University of Newcastle upon Tyne
School of Computing Science

Formal Modelling and Analysis of an Asynchronous Communication Mechanism

by
Neil Henderson

PhD Thesis

February 1, 2005

Abstract

This thesis makes a contribution towards cutting the cost of development of real-time systems. The development of real-time systems is difficult: often errors in the specification are not identified until late in the development process, and there is a requirement to reduce the amount of rework to correct flaws introduced in the early stages of development. A Real-time Network-Specification Language (RTN-SL) is being developed to allow the rigorous specification of functionality and timing properties of computations. The correct specification of end to end timing constraints, however, requires an understanding of the timing properties of the communications between components. A theory of communication is therefore required, to be used with the RTN-SL, to analyse timing properties of systems early in the development process.

The work demonstrates how a tool set can be used to gain an understanding of the behaviour of the system, to help to identify and correct ambiguities that arise in the early stages of development. An incremental development approach is recommended. Starting with an abstract model and exploring properties of increasingly realistic models of the implementation, to gain confidence about the correctness of the implementation, and an understanding its behaviour. The strengths and weaknesses of a number of tools are discussed and it is shown that it is possible to use a compositional rely-guarantee method to verify properties of systems where the individual components give few or no guarantees about their behaviour. This rely guarantee method makes it possible to record assumptions in the specification, to help ensure they are not overlooked and thereby introduce errors in the design and implementation. This approach can form the basis of a theory of communication, which can be used with the RTN-SL to reason about end to end timing properties of systems in the early stages of development.

Acknowledgements

This thesis would never have been completed without the generous and invaluable help received from many people and organisations during the course of study.

I would like to express my gratitude to my Supervisors at the University of Newcastle: Professor Cliff Jones for his invaluable insight, which helped to guide the work, and Dr John Fitzgerald for encouragement and assistance during the early work and his patience and moral support while I was completing this thesis.

I am deeply indebted to Dr Stephen Paynter of MBDA UK Limited and Dr Jim Armstrong of the University of Newcastle, for their guidance and assistance in helping me to understand the technicalities of automated theory proving, and in particular PVS. In addition Stephen provided help, encouragement and moral support while the work was progressing.

Thanks are also due to Professor Hugo Simpson of MBDA UK Limited, and many colleagues at the University of Newcastle, including Professors Alex Yakovlev and Majiec Koutny and Drs Ian Clark and Fei Xia who have commented on various aspects of the work as it has progressed.

Thanks must go to the BAE SYSTEMS Dependable Computing Systems Centre for funding, and providing time, to complete the work.

I am deeply grateful to my family, in particular my wife Alison, for their support and encouragement during yet another period of study.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis makes a contribution towards cutting the cost of development of asynchronous real real-time systems, by demonstrating how it is possible to gain an understanding, and verify properties, of such systems in an incremental manner. It recommends starting with an abstract, easy to understand, model of the required behaviour of the system, and building and verifying more realistic models as understanding increases. It also shows how it is possible to verify properties of systems using a compositional rely-guarantee method, when the individual components of the system give few or no guarantees about their individual behaviour. The work has been sponsored by the BAE SYSTEMS Dependable Computing Systems Centre (DCSC) and in particular MBDA UK Limited.

The specification and development of asynchronous real-time systems is difficult, and often errors that arise from a lack of understanding of the specifications of these systems are not identified until late in the development process. The development of relatively small fully asynchronous systems, which have apparently simple specifications, may also be difficult because their components can interact in unexpected ways. Correcting errors may require a large amount of rework, because, depending on the stage in the development process at which the error was introduced, this may require the specification, design and implementation to be modified and verification and testing work may need to be repeated for the modified system. There is therefore a requirement to identify and correct flaws and ambiguities earlier in the development process to reduce the amount of rework that is required, in order to cut the cost of, and time for, developing those systems. A classical method of dealing with complexity is to specify the system as a number of simpler components [Kop98]. There is then an obligation to verify that the complete system meets its specification, when it is composed of those components. Formal models of systems can aid the analysis of requirements and the use of formal methods makes it possible to verify the behaviour of

the system in a rigorous manner. This analysis can help to expose errors and ambiguities in the requirements and specification of the system, and identify ways of correcting those errors.

A (formal) Real-time Network-Specification Language (RTN-SL) [PAH00, Pay02] is being developed jointly by the DCSC and MBDA UK Limited, based on VDM-SL [ISO96] and Real-time Logic (RTL) [JM86, JMS88], to allow the rigorous specification of functionality and timing properties of computations in systems. The correct specification of end to end timing constraints, however, also requires an understanding of the timing properties of the communications between components in a system. Communication is often assumed to occur instantaneously, however the time taken for an item to be transmitted from one component to another can influence the overall timing of, or affect the precise item of data that is used in, a computation, depending on the type of the communication mechanism that is used between the reader and writer. A theory of communication is therefore required, to be used with the RTN-SL, to analyse the timing properties of systems early in the development process. This has motivated two requirements: first the use of a model based approach where functions can be expressed implicitly for compatibility with the RTN-SL, and; second, a method that facilitates the verification of properties of systems, where the communication mechanisms are used as components, would be advantageous.

