter 6, but even if an exact time interval is not required, there is still a problem achieving this with AORTA. The most natural way to define a system which has an output which requires a constant (or nearly constant) period is to set a timer going when the event occurs, and wait for the timer to run out before restarting. For a very simple periodic behaviour, this is possible in AORTA:

```
A = a.0[0.99,1.01>A
```

There are two problems with this solution, though

1. If the other end of the communication (i.e. whatever is connected to the gate `a`) is not ready immediately, then an extra delay comes into the cycle, which is added to the time-out delay.

2. As the time-out takes place with reference to the time that the last communication was offered, if more than one event occurs before restarting, the time-out value will not cover the whole of the periodic behaviour.

There is no way of working round these problems with basic AORTA; some such problems can be solved by using data dependent choice based on real-time clock values, but verifying the timing behaviour of such solutions is extremely difficult. It may be possible to extend AORTA to allow each process to have more than one clock, and to allow the clocks to be reset independently, with time-outs being available on all of the clocks. There are two main problems with this possible extension. Firstly, it is difficult to find a clean syntactic extension which allows arbitrary clocks to be reset and compared, and for practical reasons it is best to keep the language as small as possible. Secondly, if clocks can be reset at points other than the most recent offer of communication, then the process may be in the middle of a computation when the time-out should occur. In this case, either the time-out must wait (making the upper bound on occurrence invalid), or the computation must be interrupted, leaving a possibly inconsistent data state. Verification by model-checking would still be quite straightforward, as timed graphs [1] allow for arbitrary numbers of clocks, and for any clock to be reset on any transition. Without this kind of extension, it is difficult to express periodic events, but it is still possible to design a system and then verify that there is a minimum and maximum separation between events. Some other process algebras are able to model these kind of situations, and the tick-tock protocol in particular has been expressed in ET-LOTOS [30]. If

a practical method for implementing and analysing systems with exactly periodic processes is required, then fixed priority scheduling and rate-monotonic analysis [4] may be best suited, particularly if there is little process interaction, and specified timing constraints are simple. However, were AORTA to be extended along the lines described, it too could be used for exactly periodic processes, and could more easily handle process interaction than a rate monotonic analysis.

Although intended as a design language, the alternating bit protocol example (section 4.4) shows that AORTA can be used for modelling to some extent. In particular, the nondeterministic choice ++ may prove to be a useful tool for modelling faults. AORTA cannot be considered as a full modelling language, however, as its syntax has been restricted to allow implementation. Other timed process algebras have more operators, and can be used for modelling, (e.g. Timed CSP has been used for modelling a telephone system [57]) but they do not have the implementation techniques associated with AORTA. It would be possible to extend the language of AORTA to include features for modelling, whilst retaining an implementable subset, but this could probably not be used for refinement, because of the problems discussed in chapter 2. On the other hand, it would be possible to translate AORTA designs into other process algebras, by using nondeterministic choice for all time bounds, but the implementation techniques would still have to be based on the AORTA syntax. The main advantage of such an approach would be the possible use of compositional proof systems [51, 93], but the complexity of the translated terms may well make this infeasible. For example, in order to translate the AORTA process

```
A = a.A + b.([0.1,0.2](A ++ c.A))
```

into timed CSP [93], the binary nondeterministic choice $\sqcap$ must be extended to a general form $\bigsqcap_{i \in I} S_i$, where I can be uncountably infinite, to give the expression

$$\mu A \circ \bigsqcap_{(t_a,t_b,t) \in T} (a \xrightarrow{t_a} A \,\Box\, b \xrightarrow{t_b} (WAITt; (A \sqcap (\bigsqcap_{t_c \in delays(c)} c \xrightarrow{t_c} A))))$$

where $T = delays(a) \times delays(b) \times [0.1, 0.2]$, which is fairly difficult to read, and difficult to verify using compositional proof systems.

There are some types of design which are sometimes used for real-time systems which cannot easily be expressed in AORTA. The use of a shared area of memory for common data storage is not naturally expressed by any of the language constructs. This is not necessarily a bad thing, as concurrent access to shared memory can cause serious problems, and it may be better to have a process dedicated to the

managing of shared data. If this is not a satisfactory solution, for efficiency reasons, shared memory can be accessed during computation delays, or by the use of external actions. However, once this kind of solution has been adopted, it becomes very difficult to reason about the integrity of the data. It is possible, however, to expand the data state of the whole system to include a set of shared variables, which may be updated and read by more than one process. The use of global variables in concurrent programs is always best avoided, because of the extra effort involved in verification, so very good justification needs to be given before using this kind of design.

Perhaps more problematic is the inability to express asynchronous communication (e.g. network broadcasts), which can be used, amongst other things, to make shared data more easily and efficiently accessible. This is an important issue, particularly for distributed real-time systems, and should be investigated further.

9.3 Implementability

Chapters 6 and 7 showed that AORTA designs can be implemented verifiably on a single processor with multitasking, with a variety of scheduling mechanisms. This implementability is an important feature, and none of the process algebras mentioned in section 2.4.2 are fully implementable in this way, as they can only give exact timing information. If it can be assumed that the computer is fast enough to make response times negligible, then it may be possible to implement these other algebras. Implementations of timed synchronous languages [8] make this assumption, but no other other work has been done to date on implementing timed process algebras.

One of the advantages of using a process algebra for design is that the notion of a process is abstract enough to cover multitasking, parallel processing, distributed systems and hardware processes. It is important, then, to consider whether AORTA designs are implementable in these other ways. The main problem in implementing processes on more than one processor is that of communication, as the division of tasks has already been done in the AORTA design, and the number of processes cannot change. If the communications which take place between processors do not involve any choice, then point-to-point connections provide a very easy implementation. For inter-processor communication which includes choice, a more complicated mechanism is required: this is an area for future work. As some multitasking may also be taking place on the processors (if the number of processes exceeds the num-

ber of processors), some communication will be handled purely internally by the kernel, and some may require cooperation with another processor. This will lead to different connections having different implementations, and hence different delays. The possibility is already catered for in AORTA, as it allows communication delays to be associated with individual gates. Although interesting, the use of hardware to implement AORTA processes has not been considered.

9.4 Practicality

As one of the major motivations for this work is to make formal methods for real-time systems more practical, an evaluation of the practicality of the approach is important. Other projects which have examined the problem of formal development of real-time systems have yet to produce a practical solution to the problem of specifying, designing, implementing and verifying a real-time system. For example, the ProCoS project aims to provide a complete framework in which real-time systems can be built and verified, so has similar goals to AORTA. Indeed, many similar assumptions are made, such as the use of statically defined concurrency in the programming language tPL, but the approach is based on refinement with a common language, the duration calculus [23]. The ProCoS project is much more ambitious than the work described in this thesis, as it aims to provide verification from specification to hardware; it has added much to the underpinning knowledge of the subject area, but has yet to demonstrate the practicality of its approach.

Some features of AORTA, such as the use of only ASCII characters for the concrete syntax, are there for purely practical reasons: software tool support is made easier by having sources that can be manipulated using a standard editor. Although it is difficult to evaluate the practicality of the technique without full tool support, this section attempts to demonstrate that the approach is feasible by explaining a possible development cycle, including the use of tools that are already in place. Figure 9.1 shows how a development might proceed, starting with the writing of the specification of the system, and finishing with testing. Each of the steps is now considered in turn.

Writing the formal specification. For a system without specified data properties, the formal part of the specification will take the form of a timed temporal logic sentence, in a language such as TCTL [1]. The exact way in which the specification is obtained from the requirements may vary. A formal hazard and operability analysis (hazop) may be used to establish conditions which

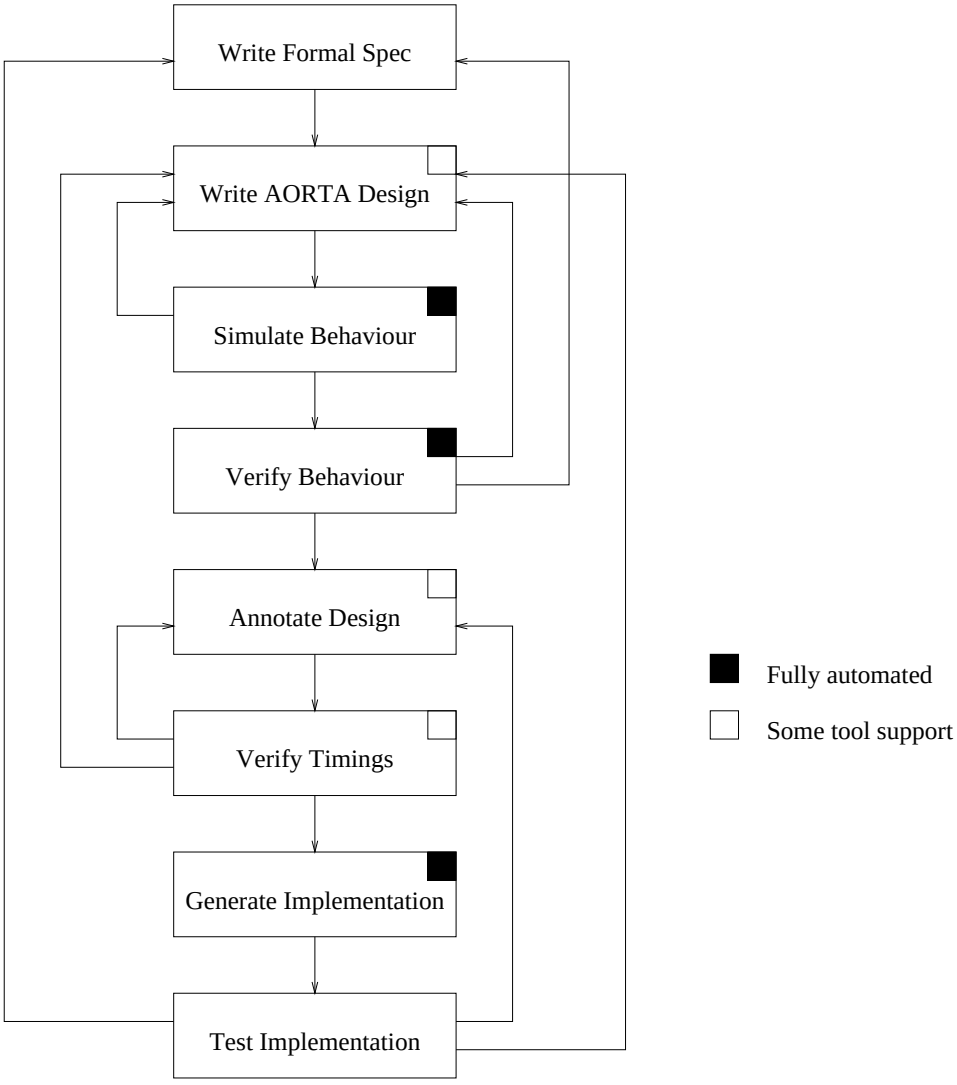

Figure 9.1: The development cycle

should not arise; these may then be translated into temporal logic. It may be necessary to use control theory to calculate some of the required response times, particularly if a complex process is being controlled. Formalisation of a specification often helps to clarify within the specifier's own mind exactly what is required of a system; it is important that this is transmitted to the designer and the end user, by use of natural language to support the formal part.