The remainder of this chapter is structured as follows. Section 1.1 discusses the difficulty in specifying and designing complex real-time systems, how hierarchical development methods can help to manage the complexity of specifications and designs, the role of communications in complex systems and how the use of formal models can help to identify errors and ambiguities in specifications. Section 1.2 then discusses the contribution of the work described in this thesis in more detail.

1.1 Verifying the Correctness of Real-time Systems

[RLKL95] defines a real-time system as:

> "A real-time system is a system that is required to react to stimuli from the environment (including the passage of physical time) within time intervals dictated by the environment."

The critical aspects of this definition are that a real-time system should be: *reactive*, that is react to its environment; and *timely*, that is react and respond to stimuli within defined time limits. This may not simply mean that the system needs to acknowledge receipt of the stimulus, but it may

be required to carry out an action, for example to complete a computation, within a specified time of receiving the stimulus.

These critical requirements make the specification and design of real-time systems complex, because the environmental stimuli can occur at any time and the system must therefore be ready to react to them at any time. There are two categories of deadline that a real-time system may be required to meet: soft deadlines and hard deadlines[Kop90]. In the case of soft deadlines, while it may be that a system is required to perform an action within the deadline, it may be acceptable for this deadline to be missed: the system may continue to operate with reduced functionality for a short period of time, for example. In the case of hard deadlines, however, it may be more critical if a deadline is missed. For example a critical system such as a flight control system on an aircraft, where missing a deadline could cause catastrophic failure (e.g. loss of life). Even then it may be acceptable to miss a hard deadline occasionally, provided it is possible to extrapolate from previous data to enable the system to continue to operate in a stable state. It is unlikely that all of the components in any system will have hard deadlines, but it is necessary to ensure that components that do not have hard deadlines are unable to interfere with components that do in such a way that those hard deadlines cannot be met. Systems which contain components with hard deadlines are referred to as hard real-time systems. The specification, design and implementation of hard real-time systems is more difficult than for soft real-time systems, because of the need to meet these critical deadlines.

The techniques described in this thesis can be used for the development of all types of complex system, however the development of hard real time systems is of particular interest. Their development is especially complex, because such systems have all of the properties of soft real-time systems and additional ones, such as the above requirement to meet safety critical deadlines.

1.1.1 The Role of Hierarchical Development Methods

A classical method of dealing with complexity in systems is to partition the system into a number of simpler components. These components can then, themselves, be split into sub-components in a hierarchical manner until the individual sub-components are simple enough to be understood and implemented. A hierarchical development process will assist in recording the relationship between the components and sub-components in the system.

The use of a hierarchical method introduces an obligation to show that the specifications of the components combine to meet the specification of the complete system. Care must be taken with the specification of the components, because they may interfere with each other, or interact in unexpected

ways[Per99]. For example, in the case of fully asynchronous systems, if two components communicate with each other using a shared area of memory, one component may overwrite the area of memory while another component is attempting to read an item of data from it. In the case of synchronous systems it is possible that the failure of one of the components may lead to deadlock. In reactive systems, where the system is required to react to stimuli from its environment, it is possible that the environment can interfere with the operation of the system at any time. For example the user of the system may cancel a partially completed operation.

MASCOT [JIM87, Sim86], which was the UK MOD preferred method [MoD91, MoD85] for the development of software systems and is still used in parts of the defence industry, is such a hierarchical development process. When using MASCOT a system is structured in terms of a number of interacting components (sub-systems, servers etc.), which, at the lowest level are decomposed into a number of activities in a Real-time Network (RTN), [Sim90c, Sim90b]. These activities are used to specify single sequential computations: parallelism can then arise because multiple activities may execute concurrently, depending on available resources. The sub-systems and activities in a RTN only communicate with each other via explicitly defined routes using a range of different types of communication mechanisms.

1.1.2 The Role of Communication Mechanisms

There is a need for the individual components in a system to communicate with each other, and the type of communication mechanism used can influence the timing properties of a system and also the outcome of a computation. A range of mechanisms is required to facilitate communication between the components of a system: from those that enable synchronous communication to those that allow fully asynchronous communication.

Synchronous communication, as the name implies, requires the components to synchronise in order to communicate with each other. This may be achieved by using a global clock to enable the processes to synchronise and communicate at particular times, or by forcing one process to wait until the other is also ready to communicate. Synchronous communication may be used, for example, where it is necessary for a component to respond to all of the outputs from another component. Its use may, however, lead to a reader of a communication mechanism being held up, while it waits for another component to write the result of a computation to the communication mechanism. The close coupling of components required by synchronous communication may also lead to deadlock, if one of the communicating components fails.