Writing the AORTA design. Writing a design to satisfy a specification is essentially a creative process, and requires a knowledge of the specification language, the design language, the application area, and some familiarity with possible implementations. It is impossible to give a general algorithm for finding a design which satisfies the specification, as the satisfiability problem for dense real-time logics such as TCTL is undecidable [1], although algorithms do exist for finding solutions in an untimed setting [85, 86]. As well as satisfying the timing requirements of the specification, the times given in a design for communication and computation delays must be achievable by the implementation. Producing a design can be thought of as balancing the demands of the specification, particularly upper bounds on response times, against the restrictions of the implementation. The development of the design is partly tool supported, as a standard text editor can be used to prepare the design, and static semantic checks are carried out (all gates are linked, and all links refer to real gates).

Simulating the behaviour. The first step in determining the correctness of the design is to validate it by simulation (as discussed in section 5.2). Simulation may reveal behaviour that is unexpected: deadlocks may occur, responses may be incorrect, or response times may not be satisfactory. If there is a problem of some sort, it may well be that only the design needs to be reworked, as it is clear that the design does not satisfy its specification. However, if a behaviour is found which satisfies the specification but not the requirements, then the specification and the design may need to be reworked. One of the advantages of simulation is that it can show up faults in the specification, by demonstrating possible behaviours to the end user (who may not fully understand the specification language).

Verifying the behaviour. Verifying the behaviour of the design is one of the most time-consuming activities in the whole development, so care must be

taken to minimise the effort required. Although model-checking procedures for verification are completely automatic, the number of states within a relatively simple system can be very large, so the computer time required may be great. Careful simulation should reduce the possibility of trying to verify a system that is incorrect, but if the timings given in the design are unimplementable then the design and verification will have to be reworked later in the development.

Annotating the design. Once a correct design has been established, it must be implemented. A large part of the implementation is generated automatically, but those parts which are concerned with data, and with external communication, must be written by hand. Each piece of computation in the design will need some code associated with it; the design should be commented to indicate what is required of each computation. Any values which are to be passed during communication must also be annotated, as well as variable declarations and function definitions.

Verifying the timings. Once code has been written for all computations, upper and lower bounds must be found for the amount of processing time required. Techniques do exist for finding such estimates [88, 95, 80, 81], but they have yet to be integrated into the AORTA tool-set. Once these times have been established, the elapsed times for computations, as well as communication and time-out times, must be verified with respect to the times given in the design. As these timings depend crucially on the hardware and software architecture of the system, the architecture has to be decided upon at this point (if it has not been made already). For a single processor system, the type and speed of processor must be decided, as well as the scheduling mechanism. The task-switch rate (fixed by the interrupt period) affects all of the calculations, and each scheduling mechanism has its own parameters: the order of processes in round robin scheduling, and the process priorities otherwise.

The simplest verification to perform is that for round-robin scheduling, with only a small piece of arithmetic required for each computation, communication and time-out, and can be done by computer. If bounds cannot be met on the available hardware, even after optimisation of the task-switch rate and process ordering, then a different scheduling policy must be chosen. Fixed priority scheduling is the next most obvious choice, and priorities should be assigned to processes corresponding to the difficulty that was found in meeting the

bounds using round-robin scheduling (e.g. processes with computations which overran upper bounds by a long way should be given high priority). Earliest deadline first scheduling is optimal if task-switching overheads are negligible, so may be considered if fixed priority scheduling cannot meet all deadlines. If none of these methods can produce guarantees that the bounds given in the design can be met, then three options are available:

1. Change the hardware: upgrade the processor type and clock speed, or consider parallel and/or distributed implementation.
2. Change the software: find more efficient implementations for pieces of computation that are overrunning.
3. Change the design: widen time bounds that cannot be met, or find some other way of dividing up computation.

Each of these options has drawbacks. Upgrading the hardware will obviously incur a cost, but this may be small compared with development costs if production numbers are low. Making the software more efficient takes time, and the verification of time bounds on required processing time may well be more involved. Changing the design means that a new validation and verification process must take place.

Generating the implementation. Generating the implementation from the annotated design is automatic: code for the processes is generated and compiled, information is given to the kernel about scheduling and communication links, and the code is linked with the kernel.

Testing the implementation. Most parts of the system design are subject to formal verification, but it is still necessary to test the final system, in order to provide assurance that the assumptions made during verification are valid. Systems can be tested outside their working environment by replacing some of the external function drivers with dummies, or with drivers suitable for a test harness. There are some parts of the system which have not been subject to any kind of verification, so particular attention should be paid to testing the data aspects of the system, as well as external communication links. Errors may result from incorrect annotations, so these may need to be revised, or it may be that part of the design needs correction to satisfy part of the requirement that was not formally specified (e.g. a data property). Finally, testing may reveal some flaw in the behaviour that should have been

eliminated by the specification, but was not picked up during simulation. In this case the specification and design will both need to be amended, and the verification reworked.

If the data language, described in chapter 8 is used, then there will be extra work in the development. During formal specification, data properties will have to be stated; these may be independent of timing properties, or they may be interrelated (e.g. within two seconds of the temperature reading exceeding the threshold level the alarm must sound). It may still be possible to state some combined properties using a timed temporal logic, using data abstraction techniques [26], but otherwise a more standard formal specification language, such as VDM-SL [56], or Z [87] should be used. During design, the data states and individual computations and communications must be specified, using the annotations described in section 8.3. Simulation of behaviour using the data language should, in principle, be easier, as data dependent choices can be resolved automatically, but more work needs to be done on the simulator(s) to achieve this. Verification is a more difficult problem: automatic verification of design correctness by model-checking may still be possible in some cases [26], otherwise more general proof techniques are required, which are yet to be developed in conjunction with AORTA. Compositional reasoning, and the use of proof-assistants, rather than automatic model-checking algorithms, may well be necessary for systems with extremely large state spaces. For this purpose, an extension of syntax to allow systems to be built in a more compositional way would be necessary, and would also be useful for the management of process re-use. After the annotation of the design with the implementation code, verification may also be required of the functional correctness of the code, as well as the the timing. Once the annotated and verified design has been achieved, the process of implementation generation and testing should be the same as before.

9.5 Conclusion

In this chapter, AORTA has been evaluated according to its aim of being a practical formal design language for hard real-time systems. Its expressivity was examined in section 9.2, with the main positive evidence being the examples of chapter 4. Possible extensions of the language, to extend the expressivity whilst retaining implementability, were also considered. Section 9.3 briefly discussed how AORTA could be implemented in ways other than those presented in chapter 6. A development method for AORTA was outlined in section 9.4 to provide some justification

for the claim that the language can be used practically, rather than just as a toy.

AORTA is by no means a general purpose language, even within the realm of real-time systems. If concurrency is not required, then a timed extension of a specification language such as Z [28, 34, 67] may be more appropriate. In situations where modelling, rather than design, is required, then a more expressive language should be used: many of the timed process algebras mentioned in section 2.4.2 are in some ways more expressive than AORTA, and Petri nets extended with time should also be considered [9, 12, 33, 72, 64, 91]. Using different assumptions for implementation, a synchronous language such as Esterel [8] may be appropriate (where computation times are negligible), and rate-monotonic analysis [4] (RMA) is a relatively simple technique which can be used where processes are periodic, and do not have too much interaction.

Whichever technique is used, the fundamental difficulty of building real-time systems cannot easily be overcome. Traditional wisdom tells us that implementation details should be considered later, rather than earlier, in the development, and proponents of formal methods argue that by performing verification early in the development, the expensive reworking of implementation can be reduced. Unfortunately, where real-time constraints are the primary concern, implementation considerations have to appear earlier in the development, and more iterations around the development loop may be necessary to achieve the required performance. This is demonstrated by figure 9.1, where annotations, designs, or even specifications may need to be rewritten late on in the development, if an implementation cannot be found. Many formal methods rely on refinement, where the final solution is reached incrementally from the specification, with proofs being given along with refinement steps; a lot of effort may be wasted in using this approach, if designs (and proofs) have to be reworked because an implementation could not be found. The way in which AORTA tries to alleviate this problem is to allow simulation early on, to automate the verification process, and to generate implementations automatically from the design, making reworking of designs and implementations quicker and less prone to error.

Chapter 10

Conclusion

In conclusion to this thesis, a review of the chapters is made, with references to possible further work where appropriate. Chapter 1 and 2 introduced the real-time problem, and some of the proposed solutions. Having concluded that formal methods may well be appropriate to real-time systems, particularly those which are safety-critical, it was noted that existing formal methods consider either only too high a level of abstraction, or too low a level. Chapter 3 introduced AORTA, which attempts to bridge the gap between high level and low level reasoning. Several restrictions are placed on the language, including static definition of parallelism and communication channels. AORTA is essentially a timed process algebra, and has the standard process algebra assumption that all communication is synchronised; the inclusion of a broadcast communication, useful for many networked systems, is an area of study which may prove to be useful.

To demonstrate that the restrictions imposed on the language do not seriously impair its expressivity for design, chapter 4 presents a series of examples of real-time designs expressed in AORTA. Following that, chapter 5 discusses how designs can be validated by simulation, and verified by model-checking. Simulation has shown its worth as a high-level debugging tool, but the extension of the menu driven simulator to allow temporal logic properties to be checked, along the lines of [90], and further work on the reactive simulator, could make validation easier and quicker. Also, the integration of model-checking with an automated proof checker could dramatically reduce the time required to verify a design.

The main novelty of the AORTA language is its implementability, and chapter 6 shows how annotated designs can be implemented automatically by using multitasking on a single processor. Distributed and parallel implementations are important

areas for future study. The analysis and verification of AORTA implementations was the subject of chapter 7, but for a more rigorous development, algorithms used by the kernel should be verified. Also, the analysis of computation times would be greatly eased by integration with an automatic code timing analyser.

Data modelling is not included in the basic language, but chapter 8 shows how to include model-oriented data specifications of communication, computation and data dependent choice. In particular, the integration of VDM into the syntactic and semantic frameworks is described. No real verification techniques are offered, so the investigation of data abstraction to allow model-checking of some data properties should be considered further. Also, the inclusion of data information into simulation should reduce the need for user intervention in simulation.

Finally, in chapter 9, AORTA is evaluated by examining its expressivity, its implementability, and its practicality. A possible design method is proposed, although little detail on requirements analysis is given. The use of a hazard and operability analysis to derive specifications from requirements would make the development process more complete. Various possible extensions to the language are discussed, perhaps the most important being the addition of a modules system for compositional reasoning and design reuse.

As well as particular developments and extensions of the language and techniques, it is important that the on-going evaluation of them is continued, by exercising the notation on the practical development of example systems. The research can only be considered complete when a practical method for specifying, designing, implementing and verifying real-time systems, supported by software tools, is in place.

References

[1] R Alur, C Courcoubetis, and D Dill. Model-checking for real-time systems. In *IEEE Fifth Annual Symposium On Logic In Computer Science, Philadelphia*, pages 414–425, June 1990.

[2] R Alur, C Courcoubetis, and T A Henzinger. Computing accumulated delays in real-time systems. In *Fifth International Conference on Computer-aided Verification (CAV 1993). Lecture Notes in Computer Science 697*, pages 181–193. Springer-Verlag, 1993.

[3] R Alur and T A Henzinger. Real-time logics: Complexity and expressiveness. In *IEEE Fifth Annual Symposium On Logic In Computer Science, Philadelphia*, pages 390–401, June 1990.

[4] N C Audsley, A Burns, R I Davis, K W Tindell, and A J Wellings. Fixed priority pre-emptive scheduling: An historical perspective. *Real-Time Systems*, 8(2/3):173–198, March/May 1995.

[5] J C M Baeten and J A Bergstra. Real time process algebra. *Formal Aspects of Computing*, 3(2):142–188, 1991.

[6] H Barringer, M Fisher, D Gabbay, G Gough, and R Owens. MetateM: A framework for programming in temporal logic. Technical Report Series UMCS-89-10-4, Department of Computer Science, University of Manchester, Oxford Rd, Manchester, October 1989.

[7] K A Bartlett, R A Scantlebury, and P T Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Communications of the ACM*, 12(5):260–261, 1969.

[8] D Berry, S Moisan, and J Rigault. Esterel: towards a synchronous and semantically sound high level language for real-time applications. In *IEEE Real time systems symposium*, pages 30–37, December 1983.

[9] B Berthomieu and M Diaz. Modeling and verification of time dependent systems using time Petri nets. *IEEE Transactions on Software Engineering*, 17(3):199–273, March 1991.

[10] R Bol and J F Groote. The meaning of negative premises in transition system specifications. In *18th International Colloquium on Automata, Languages and Programming*, pages 481–494. Springer-Verlag, 1992.

[11] T Bolognesi and F Lucidi. LOTOS-like process algebras with urgent or timed interactions. In K R Parker and G A Rose, editors, *Formal Description Techniques IV, FORTE '91, Sydney*, pages 249–264. Elsevier, November 1991.

[12] T Bolognesi, F Lucidi, and S Trigila. From timed Petri nets to timed LOTOS. In L Logrippo, R L Probert, and H Ural, editors, *Protocol Specification, Testing and Verification X*, pages 395–408. Elsevier, 1990.

[13] S Bradley, W Henderson, D Kendall, and A Robson. A formally based hard real-time kernel. *Microprocessors and Microsystems*, 18(9):513–521, November 1994.

[14] S Bradley, W Henderson, D Kendall, and A Robson. Practical formal development of real-time systems. In *11th IEEE Workshop on Real-Time Operating Systems and Software, RTOSS '94, Seattle*, pages 44–48, May 1994.

[15] S Bradley, W D Henderson, D Kendall, and A P Robson. Application-Oriented Real-Time Algebra. *Software Engineering Journal*, 9(5):201–212, September 1994.

[16] S Bradley, W D Henderson, D Kendall, and A P Robson. Designing and implementing correct real-time systems. In H Langmaack, W-P de Roever, and J Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems FTRTFT '94, Lubeck, Lecture Notes in Computer Science 863*, pages 228–246. Springer-Verlag, September 1994.

[17] S Bradley, W D Henderson, D Kendall, and A P Robson. Modelling data in a real-time algebra. Technical Report NPC-TRS-95-1, Department of Computing, University of Northumbria, UK, 1995. Submitted for publication.

[18] S Bradley, W D Henderson, D Kendall, A P Robson, and S Hawkes. A formal design and implementation method for systems with predictable performance. Technical Report NPC-TRS-95-2, Department of Computing, University of Northumbria, UK, 1995. Submitted for publication.

[19] S Bradley, D Kendall, W D Henderson, and A P Robson. Validation, verification and implementation of timed protocols using AORTA. In Piotr Dembinski and Marek Sredniawa, editors, *Protocol Specification, Testing and Verification XV (PSTV '95), Warsaw*, pages 193–208. IFIP, North Holland, June 1995.

[20] L Brim. Analysing time aspects of dynamic systems. In *EMSCR'92 Vienna*, pages 1–8. EMSCR, 1992.

[21] A Burns. Scheduling hard real-time systems : a review. *Software Engineering Journal*, 6(3):116–128, May 1991.

[22] Z Chaochen, M R Hansen, A P Ravn, and H Rischel. Duration specifications for shared processors. In J Vytopil, editor, *Formal techniques in real-time and fault-tolerant systems Second international symposium, Nijmegen, Lecture Notes in Computer Science 571*, pages 21–32. Springer-Verlag, 1992.

[23] Zhou Chaochen, C A R Hoare, and A P Ravn. A calculus of durations. *Information Processing Letters*, 40(5), December 1991.

[24] L Chen. An interleaving model for real-time systems. Technical Report ECS-LFCS-91-184, Edinburgh University, November 1991.

[25] E M Clarke, E A Emerson, and A P Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[26] E M Clarke, O Grumberg, and D E Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[27] R Cleaveland, J Parrow, and B Steffen. The concurrency workbench: A semantics-based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

[28] A Coombes and J McDermid. Specifying temporal requirements for distributed real-time systems in Z. *Software Engineering Journal*, 8(5):273–283, September 1993.

[29] M Daniels. Modelling real-time behavior with an interval time calculus. In J Vytopil, editor, *Formal techniques in real-time and fault-tolerant systems Second international symposium, Nijmegen, Lecture Notes in Computer Science 571*, pages 53–71. Springer-Verlag, 1992.

[30] C Daws, A Olivero, and S Yovine. Verifying et-lotos programs with kronos. In *7th International Conference on Formal Description Techniques (FORTE '94), Berne Switzerland*. Elsevier, 1994.

[31] H-P Eberhard. Porting and timing a hard real-time kernel. Final Year Project, Department of Medical Informatics, Heilbronn University, September 1995.

[32] E A Emerson, A K Mok, A P Sistla, and J Srinivasan. Quantitative temporal reasoning. *Real-Time Systems*, 4(4):331–352, December 1992.

[33] M Felder, C Ghezzi, and M Pezze. High-level timed Petri nets as a kernel for executable specifications. *Real Time Systems*, 5(2/3):235–248, May 1993.

[34] C J Fidge. Specification and verification of real-time behaviour using Z and RTL. In J Vytopil, editor, *Formal techniques in real-time and fault-tolerant systems Second international symposium, Nijmegen, Lecture Notes in Computer Science 571*, pages 393–409. Springer-Verlag, 1992.

[35] C J Fidge. Real-time refinement. In J C P Woodcock and P G Larsen, editors, *Formal Methods Europe '93: Industrial-Strength Formal Methods, Lecture Notes in Computer Science 670*, pages 314–331. Springer-Verlag, 1993.

[36] R Gerber and I Lee. CCSR: A calculus for communicating shared resources. In J C M Baeten and J W Klop, editors, *CONCUR '90, Amsterdam, Lecture Notes in Computer Science 458*, pages 263–276. Springer-Verlag, August 1990.

[37] R Gerber and I Lee. A layered approach to automating the verification of real-time systems. *IEEE Transactions on Software Engineering*, 18(9):768–784, September 1992.

[38] C Ghezzi, D Mandrioli, and A Morzenti. TRIO: A logic language for executable specifications of real-time systems. Technical Report 89-006, Politecnico di Milano, 1989.

[39] D Gilbert. Executable LOTOS. In Rudin and West, editors, *Protocol Specification, Testing and Verification VII*, pages 281–294. Elsevier, 1987.

[40] J C Godskesen and K G Larsen. Real-time calculi and expansion theorems. In S Purushothaman and A Zwarico, editors, *Proceedings of First North American Process Algebra Workshop (NAPAW 92), Stony Brook, New York, USA*, pages 3–12. Springer-Verlag, August 1992.

[41] J A Goguen and T Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-9, SRI, August 1988.

[42] A Goswami, M Bell, and M Joseph. ISL: An interval logic for the specification of real-time programs. In J Vytopil, editor, *Formal techniques in real-time and fault-tolerant systems Second international symposium, Nijmegen, Lecture Notes in Computer Science 571*, pages 1–20. Springer-Verlag, 1992.

[43] J F Groote. Transition system specifications with negative premises. In J C M Baeten and J W Klop, editors, *CONCUR '90, Amsterdam, Lecture Notes in Computer Science 458*, pages 332–341. Springer-Verlag, 1990.

[44] H Hansson. A calculus for communicating systems with time and probabilities. In *IEEE 11th real-time systems symposium, Lake Buena Vista*, pages 278–287. IEEE, 1990.

[45] D Harel, H Lachover, A Naamad, A Pnueli, M Politi, R Sherman, A Shtull-Trauring, and M Traktenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–413, April 1990.

[46] E Harel, O Lichtenstein, and A Pnueli. Explicit clock temporal logic. In *IEEE Fifth Annual Symposium On Logic In Computer Science, Philadelphia*, pages 402–413, June 1990.

[47] D J Hatley and I A Pirbhai. *Strategies for Real-Time System Specification.* Dorset House, New York, 1988.

[48] T A Henzinger, Z Manna, and A Pnueli. Temporal proof methodologies for real-time systems. In *Proceedings of the 18th Annual Symposium on Principles of Programming Languages*, pages 353–366. ACM Press, 1991.

[49] C J Ho-Stuart, H S M Zedan, M Fang, and C M Holt. PARTY: A process algebra with real-time from York. Technical Report YCS-92-177, York University, Department of Computer Science, 1992.

[50] C A R Hoare. *Communicating Sequential Processes.* Prentice Hall, New York, 1985.

[51] J Hooman. Compositional verification of real-time systems using extended Hoare triples. In J W de Bakker, C Huizing, W P de Roever, and G Rozenberg,

editors, *Real-Time: Theory in Practice (REX workshop), Mook, Lecture Notes in Computer Science 600*, pages 252–290. Springer-Verlag, June 1991.

[52] J J M Hooman and W P de Roever. Design and verification in real-time distributed computing: an introduction to compositional methods. In E Brinksma, G Scollo, and C A Vissers, editors, *Potocol Specification, Testing and Verification IX*, pages 37–56. Elsevier, 1990.

[53] F Jahanian, R Lee, and A K Mok. Semantics of Modecharts in real time logic. In *Proceedings of 21st Hawaii International conference on system Science*, pages 479–489. IEEE, IEEE Press, 1988.

[54] F Jahanian and A K Mok. Safety analysis of timing properties in real-time systems. *IEEE Transactions on Software Engineering*, SE-12(9):890–904, September 1986.

[55] F Jahanian and A K-L Mok. A graph-theoretic approach for timing analysis and its implementation. *IEEE Transactions on Computers*, 36(8):961–975, August 1987.

[56] C B Jones. *Systematic software development using VDM*. Prentice Hall, New York, 1986.

[57] A Kay and J N Reed. A rely and guarantee method for timed CSP: A specification and design of a telephone exchange. *IEEE Transactions on Software Engineering*, 19(6):625–639, June 1993.

[58] Y Kesten and A Pnueli. Timed and hybrid statecharts and their textual representation. In J Vytopil, editor, *Formal techniques in real-time and fault-tolerant systems Second international symposium, Nijmegen, Lecture Notes in Computer Science 571*, pages 591–620. Springer-Verlag, 1992.

[59] Klein. *A practitioner's handbook for real-time analysis: guide to rate monotonic*. Kluwer, 1993.

[60] H Kopetz, A Damm, C Koza, M Mulazzani, W Swabl, C Senft, and R Zainlinger. Distributed fault-tolerant real-time systems: The MARS approach. *IEEE Micro*, 9(1):25–40, February 1989.

[61] D Kozen. Results on the propositional mu-calculus. In *Theoretical Computer Science*, pages 333–354. Elsevier, 1983.

[62] P Krishnan. A model for real-time systems. In *16th International Symposium on Foundations of Computer Science, Kazimierz*, pages 298–307, 1991.

[63] G Leduc. An upward compatible timed extension to LOTOS. In K R Parker and G A Rose, editors, *Formal Description Techniques IV, FORTE '91, Sydney*. Elsevier, November 1991.