At the other extreme are fully asynchronous communication mechanisms

(or pure ACMs) which do not require any synchronisation between communicating components. This type of mechanism must have some means of ensuring that the reader does not attempt to read an item of data at the same time as it is being written. Pure ACMs are of particular interest because:

1. They allow components that do not share a clock to communicate with each other. This is true even when there is apparent support for synchronous communication, as such synchronous mechanisms need to be built from ACMs, although this may be at the hardware level and hidden from the software (or user).

2. They support the integration of components that run at different speeds, or which are sporadic.

3. They provide a means for decoupling the temporal interactions of components that use them: this may make it easier to analyse the timing behaviour of individual components, because one component cannot interfere with timing behaviour of another component. For example an end to end deadline for a computation can be partitioned among the components that contribute to the computation. It may then be possible to verify that the computation will be completed within an end to end deadline provided the individual components meet their deadlines.

4. They make systems more robust to deadlock of one of their components. For example if the writer is held up the reader can re-read the previous item of data.

Pure ACMs are essentially shared variables that allow communication between processes without placing any constraints on the behaviour of their reader(s) and writer(s). The reader of an ACM may end one read and start the next one while a write is in progress and so multiple reads can overlap a write. Similarly multiple writes can overlap a read. It is possible for an item to be read by the reader a number of times and it is also possible that items will be overwritten before the reader attempts to read them. The asynchronous communication that ACMs support is therefore to be distinguished from the model of "asynchronous communication" supported by (conceptually infinite) buffers, where all items written are read by the reader (normally in the order that they written), for example [JHJ89].

In between the two extremes there are implementations of communication mechanisms that allow different levels of asynchrony between the communicating processes. For example, if it is known that the reader and writer of a mechanism execute on average at the same rate, it may be acceptable

to implement the communication mechanism using a first in first out buffer. This may ensure that the reader is not held up, because there will always be data available to be read, and the writer may never be held up, because there is always space in the buffer to write a new item to. This type of mechanism may be used where it is important that the reader uses every item of data that is communicated. In such circumstances it may even be acceptable for the reader or writer to be held up for a short time, waiting for data or space to become available.

In large distributed systems it is possible for communication between two remote processes to be facilitated via a route which is composed of a number of components. In this case hierarchical methods may be used to develop the specifications of the communication mechanisms themselves.

Communication in MASCOT

MASCOT uses a range of communication protocols, which describe the manner in which the reader(s) and writer(s) communicate with each other on a particular communication route between the components. These protocols facilitate a range of different types of communication between a reader and writer, including fully asynchronous mechanisms and buffer types, where the reader may be held up waiting for data to become available and the writer may be held up waiting for space. In general MASCOT communication mechanisms support multiple readers and/or writers, however, in order to define a theory of communication, it is first necessary to gain an understanding of the behaviour of basic communication mechanisms which have single readers and writers. A range of single reader, single writer protocols is introduced in the next section.

A Range of Communication Protocols

There is a need to provide a means for developers to reason about the behaviour of different communication mechanisms, and this section describes a way of classifying this different behaviour. A set of *basic* communication protocols([PAH00, Sim94, Sim96, Sim03]), that can be used in the design of systems is introduced. These protocols, which are illustrated in Table 1.1, describe a range of levels of synchronisation that may be required between the reader and writer of a communication mechanism as follows:

Channel: similar to a single space buffer. The writer is held up if there is no space available for the item to be written, and the reader can only read each item of data once. The data is conceptually *destroyed* by a read and the reader is held up when the channel is empty.

	Non destructive read (Never held up)	Destructive read (Held up when no data)
Destructive write (Never held up)	 Pool	 Signal
Non destructive write (Held up when no space)	 Constant	 Channel

Table 1.1: The Blocking or Non-Blocking Behaviour of the Basic Protocols

Signal: similar to a single space overwriting buffer. The writer can overwrite older data and is never held up waiting for space to become available. The reader, however, removes data from the protocol and is held up when it is empty.

Constant: as its name implies the data, once written, cannot be overwritten. The reader can always re-read the item that the protocol was initialised with, and the protocol is typically used to store configuration data.

Pool: similar to a shared variable. The reader and writer are never held up. The reader can re-read items of data many times, and the writer can overwrite older items of data.

They are called basic protocols, because, conceptually, they have a single place to store data that is available for communication between the reader and writer, although they may be implemented using a multiple place area of shared memory. Each place is called slot, or buffer, and the provision of multiple slots facilitates concurrent accesses by the reader and writer by directing them to different slots. For example the channel may be implemented with three slots, one to hold the latest item of data, another to hold the item of data that is being read, and a third slot where the writer can write an additional item, before it is held up. This implementation allows a greater level of asynchrony between the communicating processes than would

otherwise be possible. The use of multiple slots ensures that the reader can obtain complete items that have been previously written as the result of a read, even if the writer is concurrently accessing the communication mechanism i.e. the protocol ensures mutual exclusion of the reader and writer on the slots rather than on the communication mechanism itself as would be the case with a Hoare monitor [Hoa74].