[64] N G Leveson and J L Stolzy. Safety analysis using Petri nets. *IEEE Transactions on Software Engineering*, 13(3):386–397, March 1987.

[65] N G Leveson and C S Turner. An investigation of the Therac-25 accidents. *IEEE Computer*, 26(7):18–41, July 1993.

[66] H R Lewis. A logic of concrete time intervals. In *IEEE Fifth Annual Symposium On Logic In Computer Science, Philadelphia*, pages 380–389, June 1990.

[67] B P Mahony and I J Hayes. A case-study in timed refinement: A mine pump. *IEEE Transactions on Software Engineering*, 18(9):817–825, September 1992.

[68] P M Melliar-Smith. Extending interval logic to real-time systems. In B Banieqbal, H Barringer, and A Pnueli, editors, *Temporal Logic in Specification, Lecture Notes in Computer Science 398*, pages 224–242. Springer-Verlag, 1987.

[69] C Miguel, A Fernandez, J M Ortuno, and L Vidaller. A LOTOS based peformance evaluation tool. To appear in Computer networks and ISDN Systems, April 1992.

[70] R Milner. *Communication and Concurrency*. Prentice Hall, New York, 1989.

[71] F Moller and C Tofts. A temporal calculus of communicating systems. Technical Report ECS-LFCS-89-104, Edinburgh University, December 1989.

[72] M K Molloy. Performance analysis using stochastic Petri nets. *IEEE Transactions on Computers*, c-31(9):913–917, September 1982.

[73] B Moszkowski. *Executing Temporal Logic Programs*. C.U.P., 1986.

[74] X Nicollin and J Sifakis. An overview and synthesis on timed process algebras. In J W de Bakker, C Huizing, W P de Roever, and G Rozenberg, editors, *Real-Time: Theory in Practice (REX workshop), Mook, Lecture Notes in Computer Science 600*, pages 526–548. Springer-Verlag, 1991.

[75] X Nicollin, J Sifakis, and S Yovine. From ATP to timed graphs and hybrid systems. In J W de Bakker, C Huizing, W P de Roever, and G Rozenberg, editors, *Real-Time: Theory in Practice (REX workshop), Mook, Lecture Notes in Computer Science 600*, pages 549–572. Springer-Verlag, June 1991.

[76] X Nicollin, J Sifakis, and S Yovine. Compiling real-time specifications into extended automata. *IEEE Transactions of Software Engineering*, 18(9):794 – 804, 1992.

[77] International Standards Organisation. *Informations processing systems - Open Systems Interconnection - LOTOS - A formal description technique based on the temporal ordering of observational behaviour*, volume ISO 8807. ISO, 1989-02-15 edition, 1989.

[78] J S Ostroff. Formal methods for the specification and design of real-time safety critical systems. *Journal of Systems and Software*, 18(1):33–60, April 1992.

[79] J S Ostroff. A verifier for real-time properties. *Real-Time Systems*, 4(1):5–36, March 1992.

[80] C Y Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, 5(1):31–62, March 1993.

[81] C Y Park and A C Shaw. Experiments with a program timing tool based on source-level timing schema. *IEEE Computer*, 24(5):48–57, May 1991.

[82] J L Peterson. Petri nets. *Computing Surveys*, 9(3):223–252, 1977.

[83] M Phillips. CICS/ESA 3.1 Experiences. In J E Nicholls, editor, *Z User Workshop, Oxford, 1989*, pages 179–185. Springer-Verlag, Workshops in Computing, 1990.

[84] G D Plotkin. A structural approach to operational semantics. Technical Report DAIMI-FN-19, Aarhus University, Computer Science Department, 1981.

[85] A Pnueli and R Rosner. On the synthesis of a reactive module. In *16th ACM symposium on Principles of Programming Languages*, pages 179–190, 1989.

[86] A Pnueli and R Rosner. On the synthesis of an asynchronous reactive module. In *16th International Colloqium on Automata, Languages and Programs*, pages 652–671, 1989.

[87] B Potter, J Sinclair, and D Till. *An Introduction to formal specification and Z*. Prentice Hall, New York, 1991.

[88] P Puschner and Ch Koza. Calculating the maximum execution time of real-time programs. *Real-Time Systems*, 1:159–176, 1989.

[89] J Quemada and A Fernandez. Introduction of quantitative relative time into LOTOS. In H Rudin and C H West, editors, *Protocol Specification, Testing and Verification VII, Zurich*, pages 105–121. Elsevier, 1987.

[90] S C V Raju and A C Shaw. A prototyping environment for specifying, executing and checking communicating real-time state machines. *Software — Practice and Experience*, 24(2):175–195, February 1994.

[91] R R Razouk and C V Phelps. Performance analysis using timed Petri nets. In S Yemini Y Yemini, R Strom, editor, *Protocol Specification, Testing and Verification, IV*, pages 561–576. Elsevier, 1985.

[92] J H Reppy. *Concurrent Programming with events. The Concurrent ML manual*. AT&T laboratories, February 1993.

[93] S Schneider, J Davies, D M Jackson, G M Reed, J N Reed, and A W Roscoe. Timed CSP: Theory and practice. In J W de Bakker, C Huizing, W P de Roever, and G Rozenberg, editors, *Real-Time: Theory in Practice (REX workshop), Mook, Lecture Notes in Computer Science 600*, pages 640–675. Springer-Verlag, June 1991.

[94] D J Scholefield and H S M Zedan. TAM: A formal framework for the development of distributed real-time systems. In J Vytopil, editor, *Formal techniques in real-time and fault-tolerant systems Second international symposium, Nijmegen, Lecture Notes in Computer Science 571*, pages 411–428. Springer-Verlag, 1992.

[95] A C Shaw. Reasoning about time in higher-level language software. *IEEE Transactions on Software Engineering*, 15(7), July 1989.

[96] A C Shaw. Communicating real-time machines. *IEEE Transactions on Software Engineering*, 18(9):805–816, September 1992.

[97] T Shepard and J A M Gagné. A pre-run-time scheduling algorithm for hard real-time systems. *IEEE Transactions on Software Engineering*, 17(8):669–677, August 1991.

[98] R Sisto, L Ciminiera, and A Valenzano. A protocol for multirendezvous of LOTOS processes. *IEEE transactions on computers*, 40(1):437–446, April 1991.

[99] H Toetenel. VDM + CCS + Time = MOSCA. In *18th IFAC/IFIP Workshop on Real-Time Programming — WRTP '92, Bruges*. Pergamon Press, June 1992.

[100] C Tofts. Temporal ordering for concurrency. Technical Report ECS-LFCS-88-49, Edinburgh University, April 1988.

[101] C Tofts. Timing concurrent processes. Technical Report ECS-LFCS-89-103, Edinburgh University, December 1989.

[102] A Valenzano, R Sisto, and L Ciminiera. Rapid prototyping of protocols from LOTOS specifications. *Software — Practice and Experience*, 23(1):31–54, January 1993.

[103] J Xu and D L Parnas. Scheduling processes with release times, deadlines, precedence and exclusion relations. *IEEE Transactions on Software Engineering*, 16(3):360–369, March 1990.

[104] W Yi. Real-time behaviour of asynchronous agents. In *CONCUR '90, Amsterdam, Lecture Notes in Computer Science 458*, pages 502–520. Springer-Verlag, 1990.

[105] S Yovine. *Méthodes et Outils pour la Vérification Symbolique de Systèmes Temporisés*. PhD thesis, Institut National Polytechnique de Grenoble, May 1993.

Appendix A

Proofs of Theorems

All theorems stated within the main body of the thesis are proved in this appendix. Lemmas which are of interest only within this appendix (i.e. those which are not quoted directly in the thesis) are stated and proved here.

Lemma 1 *For all regular S, all gate names a_i and expressions S_i, and all times t and t'*

$$t' \geq t \Longrightarrow Age([t]S, t') = Age(\sum_{i \in I} a_i.S_i \rhd^t S, t')$$

Proof

Immediate from definition of Age □

Lemma 2 *For all regular S and all times t_1 and t_2,*

$$Age(Age(S, t_1), t_2) = Age(S, t_1 + t_2)$$

Proof

By structural induction on S, where S is regular so that there are only three cases $S = \sum_{i \in I} a_i.S_i$ In this case

$$\begin{aligned} Age((Age(S, t_1), t_2) &= Age(Age(\sum_{i \in I} a_i.S_i, t_1), t_2) \\ &= Age(\sum_{i \in I} a_i.S_i, t_2) \\ &= \sum_{i \in I} a_i.S_i \\ &= Age(\sum_{i \in I} a_i.S_i, t_1 + t_2) \\ &= Age(S, t_1 + t_2) \end{aligned}$$

as required.

$S = [t]S'$. Here we resort to case analysis on t and t_1, with three cases

1. $t > t_1$ so $Age(Age(S, t_1), t_2) = Age([t\text{-}t_1]S', t_2)$ which leads us into a further case analysis on $t\text{-}t_1$ and t_2:
   (a) $t\text{-}t_1 > t_2$ so $Age([t\text{-}t_1]S', t_2) = [t\text{-}(t_1 + t_2)]S' = Age([t]S', t_1 + t_2)$
   (b) $t\text{-}t_1 = t_2$ so $Age([t\text{-}t_1]S', t_2) = S' = Age([t]S', t_1 + t_2)$ because $t = t_1 + t_2$
   (c) $t\text{-}t_1 < t_2$ so $Age([t\text{-}t_1]S', t_2) = Age(S', t_2\text{-}(t\text{-}t_1)) = Age([t]S', t_1 + t_2)$
2. $t = t_1$ so

$$\begin{aligned} Age(Age([t]S', t_1), t_2) &= Age(S', t_2) \\ &= Age([t_1]S', t_1 + t_2) \\ &= Age([t]S', t_1 + t_2) \end{aligned}$$

3. $t < t_1$ so

$$\begin{aligned} & Age(Age([t]S', t_1), t_2) \\ &= Age(Age(S', t_1\text{-}t), t_2) \\ &= Age(S', t_1\text{-}t + t_2) \text{ by induction hypothesis} \\ &= Age([t]S', t_1 + t_2) \text{ as } t_1 + t_2 > t \end{aligned}$$

$S = \sum_{i \in I} a_i.S_i \rhd^t S'$ proceeds by a similar argument to the case $S = [t]S'$, with a case split on t and t_1 and a sub-case split on $t\text{-}t_1$ and t_2.

1. $t > t_1$ so $Age(Age(S, t_1), t_2) = Age(\sum_{i \in I} a_i.S_i \rhd^{t\text{-}t_1} S', t_2)$ which leads us into a further case analysis on $t\text{-}t_1$ and t_2:
   (a) $t\text{-}t_1 > t_2$ so $Age(\sum_{i \in I} a_i.S_i \rhd^{t\text{-}t_1} S', t_2) = \sum_{i \in I} a_i.S_i \rhd^{t\text{-}(t_1+t_2)} S' = Age(\sum_{i \in I} a_i.S_i \rhd^t S', t_1 + t_2)$
   (b) $t\text{-}t_1 \leq t_2$. Follows from lemma 1 and cases 1(b) and 1(c) of $S = [t]S'$
2. $t \leq t_1$. Follows from lemma 1 and cases 2 and 3 of $S = [t]S'$

These are all of the cases for a regular sequential expression, so

$$Age(Age(S, t_1), t_2) = Age(S, t_1 + t_2)$$

as required. □

Theorem 1 *For all regular sequential expressions S and S', and all times t*

$$S \xrightarrow{(t)} S' \Leftrightarrow Age(S, t) = S'$$

where $=$ is syntactic identity on abstract syntax terms modulo equality on time expressions

Proof

($\Rightarrow$)

We need to show $S \xrightarrow{(t)} S' \Rightarrow Age(S,t) = S'$ and we do it by transition induction (on the depth of inference trees). There are six rules which can be used to deduce a time transition, five of which are base cases as they have no premises. The cases are

1. $$\frac{}{\sum_{i\in I} a_i.S_i \xrightarrow{(t)} \sum_{i\in I} a_i.S_i}$$

   so $S = S' = \sum_{i\in I} a_i.S_i$. But $Age(\sum_{i\in I} a_i.S_i, t) = \sum_{i\in I} a_i.S_i$.