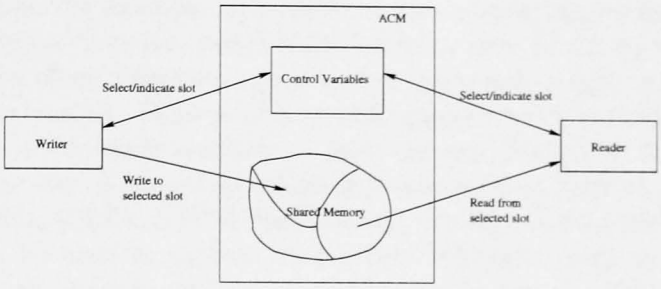

Figure 1.1: A Generic ACM

Of the mechanisms described above the only pure ACM is the pool. The remaining mechanisms require a certain amount of synchronisation between the reader and writer. Pure (true) ACMs typically use a number of areas of shared memory, with separate control variables to direct the reader and writer to different slots if they both access the mechanism at the same time. These control variables are implemented in shared memory themselves, and in fully asynchronous implementations the writer of a control variable can interfere with the reader by writing to the variable while it is being read. This may result in the reader obtaining an incorrect value as the result of reading a control variable. The implementation must still ensure that the reader: accesses a different slot to the writer when this occurs, so that it returns valid data as the result of a read and; second, that it reads a recently written item of data. A generic ACM is illustrated in Figure 1.1. The implementation of a pool that is described in this thesis, Simpson's 4-slot [Sim90a], uses four slots and four control variables to allow the reader and writer to access the mechanism in a fully asynchronous manner.

The techniques described in this thesis have been used to analyse basic ACMS, such as those described in this section, although it may be possible to extend them to enable analysis of the more complex ones: for example multiple reader and writer mechanisms, and mechanisms which are implemented on routes which are comprised of a number of components. This is to be the subject of further work and this point will be revisited in Chapter 8.

1.1.3 The Role of Formal Methods

Section 1.1.1 describes how the use of hierarchical methods can assist in the development of complex systems and Section 1.1.2 describes the role of communication in those systems. This section discusses the role of formal methods in the development of such complex systems.

Errors can arise in the specifications of asynchronous systems for a number of reasons, for example as a result of unexpected interactions between the components (it is also possible for errors to arise in the design and implementation of asynchronous systems with apparently simple specifications for the same reason). There is also an obligation to verify that the specifications of the components combine to meet the specification of the complete system. The use of formal specifications can assist in both of these areas [Hal90, BH95a, BH95b, LFB96, Bic98, HB99, Jon90]. Formal models of the system can be used to explore its possible behaviours and this can help to expose, and correct, errors and ambiguities in the specifications of the system and its components. In addition by starting with an abstract specification and progressively building, and exploring properties of, more realistic models it is possible to gain a better understanding of the behaviour of the system and the interactions of its components. Established techniques can then be used to verify that the composition of a set of components meets the specification of the complete system. For small systems it may be possible to verify the correctness of the complete system by showing that the required properties hold of the complete system in an *exhaustive manner* i.e. that those properties hold for all possible states of the system. For larger systems, however, the state space of the complete system may be too big, or too complex, for this type of method to be practical. Even if it is possible to construct a finite state model of the system, it is extremely difficult to ensure that the the model is constructed correctly. In addition it may not be tractable to discharge the required proof obligations for all possible states of the system. Model checking methods may be used to verify properties of some systems, but this may require abstraction to be used, or a model checking technique to reduce the state space. There is a danger that any abstraction hides a crucial property that may invalidate the results obtained. In any case, even with modern fast machines, model checking large systems with very large state spaces may not be tractable. It may be possible to use a *compositional* proof method to overcome these disadvantages: if it is possible to establish invariants (or assertions) that hold in the different states of the individual components that are sufficient in themselves to ensure the required properties of the system hold. There is then a requirement to prove that the individual components do not *interfere* with each other. In other words it is necessary to show that, if an assertion holds for a component, any

actions executed by the other components do not invalidate that assertion.

A formal specification language, called RTN-SL, is being developed to allow the rigorous specification of functionality and timing properties of activities in RTNs, so that it is possible to analyse and verify properties of the specification of those activities in a rigorous manner. A state machine is used to specify the ordering and timing of operations within the computation with a VDM-SL like language to specify the functionality of those operations. However, there is currently no means of analysing the timing behaviour of the communication between activities and there is a need to develop a theory of communication, which is compatible with the RTN-SL, for this purpose. This theory may then be used with the RTN-SL to verify properties of complete systems in a compositional manner. Discharging the proof obligations to verify properties of the complete system may be difficult and some form of machine assistance will be invaluable in making the proofs more tractable.

Machine Assisted Proofs

There is a need to discharge proof obligations in order to verify properties of formal models: the proofs are often long and tedious, and the probability of errors completing such proofs by hand is high. While it is acknowledged that proofs may also be long and involved even with machine assistance, the use of a proof tool will help to make the proofs less error prone. Using such a tool to assist with completion of the proofs is therefore felt to be essential. PVS[OSRSC99a, OSRSC99b] has been used to assist with the completion of the proofs described in this thesis.

PVS is an interactive environment for writing formal specifications that facilitates machine assistance for discharging formal proofs. It provides an expressive specification language, which augments classical higher order logic, with a sophisticated type system containing, for example, predicate sub-types, combined with a mechanism for defining abstract data types, such as lists and trees.

PVS has a powerful interactive theorem prover with built in proof *tactics* that can make the individual proof steps much larger than is possible with comparable systems. It has been used to verify properties of complex fault-tolerant algorithms [LR93a, LR93b, LR94]. The use of PVS to verify properties of the models has helped in making some of the complex proof obligations, that need to be discharged to show that the 4-slot is Lamport-atomic, more tractable, and also helped in ensuring correctness of the models[1].

[1] The prover automatically checked the models for type correctness, for example.

1.2 Contribution

This thesis shows how it is possible to use a range of tools to explore the properties of asynchronous real-time systems, to gain a better understanding of those systems and increase confidence that they meet their requirements. A number of methods are available to help cope with the complexity. These include the use of a hierarchical development method to partition the system into a number of simpler components. There is then a proof obligation to show that the system meets its specification when it is composed of those components. In addition formal models can be used to explore properties of the system, to gain a better understanding of its behaviour.

The contribution of this work is that it identifies a means for analysing the behaviour of asynchronous real-time systems, which can form the basis of a method to develop a theory of communication, and assist to reduce the amount of rework that is required as a result of flaws in the earlier stages of development. Specifically it:

1. demonstrates how a tool set can be used to gain an understanding of the behaviour of the system, to help to identify and correct ambiguities that arise in the earlier stages of development;

2. shows how an incremental development approach can be used: first to verify properties of increasingly realistic models of the system, building confidence about the correctness of a model of the implementation at each stage; and second to gain an increased understanding of the behaviour of the system as properties of those increasingly sophisticated models are explored. The better understanding gained in the earlier stages can to help to identify properties that need to be observed by the models in the later stages;

3. identifies a means of recording assumptions in the specification, to help ensure they are not overlooked, thereby introducing errors in the design and implementation; and

4. shows that it is possible to use a compositional rely-guarantee method to verify properties of systems where the individual components give few or no guarantees about their individual behaviour. It may then be possible to use the rely-guarantee conditions that have been verified to hold, to explore and verify properties of larger systems, where the system is itself used as a component.

1.3 Thesis Structure

The remainder of this thesis is structured as follows: First Chapter 2 introduces a taxonomy of asynchronous communication mechanisms. [Lam86b] gives a taxonomy of ACMs that give increasing guarantees about their behaviour: the ACM that gives the strongest guarantees is called atomic. The taxonomy described in this thesis builds on that from [Lam86b] and includes additional types of ACM that can be used to build more complex ACMs. The taxonomy uses Real-time Logic (RTL), [JM94, JMS88], to reason about the timing of actions of the reader and writer to the ACMs. Chapter 3 introduces a number of communication mechanisms: first a series of mechanisms are described that require varying degrees of synchronisation between their readers and writers; then Simpson's fully asynchronous atomic ACM implementation is introduced. While the taxonomy in the previous chapter is not used directly to verify the correctness of any implementations, Chapter 3 also shows how Simpson's 4-slot can be constructed from components, some of which are described by the taxonomy.

Chapter 4 describes an abstract model of atomicity and verifies the correctness of the model. This model forms the basis for the proofs in the remainder of the thesis, when Simpson's 4-slot (an implementation of a fully asynchronous ACM) is used as a case study to demonstrate the use of a tool set in developing a system in an incremental manner. Chapter 5 shows how the 4-slot implementation can be shown to be a refinement of the model in Chapter 4 (subject to an assumption about the atomicity of the actions of the component processes). Chapter 6 then describes a compositional rely-guarantee method that can be used to verify properties of implementations where the individual actions of the components are (Hoare) atomic (for example implementations on single processors). This method is used to verify that the 4-slot implementation is (Lamport) atomic when the actions of the reader and writer can interleave in an unconstrained manner. The models described in Chapters 2 to 6 are given using a VDM-like syntax, and have all been encoded in the PVS logic using the encoding of VDM-SL operations from [ABM98]. Chapter 7 describes how it is possible to verify properties of fully asynchronous implementations of the 4-slot, using CSP, [Hoa85, Ros98], and the FDR tool [Ros98]. The conclusions from the work are given in Chapter 8. Complete details of the formal models that have been used to verify properties of the implementation are contained in Appendices D to I, in the PVS logic (a brief explanation of the translation from VDM-SL to the PVS logic is given in Appendix A), and the complete CSP model from Chapter 7 is given in a further appendix.