2. $$\frac{}{[t]S \xrightarrow{(t')} [t\text{-}t']S}\ t' < t$$

   but $Age([t]S, t') = [t\text{-}t']S$ when $t < t'$.

3. $$\frac{}{[t]S \xrightarrow{(t)} S}$$

   but $Age([t]S, t) = S$.

4. $$\frac{}{\sum_{i\in I} a_i.S_i \rhd^t S \xrightarrow{(t)} \sum_{i\in I} a_i.S_i \rhd^{t\text{-}t'} S}\ t' < t$$

   but $Age(\sum_{i\in I} a_i.S_i \rhd^t S, t') = \sum_{i\in I} a_i.S_i \rhd^{t=t'} S$ when $t < t'$.

5. $$\frac{}{\sum_{i\in I} a_i.S_i \rhd^t S \xrightarrow{(t)} S}$$

   but $Age(\sum_{i\in I} a_i.S_i \rhd^t S, t) = S$

6. $$\frac{S_1 \xrightarrow{(t_1)} S_2 \quad S_2 \xrightarrow{(t_2)} S_3}{S_1 \xrightarrow{(t_1+t_2)} S_3}$$

   This is the only non-base case.

   The inductive hypotheses are that $Age(S_1, t_1) = S_2$ and $Age(S_2, t_2) = S_3$ and we need to prove $Age(S_1, t_1 + t_2) = S_3$. However

$Age(S_1, t_1 + t_2)$	$=$	$Age(Age(S_1, t_1), t_2)$	by lemma 2
	$=$	$Age(S_2, t_2)$	first inductive hypothesis
	$=$	S_3	second inductive hypothesis

This completes the $\Rightarrow$ half of the proof.

($\Leftarrow$)

We need to prove that $Age(S,t) = S' \Rightarrow S \xrightarrow{(t)} S'$ and we proceed by induction on the structure of S. As S is regular there are only three cases to consider:

$S = \sum_{i \in I} a_i.S_i$ This case is easy, as $Age(\sum_{i \in I} a_i.S_i, t) = \sum_{i \in I} a_i.S_i$ and we can deduce $\sum_{i \in I} a_i.S_i \xrightarrow{(t)} \sum_{i \in I} a_i.S_i$ immediately.

$S = [t']R$ We wish to prove that $[t']R \xrightarrow{(t)} Age([t']R, t)$, which calls for a case split on t and t'.

1. $t < t'$ gives $Age([t']R, t) = [t'\text{-}t]R$ which yields the result immediately
2. $t = t'$ gives $Age([t']R, t) = R$ again giving the result immediately
3. $t > t'$ gives $Age([t']R, t) = Age(R, t\text{-}t')$. Using the inductive hypothesis $Age(R, t\text{-}t') = S' \Rightarrow R \xrightarrow{(t\text{-}t')} S'$ and the transition $[t']R \xrightarrow{(t')} R$ we can apply the time transitivity rule

$$\frac{[t']R \xrightarrow{(t')} R \quad R \xrightarrow{(t\text{-}t')} S'}{[t']R \xrightarrow{(t'+t\text{-}t')} S'}$$

   i.e. $[t']R \xrightarrow{(t)} S'$ as required.

$S = \sum_{i \in I} a_i.S_i \rhd^{t'} R$ This case is proved in a very similar manner to the case $S = [t']R$, by case split on t and t'.

1. $t < t'$ gives $Age(\sum_{i \in I} a_i.S_i \rhd^{t'} R, t) = \sum_{i \in I} a_i.S_i \rhd^{t'\text{-}t} R$ which yields the result immediately
2. $t = t'$ gives $Age(\sum_{i \in I} a_i.S_i \rhd^{t'} R, t) = R$ again giving the result immediately
3. $t > t'$ gives $Age(\sum_{i \in I} a_i.S_i \rhd^{t'} R, t) = Age(R, t\text{-}t')$. Using the inductive hypothesis $Age(R, t\text{-}t') = S' \Rightarrow R \xrightarrow{(t\text{-}t')} S'$ and the transition $\sum_{i \in I} a_i.S_i \rhd^{t'} R \xrightarrow{(t')} R$ we can apply the time transitivity rule

$$\frac{\sum_{i \in I} a_i.S_i \rhd^{t'} R \xrightarrow{(t')} R \quad R \xrightarrow{(t\text{-}t')} S'}{\sum_{i \in I} a_i.S_i \rhd^{t'} R \xrightarrow{(t'+t\text{-}t')} S'}$$

   i.e. $\sum_{i \in I} a_i.S_i \rhd^{t'} R \xrightarrow{(t)} S'$ as required.

which completes the proof of the theorem.

□

Theorem 2 *For all AORTA systems, the resultant transition system has bounded variability.*

Proof

Let $\hat{t}$ be the smallest possible communication delay of any gate; $\hat{t} > 0$. Each process must then have a time transition of at least $\hat{t}$ between each action transition. However, each system action transition requires at least one process to have an action transition, so in a system with n processes, a run segment of duration $\hat{t}$ has at most n action transitions. Therefore, any run segement of duration t will have at most

$$\lceil \frac{t}{\hat{t}} \times n \rceil$$

action transitions, guaranteeing the bounded variability of the system. □

Theorem 3 *For all AORTA systems, the resultant transition system is time deterministic.*

Proof

From theorem 1 and the fact the Age is a function, it can be deduced that individual processes are time deterministic, as if $S_i \xrightarrow{t} S_i'$ and $S_i \xrightarrow{t} S_i''$ then $S_i' = Age(S_i, t) = S_i''$. However, all system time transitions depend on process time transitions by the rule

$$\frac{\forall i \in I . S_i \xrightarrow{(t)} S_i'}{\prod_{i \in I} S_i < K > \xrightarrow{(t)} \prod_{i \in I} S_i' < K >} \; \forall t' < t . \prod_{i \in I} Age(S_i, t') < K > \stackrel{\tau}{\not\longrightarrow}$$

and this is the only rule for time transitions, so all AORTA systems are time deterministic. □

Theorem 4 *For all AORTA systems, the resultant transition system has time additivity.*

Proof

We need to show that

$$\prod_{i \in I} S_i < K > \xrightarrow{(t_1)} \prod_{i \in I} S_i' < K > \; \wedge \prod_{i \in I} S_i' < K > \xrightarrow{(t_2)} \prod_{i \in I} S_i'' < K > \; \Rightarrow$$
$$\prod_{i \in I} S_i < K > \xrightarrow{(t_1 + t_2)} \prod_{i \in I} S_i'' < K >$$

From the transition rule for processes

$$\frac{S \xrightarrow{(t_1)} S' \quad S' \xrightarrow{(t_2)} S''}{S \xrightarrow{(t_1 + t_2)} S''}$$

it follows immediately that processes have time additivity. Hence for all processes S_i, S_i' and S_i'',

$$S_i \xrightarrow{(t_1)} S_i' \wedge S_i' \xrightarrow{(t_2)} S_i'' \Rightarrow S_i \xrightarrow{(t_1+t_2)} S_i''$$

Time additivity for systems follows from the rule for system time transitions, as long as the side condition

$$\forall t' < t_1 + t_2 . \prod_{i \in I} Age(S_i, t') < K > \stackrel{\tau}{\not\rightarrow} \ (*)$$

on the rule for system time transitions holds. From the hypothesis

$$\prod_{i \in I} S_i < K > \xrightarrow{(t_1)} \prod_{i \in I} S_i' < K >$$

we deduce from the side condition which held in the rule that derived the transition that

$$\forall t' < t_1 . \prod_{i \in I} Age(S_i, t') < K > \stackrel{\tau}{\not\rightarrow} \ (\dagger)$$

and from the hypothesis

$$\prod_{i \in I} S_i' < K > \xrightarrow{(t_2)} \prod_{i \in I} S_i'' < K >$$

we deduce that

$$\forall t' < t_2 . \prod_{i \in I} Age(S_i', t') < K > \stackrel{\tau}{\not\rightarrow}$$

which is equivalent to

$$\forall t_1 \leq t' < t_1 + t_2 . \prod_{i \in I} Age(S_i', t'\text{-}t_1) < K > \stackrel{\tau}{\not\rightarrow}$$

which is in turn equivalent to

$$\forall t_1 \leq t' < t_1 + t_2 . \prod_{i \in I} Age((Age(S_i, t_1), t'\text{-}t_1) < K > \stackrel{\tau}{\not\rightarrow}$$

by theorem 1, and the fact (from the first hypothesis and the rule for system time transitions) that $S_i \xrightarrow{(t_1)} S_i'$. Hence we deduce, by lemma 2, that

$$\forall t_1 \leq t' < t_1 + t_2 . \prod_{i \in I} Age(S_i, t') < K > \stackrel{\tau}{\not\rightarrow}$$

Combining this with (†), we deduce that the condition holds over the whole interval $[0, t_1 + t_2)$, satisfying the condition $(*)$ as required. □

Theorem 5 *For all AORTA systems, the resultant transition system has no temporal deadlock.*

Proof

We need to show that all AORTA systems have some transition. From the rules for sequential processes, it is easy to see that all sequential processes have a time transition, and some may also have action transitions. The rule for system time transitions will then yield a time transition for some time t unless the side-condition cannot be satisfied. However, if no time t can be found to satsify the side condition, then the must be a τ transition of the system at time 0, i.e. immediately. Hence every system has either a time transition or a τ transition. □

Lemma 3 *for all regular S,* $0 \in crucial(S)$

Proof

Direct from the definition. □

Lemma 4 *for all regular S,* $t \in crucial(S) \;\Rightarrow\; t \geq 0$

Proof

Direct from the definition. □

Lemma 5 *for all regular S,* $crucial([t]S) = crucial(\sum_{i \in I} a_i.S_i \rhd^t S)$

Proof

Direct from the definition. □

Definition 8

$$change(S, t') = \max(crucial(S) \cap [0, t'])$$

Lemma 6 *for all regular S and all t'*

$$Age(S, t') \xrightarrow{a} S' \;\Leftrightarrow\; Age(S, change(S, t')) \xrightarrow{a} S'$$

Proof

The proof is by induction on the structure of S, giving three cases

$S = \sum_{i \in I} a_i.S_i$. Here $Age(S, t') = \sum_{i \in I} a_i.S_i$ and

$$\begin{aligned} change(S, t') &= \max(crucial(S) \cap [0, t']) \\ &= \max(\{0\} \cap [0, t']) \\ &= \max(\{0\}) \\ &= 0 \end{aligned}$$

so $Age(S, change(S, t')) = Age(S, 0) = \sum_{i \in I} a_i.S_i = Age(S, t')$ which implies that

$$Age(S, t') \xrightarrow{a} S' \;\Leftrightarrow\; Age(S, change(S, t')) \xrightarrow{a} S'$$

as required.

$S = [t]R$. A case split on t and t' is required here

1. $t' < t$ gives $Age(S, t') = [t\text{-}t']R$ and

$$\begin{aligned} change(S, t') &= \max(crucial([t]R) \cap [0, t']) \\ &= \max((\{0\} \cup \{t'' + t \mid t'' \in crucial(R)\}) \cap [0, t']) \\ &= \max((\{0\} \cap [0, t']) \cup (\{t'' + t \mid t'' \in crucial(R)\} \cap [0, t'])) \\ &= \max(\{0\} \cup \{\,\}) \\ &\quad \text{as } t' < t, \text{ and by lemma 4} \\ &= 0 \end{aligned}$$

   so $Age(S, change(S, t')) = Age([t]R, 0) = [t]R$. This means that $Age(S, change(S, t')) \stackrel{a}{\not\longrightarrow}$, and also $Age(S, t') \stackrel{a}{\not\longrightarrow}$, so we get the result

$$Age(S, change(S, t')) \stackrel{a}{\longrightarrow} S' \;\Leftrightarrow\; Age(S, t') \stackrel{a}{\longrightarrow} S'$$

   as required

2. $t' = t$ gives $Age(S, t') = R$ and

$$\begin{aligned} change(S, t') &= change(S, t) \\ &= \max(crucial([t]R) \cap [0, t]) \\ &= \max((\{0\} \cup \{t'' + t \mid t'' \in crucial(R)\}) \cap [0, t]) \\ &= \max(\{0, t\}) \\ &= t \end{aligned}$$

   so $Age(S, change(S, t')) = Age([t]R, t) = R = Age(S, t')$ and hence

$$Age(S, change(S, t')) \stackrel{a}{\longrightarrow} S' \;\Leftrightarrow\; Age(S, t') \stackrel{a}{\longrightarrow} S'$$

3. $t' > t$ gives $Age(S, t') = Age(R, t'\text{-}t)$ and

$$\begin{aligned} change(S, t') &= \max(crucial([t]R) \cap [0, t']) \\ &= \max((\{0\} \cup \{t'' + t \mid t'' \in crucial(R)\}) \cap [0, t']) \\ &= \max(\{t'' + t \mid t'' \in crucial(R)\} \cap [0, t']) \\ &\quad \text{as } t' < t \text{ and } 0 \in crucial(R) \\ &= \max(\{t'' + t \mid t'' \in crucial(R) \wedge t'' + t \leq t'\}) \\ &= \max(\{t'' + t \mid t'' \in crucial(R) \wedge t'' \leq t'\text{-}t\}) \\ &= t + \max(\{t'' \mid t'' \in crucial(R)\} \cap [0, t'\text{-}t]) \\ &= t + \max(crucial(R) \cap [0, t'\text{-}t]) \\ &= t + change(R, t'\text{-}t) \end{aligned}$$

so

$$\begin{aligned} Age(S, change(S, t')) &= Age([t]R, t + change(R, t'\text{-}t)) \\ &= Age(Age(t[R], t), change(R, t'\text{-}t)) \\ &\quad \text{by lemma 2} \\ &= Age(R, change(R, t'\text{-}t)) \end{aligned}$$

but by the inductive hypothesis

$$Age(R, t'\text{-}t) \xrightarrow{a} S' \;\Leftrightarrow\; Age(R, change(R, t'\text{-}t)) \xrightarrow{a} S'$$

which gives

$$Age(S, t') \xrightarrow{a} S' \;\Leftrightarrow\; Age(S, change(S, t')) \xrightarrow{a} S'$$

and completes the case $S = [t]R$

$S = \sum_{i \in I} a_i.S_i \triangleright^t R$. Again a case split is needed on t' and t:

1. $t' < t$ gives $Age(S, t') = \sum_{i \in I} a_i.S_i \triangleright^{t\text{-}t'} R$ and
   $change(S, t') = change([t]R, t') = 0$ (using lemma 5), so

   $$Age(S, change(S, t')) = Age(S, 0) = \sum_{i \in I} a_i.S_i \triangleright^t R$$

   However, the action rules for timeouts do not depend on the value of the timeout, so

   $$\sum_{i \in I} a_i.S_i \triangleright^t R \xrightarrow{a} S' \;\Leftrightarrow\; \sum_{i \in I} a_i.S_i \triangleright^{t\text{-}t'} R \xrightarrow{a} S'$$

   i.e.

   $$Age(S, t') \xrightarrow{a} S' \;\Leftrightarrow\; Age(S, change(S, t')) \xrightarrow{a} S'$$

2. $t' \geq t$. In this case (these cases), by applying lemmas 5 and 1 the proof is exactly the same as for the case $S = [t]R$, which completes the proof.

□

Lemma 7

$$(\forall i \in I.R_i \xrightarrow{a} S_i' \;\Leftrightarrow\; S_i \xrightarrow{a} S_i') \;\Rightarrow\; (\prod_{i \in I} R_i < K > \stackrel{\tau}{\not\longrightarrow} \;\Leftrightarrow\; \prod_{i \in I} S_i < K > \stackrel{\tau}{\not\longrightarrow})$$

Proof

Direct from transition rule for systems. □

Theorem 6 *for all regular* S_i *and all* $t > 0$

$$(\forall t' < t. \prod_{i \in I} Age(S_i, t') < K > \stackrel{\tau}{\not\rightarrow}) \;\Leftrightarrow$$

$$(\forall t' \in (\bigcup_{i \in I} crucial(S_i) \cap [0, t)\ ). \prod_{i \in I} Age(S_i, t') < K > \stackrel{\tau}{\not\rightarrow})$$

Proof

($\Rightarrow$)

As all of the t' of the RHS are less than t, the result is immediate.

($\Leftarrow$)

By induction on the size of $\bigcup_{i \in I} crucial(S_i) \cap [0, t)$.

Base case: $|\bigcup_{i \in I} crucial(S_i) \cap [0, t)| = 1$

As $0 \in crucial(S_i)$ (lemma 3) we have $0 \in \bigcup_{i \in I} crucial(S_i)$ which gives

$$\bigcup_{i \in I} crucial(S_i) \cap [0, t) = \{0\}$$

and hence

$$crucial(S_i) \cap [0, t) = \{0\}$$

for all i. Then for all $t' < t$ and for all i

$$\begin{aligned} change(S_i, t') &= \max(crucial(S_i) \cap [0, t']) \\ &= \max(crucial(S_i) \cap [0, t) \cap [0, t']) \\ &\quad \text{as } t' < t \\ &= \max(\{0\} \cap [0, t']) \\ &= 0 \end{aligned}$$

so by lemma 6

$$\forall t' < t. \forall i. Age(S_i, t') \stackrel{a}{\longrightarrow} S_i' \;\Leftrightarrow\; Age(S_i, 0) \stackrel{a}{\longrightarrow} S_i'$$

and hence by lemma 7

$$(\forall t' < t. \prod_{i \in I} Age(S_i, t') < K > \stackrel{\tau}{\not\rightarrow}) \;\Leftrightarrow\; (\prod_{i \in I} Age(S_i, 0) < K > \stackrel{\tau}{\not\rightarrow})$$

As $\bigcup_{i \in I} crucial(S_i) \cap [0, t) = \{0\}$, this proves the result in the base case.

In the induction case we assume the result for sets of size n and deduce the result for the case

$$|\bigcup_{i \in I} crucial(S_i) \cap [0, t)| = n + 1$$

If we define

$$\hat{t} = \max(\{\bigcup_{i \in I} crucial(S_i) \cap [0, t)\})$$

so that

$$|\bigcup_{i \in I} crucial(S_i) \cap [0, \hat{t}) \,| = n$$

the induction hypothesis tells us that

$$(\forall t' \in \bigcup_{i \in I} crucial(S_i) \cap [0, \hat{t}).\prod_{i \in I} Age(S_i, t') < K > \stackrel{\tau}{\not\rightarrow}) \;\Rightarrow$$

$$(\forall t' < \hat{t}.\prod_{i \in I} Age(S_i, t') < K > \stackrel{\tau}{\not\rightarrow})$$

With this, it is sufficient to prove

$$(\forall t' \in \bigcup_{i \in I} crucial(S_i) \cap [0, t).\prod_{i \in I} Age(S_i, t') < K > \stackrel{\tau}{\not\rightarrow}) \;\Rightarrow$$

$$(\forall \hat{t} \leq t' < t.\prod_{i \in I} Age(S_i, t') < K > \stackrel{\tau}{\not\rightarrow})$$

or in particular that

$$\prod_{i \in I} Age(S_i, \max(\bigcup_{i' \in I} crucial(S_{i'}) \cap [0, t)\,)) < K > \stackrel{\tau}{\not\rightarrow} \;\Rightarrow$$

$$(\forall \hat{t} \leq t' < t.\prod_{i \in I} Age(S_i, t') < K > \stackrel{\tau}{\not\rightarrow})$$

which is equivalent to

$$\prod_{i \in I} Age(S_i, \hat{t}) < K > \stackrel{\tau}{\not\rightarrow} \;\Rightarrow$$

$$(\forall \hat{t} \leq t' < t.\prod_{i \in I} Age(S_i, t') < K > \stackrel{\tau}{\not\rightarrow})$$

by the definition of $\hat{t}$. In order to prove this we make the following claim:

Claim

$\hat{t} \leq t' < t \;\Rightarrow\; \forall i \in I.change(S_i, t') = change(S_i, \hat{t})$

Proof

To prove the result, we show that

$$change(S_i, t') \geq change(S_i, \hat{t})$$

and

$$change(S_i, t') \leq change(S_i, \hat{t})$$

Firstly

$$\begin{aligned} change(S_i, t') &= \max(crucial(S_i) \cap [0, t']) \\ &\geq \max(crucial(S_i) \cap [0, \hat{t}]) \\ &\quad \text{as } t' \geq \hat{t} \\ &= change(S_i, \hat{t}) \end{aligned}$$

and secondly

$$\begin{aligned} change(S_i, t') &= \max(crucial(S_i) \cap [0, t']) \\ &\leq \max(crucial(S_i) \cap [0, t)\ ) \\ &\quad \text{as } t' < t \\ &= \max(crucial(S_i) \cap [0, \hat{t}]) \\ &\quad \text{otherwise } \hat{t} \text{ is not a maximum} \\ &= change(S_i, \hat{t}) \end{aligned}$$

which proves the claim. □

Combining the claim with lemma 6, we get

$$\forall i \in I. \forall \hat{t} \leq t' < t. Age(S_i, t') \xrightarrow{a} S' \ \Leftrightarrow \ Age(S_i, change(S_i, \hat{t})) \xrightarrow{a} S'$$

which gives the result by an application of lemma 7. □

Lemma 8 *The translation function bar from the data enriched language to the pure language*

$$bar(S) = \overline{S}$$

(defined on page 101) is surjective.

Proof

Simple. All constructs in the pure language have parallels in the data enriched language. □

Lemma 9 *For all regular expressions S of the data enriched language, and all data states Φ*

$$\overline{Poss_{DATA}(S, \Phi)} \subseteq Poss_{PURE}(\overline{S})$$

where $Poss_{DATA}$ is the regularising function for the data language (defined on page 99), $Poss_{PURE}$ is the regularising function for the pure language (defined on page 27), and $\overline{X} \triangleq \{\overline{x} \mid x \in X\}$.