Chapter 2

A Taxonomy of Asynchronous Communication Mechanisms

A range of asynchronous communication mechanisms that is available to developers was briefly introduced in Section 1.1.2, and a range of single reader, single writer ACMs will be further described in Chapter 3. This chapter presents Lamport's taxonomy of registers[1] [Lam86b] and extends it to encompass the ACMs that are of interest in the design methods under consideration in this thesis. Lamport's taxonomy describes three different types of register, called safe, regular and atomic, which give increasingly strong guarantees about their behaviour when readers and writers access them, in terms of the items that are communicated between those readers and writers. Safe registers give the weakest guarantees about their behaviour and atomic registers give the strongest guarantees: an atomic register guarantees that the behaviour of the register will be equivalent to some Hoare-atomic interleaving of the read and write accesses. A second paper published at the same time, [Lam86a], introduces a formal definition of the meaning of implementing a system with (instances of) a lower level one, and for reasoning about concurrent systems. [Lam86b] gives examples of registers with stronger guarantees being implemented with registers that give weaker guarantees, including an atomic register that uses regular ones, and the formalism is used to prove the correctness of these implementations. The taxonomy is not used directly to explore properties of communication mechanisms in this thesis, however it includes formal definitions of the desirable properties of ACMs, as will be described later in this chapter. In addition Section 3.3 describes how instances of one the types of ACM from the extended taxonomy can be used as components to construct Simpson's 4-slot ACM and

[1] These registers are used for asynchronous communication between processes or components in a system and they will be referred to as ACMs in the sequel, except where direct reference is made to Lamport's work.

a model of (Lamport) atomicity is used to verify properties of the 4-slot in Chapters 5 and 6.

The rest of this Chapter is organised as follows. The formal definitions of the ACMs in the taxonomy use RTL to reason about the ordering of actions of the readers and writer of ACMs, and RTL is introduced in Section 2.1. Lamport's taxonomy of asynchronous registers is described in Section 2.2, and Section 2.3 critiques Lamport's taxonomy. An extended taxonomy, which includes additional types of ACM that are used in practice, both for communication between processes in distributed systems and for constructing other ACMs, is given in Section 2.4 and a formal model of the extended taxonomy is given. Section 2.5 describes some desirable properties of ACMs and relates these properties to the taxonomy, and finally Section 2.6 discusses why the taxonomy has not been used directly to verify properties of ACM implementations.

2.1 Real-time Logic

The taxonomy in this chapter requires a means of reasoning about the timing properties of the ACMs it defines. One of the first methods proposed for specifying timing properties of real-time systems was Real-Time Logic (RTL), [JM86, JMS88]. RTL is based around the concept of timed events, which can be the start and end events of a particular action, for example. In RTL events occur at specific times, have no duration and can recur many times during the operation of a system. Each occurrence of a particular RTL event must occur at a different time, and later occurrences must occur at a later time to earlier occurrences[2]. RTL can, therefore, be used to reason about the ordering of events during the execution of a computation, for example. RTL has been used in the definition of the semantics of several graphical notations for specifying and designing real-time systems, for example Modecharts, [JLM88, JM94, MSJ96]; in defining the semantics of a hierarchy of communication protocols [Sim03] and for for defining the semantics of Real-Time Kernels, [FW96, FW97]. [Pay01] proposes an extension to RTL to allow the use of *continuous* time, rather than discrete time. Time can then be specified using $\mathbb{R}_{\geq 0}$, the positive real numbers including zero. That extension is not considered in this thesis.

RTL associates events with the number of occurrences of those events up to a particular time. The original RTL syntax used an uninterpreted function, @, which returned the time of a particular occurrence of an event:

$$@ : Event * Occ \rightarrow Time \tag{2.1}$$

[2]RTL does not support the super dense *micro model* described in [MP93].

where *Event*, *Occ* and *Time* represent types of events, occurrence numbers and times, respectively. A type of event can be, for example, the start or end of a computation, *Occ* is often represented using $\mathbb{N}$, the set of all natural numbers, where 0 is used as the occurrence number for the first, or *initial* occurrence of an event. Time is taken to be discrete, or more precisely as $\mathbb{N}^-$, the set of positive natural numbers. There is no concept in RTL of different types of events that occur at the same *instant* of time being ordered, nor of any causal relationship between events[3]. This original syntax has the drawback that the function, @, is partial: for example, the time returned is undefined if the event does not have an *ith* occurrence.

@ has been used in later papers (for example [JM94]), however [JMS88] advocated the used of an occurrence relation, Θ to replace @. The relation, Θ, has the following signature:

$$\Theta : Event * Occ * Time \rightarrow \mathbb{B} \tag{2.2}$$

and asserts that a particular occurrence of an event occurs at a particular time. This relation is used in the definitions of the different types of ACM in the taxonomy given in this chapter, because it has the advantage of being total, which considerably simplifies the logic, and allows classical theorem provers, such as PVS [OSRSC99b], to be used to reason in RTL.

It should be noted that it is not essential to include occurrence numbers in the definition of the occurrence relation, since occurrence numbers can be derived from the event types and times. Indeed the embedding of RTL in the PVS logic given in Appendix B uses such a relation, called ψ. The inclusion of occurrence numbers in the θ relation does, however, help to simplify the some of the definitions of the extended taxonomy given in Section 2.4, and also simplifies the proofs of some of the theorems that verify properties of RTL and the taxonomy, for example where the proof of a theorem is discharged by induction.

2.2 Lamport's Taxonomy of Asynchronous Registers

This section describes Lamport's taxonomy of asynchronous registers, but first the distinction between the *base type* that an ACM can communicate and the *valid type* that it is specified to communicate is introduced. This distinction is important in the definitions and discussion that follow.

[3]It is not possible to specify that one event caused another, simply that they are ordered in time.

2.2.1 Base Type and Valid Type

[Lam86b] distinguishes between the different values that a register is capable of communicating, its *base type*, and the values that it is specified to communicate, its *valid type*. A register implementation consists an area of memory that is used to communicate data. The register is capable of containing any value that can be represented by the different combinations of values of the individual bits from which that area of memory is composed, its *base type*. For example, a register that uses an 8 bit area of memory for communication between its reader and writer can communicate 256 different values. If the ACM is designed to communicate natural numbers these 256 values may correspond to the numbers 0 to 255. It may be, however, that the specification of the ACM states that it should communicate the numbers 0 - 199 between its reader(s) and writer(s). This smaller set of values constitutes the *valid type* for that particular ACM implementation. In some implementations the base type and the valid type are the same, as would be the case in the above implementation if the ACM was specified to communicate the values 0 to 255.

2.2.2 Lamport's Taxonomy

[Lam86b] describes three types of asynchronous register, for which following descriptions are given:

> ...The weakest possibility is a *safe* register, in which it is assumed only that a read not concurrent with any write actions obtains the correct value - that is the most recently written one. No assumption is made about the value obtained by a read that overlaps a write, except that it must obtain one of the possible values of the register...
> ...The next stronger possibility is a *regular* register, which is safe, (a read not concurrent with a write gets the correct value) and in which a read that overlaps with a write obtains either the old or new value. ... More generally a read that overlaps any series of writes obtains either the value before the first of the writes or one of the values being written...
> ...The final possibility is an *atomic* register, which is safe and in which reads and writes behave as if they occur in a definite order. In other words, for any execution of the system, there is some way of totally ordering the reads and writes so that the values returned by the reads are the same as if the operations had been performed in that order, with no overlapping...

These concepts are now described in more detail in order to clarify the definitions. A distinction is made in this thesis between items and values communicated: this distinction is important when it is necessary to reason about the ordering of reads and writes to the mechanisms. It is possible for the writer of an ACM to coincidently write the same value on a number of (possibly consecutive) occasions. In order to distinguish between several attempts to communicate the same value, in the descriptions and models, each value written to the ACMs is encapsulated in an item, and each item is given a unique serial number[4], which increases by one for each item written. Implementations of the ACMs may only be required to communicate values, if no distinction is necessary between different instances of the same value being communicated.

Safe Registers

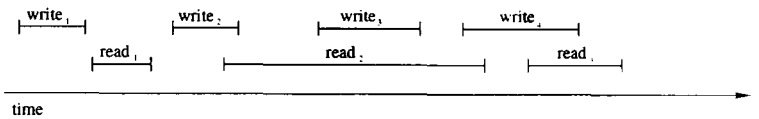

Figure 2.1: Reading From and Writing To an ACM

The behaviour of a safe register is described using the example behaviour illustrated in Figure 2.1, which shows a possible set of interactions of a writer and reader to the register. There are four write accesses and three read accesses, where the durations of the reads and writes are indicated by the length of the line segments. In this example, because $read_1$ does not overlap any of the writes to the register, it will read the latest item that is available; the item written by $write_1$. In the cases of $read_2$ and $read_3$, however, which do overlap with writes to the register, no guarantee is given about the items returned, other than that their values will be of its valid type. These reads can return any of the possible values that are specified to be communicated by the register, including ones that have never been written. This is because the reader and writer are both accessing the same *area of memory* at the same time, and there is no guarantee about how their actions will interleave.

[4]Chapter 3 introduces a model which uses a sequence to represent the items of data that are available to a reader. The model conforms to the VDM convention that the index numbers of sequences start at 1, and the indices of the data items the model, and all of the models in subsequent chapters, also start at 1. The models in the Appendices, that were used to verify properties of ACMs, are given using the PVS logic and the index numbers there commence at zero, since the indices of PVS finite sequences commence at zero.

In the case of $read_2$ the reader may return part(s) of the items written by $write_1$, $write_2$, $write_3$ and $write_4$, and $read_3$ may return part(s) of the items written by $write_3$ and $write_4$, although it is possible that one or both of the reads may return complete items written by the respective writes. It should be noted that this behaviour is even possible where some guarantees are given about atomic accesses to data by the underlying hardware. For example, the hardware may guarantee that accesses to individual *words* are atomic, where the length of the word will be implementation specific. However, if a large data structure is being communicated, for example a database with many different fields, this guarantee is unlikely to be sufficient to ensure valid data is communicated if the reader attempts to read an item of data at the same time as it is being updated by the writer. The simplest implementation of a safe ACM is where the number of values in the data-set that is to be communicated is the same as the number of possible values that the register can represent. For example, if the ACM was implemented using 8 bits and there were 256 different values in the data-set, it would only be possible for the reader to return one of these (valid) values.