Proof

By induction on the structure of S

$S = \sum_{i \in I} a_i?A_i!B_i.S_i$ Then

$$\begin{aligned}
\overline{Poss_{DATA}(S, \Phi)} &= \overline{\{\sum_{i \in I} a_i?A_i!B_i.S_i\}} \\
&= \{\sum_{i \in I} a_i.\overline{S_i}\} \\
&= Poss_{PURE}(\sum_{i \in I} a_i.\overline{S_i}) \\
&= Poss_{PURE}(\overline{S})
\end{aligned}$$

$S = [t\{\Delta\}]S'$ Then

$$\begin{aligned}
\overline{Poss_{DATA}(S, \Phi)} &= \overline{\{[t\{\Delta\}]S'' \mid S'' \in Poss_{DATA}(S', \Delta(\Phi_{+t}))\}} \\
&= \{\overline{[t\{\Delta\}]S''} \mid S'' \in Poss_{DATA}(S', \Delta(\Phi_{+t}))\} \\
&= \{[t]\overline{S''} \mid S'' \in Poss_{DATA}(S', \Delta(\Phi_{+t}))\} \\
&\subseteq \{[t]\overline{S''} \mid \overline{S''} \in Poss_{PURE}(\overline{S'})\} \text{ by inductive hypothesis} \\
&= \{[t]S''' \mid S''' \in Poss_{PURE}(\overline{S'})\} \text{ by lemma 8} \\
&= Poss_{PURE}([t]\overline{S'}) \\
&= Poss_{PURE}(\overline{[t\{\Delta\}]S'}) \\
&= Poss_{PURE}(\overline{S})
\end{aligned}$$

$S = \sum_{i \in I} a_i?A_i!B_i.S_i \triangleright^t S'$ Then

$$\begin{aligned}
\overline{Poss_{DATA}(S, \Phi)} &= \overline{\{\sum_{i \in I} a_i?A_i!B_i.S_i \triangleright^t S'' \mid S'' \in Poss_{DATA}(S', \Phi_{+t})\}} \\
&= \{\sum_{i \in I} a_i.\overline{S_i} \triangleright^t \overline{S''} \mid S'' \in Poss_{DATA}(S', \Phi_{+t})\} \\
&\subseteq \{\sum_{i \in I} a_i.\overline{S_i} \triangleright^t \overline{S''} \mid \overline{S''} \in Poss_{PURE}(\overline{S'})\} \text{ by inductive hypothesis} \\
&= \{\sum_{i \in I} a_i.\overline{S_i} \triangleright^t S''' \mid S''' \in Poss_{PURE}(\overline{S'})\} \text{ by lemma 8} \\
&= Poss_{PURE}(\sum_{i \in I} a_i.\overline{S_i} \triangleright^t \overline{S'}) \\
&= Poss_{PURE}(\overline{\sum_{i \in I} a_i?A_i!B_i.S_i \triangleright^t S'}) \\
&= Poss_{PURE}(\overline{S})
\end{aligned}$$

$S = [t_1, t_2\{\Delta\}]S'$

$$
\begin{aligned}
\overline{Poss_{DATA}(S, \Phi)} &= \overline{\{[t\{\Delta\}]S'' \mid t \in [t_1, t_2], S'' \in Poss_{DATA}(S', \Delta(\Phi_{+t}))\}} \\
&= \{\overline{[t\{\Delta\}]S''} \mid t \in [t_1, t_2], S'' \in Poss_{DATA}(S', \Delta(\Phi_{+t}))\} \\
&= \{[t]\overline{S''} \mid t \in [t_1, t_2], S'' \in Poss_{DATA}(S', \Delta(\Phi_{+t}))\} \\
&\subseteq \{[t]\overline{S''} \mid t \in t_1, t_2, \overline{S''} \in Poss_{PURE}(\overline{S'})\} \text{ by inductive hypothesis} \\
&= \{[t]S''' \mid t \in [t_1, t_2], S''' \in Poss_{PURE}(\overline{S'})\} \text{ by lemma 8} \\
&= Poss_{PURE}([t_1, t_2]\overline{S'}) \\
&= Poss_{PURE}(\overline{[t_1, t_2\{\Delta\}]S'}) \\
&= Poss_{PURE}(\overline{S})
\end{aligned}
$$

$S = \sum_{i \in I} a_i?A_i!B_i.S_i \rhd_{t_1}^{t_2} S'$

$$
\begin{aligned}
\overline{Poss_{DATA}(S, \Phi)} &= \overline{\{\sum_{i \in I} a_i?A_i!B_i.S_i \rhd^t S'' \mid t \in [t_1, t_2], S'' \in Poss_{DATA}(S', \Phi_{+t})\}} \\
&= \{\sum_{i \in I} a_i.\overline{S_i} \rhd^t \overline{S''} \mid t \in [t_1, t_2], S'' \in Poss_{DATA}(S', \Phi_{+t})\} \\
&\subseteq \{\sum_{i \in I} a_i.\overline{S_i} \rhd^t \overline{S''} \mid t \in [t_1, t_2], \overline{S''} \in Poss_{PURE}(\overline{S'})\} \\
&\quad \text{by inductive hypothesis} \\
&= \{\sum_{i \in I} a_i.\overline{S_i} \rhd^t S''' \mid t \in [t_1, t_2], S''' \in Poss_{PURE}(\overline{S'})\} \text{ by lemma 8} \\
&= Poss_{PURE}(\sum_{i \in I} a_i.\overline{S_i} \rhd_{t_1}^{t_2} \overline{S'}) \\
&= Poss_{PURE}(\overline{\sum_{i \in I} a_i?A_i!B_i.S_i \rhd_{t_1}^{t_2} S'}) \\
&= Poss_{PURE}(\overline{S})
\end{aligned}
$$

$S = \bigoplus_{i \in I} S_i\{p_i\}$

$$
\begin{aligned}
\overline{Poss_{DATA}(S, \Phi)} &= \overline{\{S_i' \mid i \in I, p_i(\Phi), S_i' \in Poss_{DATA}(S_i, \Phi)\}} \\
&= \{\overline{S_i'} \mid i \in I, p_i(\Phi), S_i' \in Poss_{DATA}(S_i, \Phi)\} \\
&\subseteq \{\overline{S_i'} \mid i \in I, S_i' \in Poss_{DATA}(S_i, \Phi)\} \\
&\subseteq \{\overline{S_i'} \mid i \in I, \overline{S_i'} \in Poss_{PURE}(\overline{S_i})\} \text{ by inductive hypothesis} \\
&= \{S_i' \mid i \in I, S_i' \in Poss_{PURE}(\overline{S_i})\} \text{ by lemma 8} \\
&= Poss_{PURE}(\bigoplus_{i \in I} \overline{S_i}) \\
&= Poss_{PURE}(\overline{\bigoplus_{i \in I} S_i\{p_i\}}) \\
&= Poss_{PURE}(\overline{S})
\end{aligned}
$$

$S = X$ **and** $X \triangleq S'$

$$\begin{aligned}
\overline{Poss_{DATA}(S, \Phi)} &= \overline{Poss_{DATA}(S', \Phi)} \\
&\subseteq Poss_{PURE}(\overline{S'}) \text{ using least fixed point property} \\
&= Poss_{PURE}(X) \text{ as } X \triangleq \overline{S'} \\
&= Poss_{PURE}(\overline{X}) \\
&= Poss_{PURE}(\overline{S})
\end{aligned}$$

which completes the proof. □

Lemma 10 *For all regular sequential expressions S and S' of the data enriched language, and all data states Φ and Φ'*

$$S[\![\Phi]\!] \xrightarrow{\alpha}_{DATA} S'[\![\Phi']\!] \;\Rightarrow\; \overline{S} \xrightarrow{\overline{\alpha}}_{PURE} \overline{S'}$$

where $\longrightarrow_{DATA}$ is the transition relation for the language with data, and $\longrightarrow_{PURE}$ is the transition relation for the language without data.

Proof

By induction on the depth of inference trees of the transition $S[\![\Phi]\!] \xrightarrow{\alpha}_{DATA} S'[\![\Phi']\!]$. There are 8 rules to consider:

1. For the rule
$$\frac{}{[t\{\Delta\}]S[\![\Phi]\!] \xrightarrow{(t')}_{DATA} [t\text{-}t'\{\Delta\}]S[\![\Phi_{+t'}]\!]}\; t' < t$$

   we need to show that
$$t' < t \;\Rightarrow\; \overline{[t\{\Delta\}]S} \xrightarrow{\overline{(t')}}_{PURE} \overline{[t\text{-}t'\{\Delta\}]S}$$

   or equivalently that
$$t' < t \;\Rightarrow\; [t]\overline{S} \xrightarrow{(t')}_{PURE} [t\text{-}t']\overline{S}$$

   which can be deduced immediately from the rule
$$\frac{}{[t]S \xrightarrow{(t')}_{PURE} [t\text{-}t']S}\; t' < t$$

2. For the rule
$$\frac{}{[t\{\Delta\}]S[\![\Phi]\!] \xrightarrow{(t)}_{DATA} S[\![\Delta(\Phi_{+t})]\!]}$$

we need to show that

$$[t]\overline{S} \xrightarrow{(t)}_{PURE} \overline{S}$$

which is immediate from the rule

$$\frac{}{[t]S \xrightarrow{(t)}_{PURE} S}$$

3. For the rule

$$\frac{}{\sum_{i\in I} a_i?A_i!B_i.S_i[\![\Phi]\!] \xrightarrow{(t)}_{DATA} \sum_{i\in I} a_i?A_i!B_i.S_i[\![\Phi_{+t}]\!]}$$

we need to show that

$$\sum_{i\in I} a_i.\overline{S_i} \xrightarrow{(t)}_{PURE} \sum_{i\in I} a_i.\overline{S_i}$$

which is immediate from the rule

$$\frac{}{\sum_{i\in I} a_i.S_i \xrightarrow{(t)}_{PURE} \sum_{i\in I} a_i.S_i}$$

4. For the rule

$$\frac{}{\sum_{i\in I} a_i?A_i!B_i.S_i[\![\Phi]\!] \xrightarrow{a_j?v!\Phi.B_j}_{DATA} [t\{\Xi\}]S'_j[\![\Phi[A_j=v]]\!]} \quad \begin{array}{l} j \in I \\ S'_j \in Poss_{DATA}(S_j, \Phi[A_j=v]) \\ t \in delays(a_j) \\ v \in values(A_j) \end{array}$$

we need to show that

$$j \in I, S'_j \in Poss_{DATA}(S_j, \Phi[A_j=v]), t \in delays(a_j) \;\Rightarrow\; \sum_{i\in I} a_i.\overline{S_i} \xrightarrow{a_j}_{PURE} [t]\overline{S'_j}$$

which is immediate from lemma 9 and the rule

$$\frac{}{\sum_{i\in I} a_i.S_i \xrightarrow{a_j} [t]S'_j} \quad \begin{array}{l} j \in I, S'_j \in Poss_{PURE}(S_j) \\ t \in delays(a_j) \end{array}$$

5. For the rule

$$\frac{}{\sum_{i\in I} a_i?A_i!B_i.S_i \rhd^t S[\![\Phi]\!] \xrightarrow{(t')}_{DATA} \sum_{i\in I} a_i?A_i!B_i.S_i \rhd^{t-t'} S[\![\Phi_{+t'}]\!]} \; t' < t$$

we need to show that

$$t' < t \;\Rightarrow\; \sum i \in I \overline{S_i} \rhd^t \overline{S} \xrightarrow{(t')}_{PURE} \sum i \in I \overline{S_i} \rhd^{t-t'} \overline{S}$$

which is immediate from the rule

$$\frac{}{\sum_{i\in I} a_i.S_i \rhd^t S \xrightarrow{(t')}_{PURE} \sum_{i\in I} a_i.S_i \rhd^{t-t'} S} \; t' < t$$

6. For the rule
$$\frac{}{\sum_{i\in I} a_i?A_i!B_i.S_i \rhd^t S[\![\Phi]\!] \xrightarrow{(t)}_{DATA} S[\![\Phi_{+t}]\!]}$$
we need to show that
$$\sum_{i\in I} a_i.\overline{S_i} \rhd^t \overline{S} \xrightarrow{(t)}_{PURE} \overline{S}$$
which is immediate from the rule
$$\frac{}{\sum_{i\in I} a_i.S_i \rhd^t S \xrightarrow{(t)}_{PURE} S}$$

7. For the rule
$$\frac{}{\sum_{i\in I} a_i?A_i!B_i.S_i \rhd^t S[\![\Phi]\!] \xrightarrow{a_j?v!\Phi.B_j}_{DATA} [t\{\Xi\}]S'_j[\![\Phi[A_j=v]]\!]} \quad \begin{array}{l} j\in I \\ S'_j \in Poss_{DATA}(S_j, \Phi[A_j=v]) \\ t \in delays(a_j) \\ v \in values(A_j) \end{array}$$
we need to show
$$j\in I, S'_j \in Poss_{DATA}(S_j, \Phi[A_j=v]), t\in delays(a_j) \;\Rightarrow\; \sum_{i\in I} a_i.\overline{S_i} \rhd^t \overline{S} \xrightarrow{a_j}_{PURE} [t]\overline{S'_j}$$
which is immediate from lemma 9 and the rule
$$\frac{}{\sum_{i\in I} a_i.S_i \rhd^t S \xrightarrow{a_j}_{PURE} [t']S'_j} \quad \begin{array}{l} j\in I, S'_j \in Poss_{PURE}(S_j) \\ t' \in delays(a_j) \end{array}$$

8. For the rule
$$\frac{S[\![\Phi_1]\!] \xrightarrow{(t_1)}_{DATA} S'[\![\Phi_2]\!] \quad S'[\![\Phi_2]\!] \xrightarrow{(t_2)}_{DATA} S''[\![\Phi_3]\!]}{S[\![\Phi_1]\!] \xrightarrow{(t_1+t_2)}_{DATA} S''[\![\Phi_3]\!]}$$
by applying the inductive hypothesis, we need to show that
$$\overline{S} \xrightarrow{(t_1)}_{PURE} \overline{S'} \wedge \overline{S'} \xrightarrow{(t_2)}_{PURE} \overline{S''} \;\Rightarrow\; \overline{S} \xrightarrow{(t_1+t_2)}_{PURE} \overline{S''}$$
which is immediate from the rule
$$\frac{S \xrightarrow{(t_1)} S' \quad S' \xrightarrow{(t_2)} S''}{S \xrightarrow{(t_1+t_2)} S''}$$

□

Theorem 7 *For all regular sequential expressions S_i and S'_i of the data enriched language, and all data states Φ_i and Φ'_i*

1.

$$\prod_{i\in I} S_i[\![\Phi_i]\!] < K > \xrightarrow{\tau}_{DATA} \prod_{i\in I} S_i'[\![\Phi_i']\!] < K > \;\Rightarrow$$

$$\prod_{i\in I} \overline{S_i} < K > \xrightarrow{\tau}_{PURE} \prod_{i\in I} \overline{S_i'} < K >$$

2.

$$\prod_{i\in I} S_i[\![\Phi_i]\!] < K > \xrightarrow{a}_{DATA} \prod_{i\in I} S_i'[\![\Phi_i']\!] < K > \;\wedge$$

$$\prod_{i\in I} \overline{S_i} < K > \stackrel{\tau}{\not\rightarrow}_{PURE} \;\Rightarrow$$

$$\prod_{i\in I} \overline{S_i} < K > \xrightarrow{\overline{a}}_{PURE} \prod_{i\in I} \overline{S_i'} < K >$$

3.

$$\prod_{i\in I} S_i[\![\Phi_i]\!] < K > \xrightarrow{(t)}_{DATA} \prod_{i\in I} S_i'[\![\Phi_i']\!] < K > \;\wedge$$

$$\forall t' < t. \prod_{i\in I} Age(\overline{S_i}, t') < K > \stackrel{\tau}{\not\rightarrow}_{PURE} \;\Rightarrow$$

$$\prod_{i\in I} \overline{S_i} < K > \xrightarrow{(t)}_{PURE} \prod_{i\in I} \overline{S_i'} < K >$$

where $\longrightarrow_{DATA}$ is the transition relation for the language with data, and $\longrightarrow_{PURE}$ is the transition relation for the language without data.

Proof

The three cases correspond to the three rules which can be used to deduce a system transition, so the proof is by induction on the transition rule applied.

1.

$$\frac{S_j[\![\Phi_j]\!] \xrightarrow{a?u!v}_{DATA} S_j'[\![\Phi_j']\!] \quad S_k[\![\Phi_k]\!] \xrightarrow{b?v!u}_{DATA} S_k'[\![\Phi_k']\!]}{\prod_{i\in I} S_i[\![\Phi_i]\!] < K > \xrightarrow{\tau}_{DATA} \prod_{i\in I} S_i'[\![\Phi_i']\!] < K >} \quad \begin{array}{l}(j.a, k.b) \in K \\ S_i' = S_i \text{ if } i \neq j, k \\ \Phi_i' = \Phi_i \text{ if } i \neq j, k\end{array}$$

here it is sufficient to show that
$S_j[\![\Phi_j]\!] \xrightarrow{a?u!v}_{DATA} S_j'[\![\Phi_j']\!] \wedge$
$S_k[\![\Phi_k]\!] \xrightarrow{b?v!u}_{DATA} S_k'[\![\Phi_k']\!] \wedge$
$(j.a, k.b) \in K, S_i' = S_i \text{ if } i \neq j, k \;\Rightarrow$
$\prod_{i\in I} \overline{S_i} < K > \xrightarrow{\tau}_{PURE} \prod_{i\in I} \overline{S_i'} < K >$
which is immediate from lemma 10 and the rule

$$\frac{S_j \xrightarrow{a} S_j' \quad S_k \xrightarrow{b} S_k'}{\prod_{i\in I} S_i < K > \xrightarrow{\tau}_{PURE} \prod_{i\in I} S_i' < K >} \quad \begin{array}{l}(j.a, k.b) \in K \\ S_i' = S_i \text{ if } i \neq j, k\end{array}$$

2.

$$\frac{S_j[\![\Phi_j]\!] \xrightarrow{a?u!v}_{DATA} S'_j[\![\Phi'_j]\!]}{\prod_{i\in I} S_i[\![\Phi_i]\!] < K > \xrightarrow{a?u!v}_{DATA} \prod_{i\in I} S'_i[\![\Phi'_i]\!] < K >} \quad \begin{array}{l} j \in I \\ (j.a, \_) \notin K \\ S'_i = S_i \text{ if } i \neq j \\ \Phi'_i = \Phi_i \text{ if } i \neq j \\ \prod_{i\in I} S_i[\![\Phi_i]\!] < K > \overset{\tau}{\not\longrightarrow}_{DATA} \end{array}$$

here it is sufficient to show that
$S_j[\![\Phi_j]\!] \xrightarrow{a?u!v}_{DATA} S'_j[\![\Phi'_j]\!] \wedge$
$j \in I, (j.a, \_) \notin K, S'_i = S_i \text{ if } i \neq j \Rightarrow$
$\prod_{i\in I} \overline{S_i} < K > \xrightarrow{\overline{a}}_{PURE} \prod_{i\in I} \overline{S'_i} < K >$
which follows from lemma 10, the rule

$$\frac{S_j \xrightarrow{a}_{PURE} S'_j}{\prod_{i\in I} S_i < K > \xrightarrow{a} \prod_{i\in I} S'_i < K >} \quad \begin{array}{l} j \in I \\ (j.a, \_) \notin K \\ S'_i = S_i \text{ if } i \neq j \\ \prod_{i\in I} S_i < K > \overset{\tau}{\not\longrightarrow}_{PURE} \end{array}$$

and the hypothesis

$$\prod_{i\in I} \overline{S_i} < K > \overset{\tau}{\not\longrightarrow}_{PURE}$$

3.

$$\frac{\forall i \in I. S_i[\![\Phi_i]\!] \xrightarrow{(t)}_{DATA} S'_i[\![\Phi'_i]\!]}{\prod_{i\in I} S_i[\![\Phi_i]\!] < K > \xrightarrow{(t)}_{DATA} \prod_{i\in I} S'_i[\![\Phi'_i]\!] < K >} \quad \forall t' < t. \prod_{i\in I} Age(S_i[\![\Phi_i]\!], t') < K > \overset{\tau}{\not\longrightarrow}_{DATA}$$

here it is sufficient to show that
$\forall i \in I. S_i[\![\Phi_i]\!] \xrightarrow{(t)}_{DATA} S'_i[\![\Phi'_i]\!] \Rightarrow$
$\prod_{i\in I} \overline{S_i} < K > \xrightarrow{(t)}_{PURE} \prod_{i\in I} \overline{S'_i} < K >$
which follows from lemma 10, the rule

$$\frac{\forall i \in I. S_i \xrightarrow{(t)}_{PURE} S'_i}{\prod_{i\in I} S_i < K > \xrightarrow{(t)}_{PURE} \prod_{i\in I} S'_i < K >} \quad \forall t' < t. \prod_{i\in I} Age(S_i, t') < K > \overset{\tau}{\not\longrightarrow}_{PURE}$$

and the hypothesis

$$\forall t' < t. \prod_{i\in I} Age(\overline{S_i}, t') < K > \overset{\tau}{\not\longrightarrow}_{PURE}$$

□

Appendix B

Published Work

This appendix contains all published work based on the content of this thesis.
The papers are (in chronological order of publication):

- Practical Formal Development of Real-Time Systems. Appeared in the proceedings of *11th IEEE Workshop on Real-Time Operating Systems and Software, RTOSS '94, Seattle (ed. A Shaw)* May 1994, pp44-48.

- Application Oriented Real-Time Algebra. Appeared in *Software Engineering Journal* 9(5) September 1994, pp201-202.

- Designing and Implementing Correct Real-Time Systems. Appeared in the proceedings of *Formal Techniques in Real-Time and Fault-Tolerant Systems FTRTFT '94, Lubeck, Lecture Notes in Computer Science 863 (ed. H Langmaack, W-P de Roever and J Vytopil)* September 1994, pp228-246.

- A Formally Based Hard Real-Time Kernel. Appeared in *Microprocessors and Microsystems* 18(9) November 1994, pp513-521.

- Validation, Verification and Implementation of Timed Protocols using AORTA. Appeared in proceedings of *Protocol Specification, Testing and Verification XV, PSTV '95, Warsaw (ed. P Dembinski and M Sredniawa)* June 1995, pp193-208.