Regular Registers

A regular register is safe, in that a read that does not overlap with a write will get the latest item previously written, so in the example shown in Figure 2.1 $read_1$ will again get the item written by $write_1$. A read that overlaps with a write, or more generally a number of writes, will return either the item previously written (by the latest write to finish before the read starts) or the item written by (one of) the overlapping write(s). So in Figure 2.1 $read_3$ could return the item written by $write_3$ or $write_4$, and $read_2$ could return any one of the items written by $write_1$, $write_2$, $write_3$ or $write_4$. Each of the reads must return a valid item, but it is possible for $read_2$ to return the item written by $write_4$ and subsequently for $read_3$ to return the item written by $write_3$. This behaviour seems strange and may be undesirable in an implementation: it is likely that one of the assumptions of an implementation would be that the items would be read in the order they are written.

Atomic Registers

An atomic register has all of the properties of a regular one, but in addition the reads and writes behave as if they had occurred in a particular total order. In other words the implementation is equivalent to a Hoare-atomic sequence of reads and writes; although it is possible for the reader to read the same item a number of times, and for the writer to overwrite items before they have been read. A read that does not overlap with a write will return the item previously written, so $read_1$ in Figure 2.1 will return the

item written by $write_1$. A read that does overlap with a write, or more generally a number of writes, will return either the item written by the latest write that finished before the start of the read, or (one of) the item(s) overlapping write(s), as with a regular register. A read to an atomic register, however, cannot return an item that was written before the item returned by the previous read; i.e. the items must be read in the order that they were written. So in the example in Figure 2.1 an atomic register would behave in the same manner as a regular register, except that if $read_2$ returned the item written by $write_4$, $read_3$ would also return the same item (the item written by the earlier write is not then available to the later read).

2.3 A Critique of Lamport's Taxonomy of Asynchronous Registers

Safe is a slightly strange name for Lamport's first class of register, where it is possible for a reader, that accesses the register at the same time as the writer, to read any valid value of the type being communicated. A more appropriate term is *type-safe* or *type-compatible*, and the term *type-safe* is used in the rest of this thesis to describe such a mechanism. *Atomic* is also a slightly unexpected description of any ACM, because this term usually refers to devices that achieve total ordering of reads and writes (not merely the appearance of it) via synchronisation of the reader and writer, for example Hoare's monitors[Hoa74]. This type of ACM will be referred to as L-atomic in the remainder of this thesis to distinguish this type of atomicity from Hoare-atomicity

[Lam86b] states that any single bit register is type safe. This is because, since the register can only hold two values, the reader must return one of the possible valid values that the register is specified to communicate as the result of a read. It is claimed that this is true even when the value is being overwritten by a new value, since the reader must return either the old or the new value. However a single bit control variable may not even be type safe in some implementations, if read and write accesses to the variable are fully asynchronous. For example, in a hardware implementation it is possible that the reader of the variable will access it when the value is changing, and the result returned by the read may not be a zero or a one. This possibility is fully described and addressed in Chapter 7.

An important point to note is that Lamport's definitions are all couched in terms of complete reads and writes to the registers. However there is a subclass of ACM where there is a critical point during a read when the reader *chooses* to read a particular item, and also a critical point where an item written becomes available to the reader. Simpson's 4-slot is an example of

this type of ACM, as are the L-atomic ACMs in [Tro89,HS94,AG92]. These algorithms use buffers to communicate data between the reader and writer, and control variables; that the writer uses to indicate the buffer that the latest item of data has been written to, and that the reader uses to choose the buffer to read from. These control variables are used to ensure that the reader and writer access different buffers, if a read and write occur at the same time. Effectively the end of the write is when the writer has indicated the buffer that the new item has been written to, which is the time when the new item is available to the reader.

[Lam86b] gives an implementation of a regular register using single bit registers, and of a L-atomic register implemented using regular registers, and states that the weakest (presumably useful) possibility for an ACM is a type-safe one, because situations where anything weaker is used in implementations cannot be envisaged. Many later implementations of L-atomic ACMs (including Simpson's 4-slot) use a type of ACM, to implement buffers which are used to communicate the data items between the reader and writer, that has weaker guarantees than type-safeness. A definition of this weaker type, called persistent, is given in the next section, which describes some useful extensions to Lamport's taxonomy.

2.4 An Extended Taxonomy of ACMs

This section describes an extended taxonomy that includes some useful types of ACM, in addition to those in Lamport's taxonomy, and describes a formal model of the extended taxonomy (The model given here is presented using an adaptation of VDM-SL, which uses the RTL Θ relation to reason about the relative timing of actions. The model uses a shallow embedding of RTL in the PVS logic (due to Paynter), which is described in Appendix B, and the full formal model of the taxonomy, also in the PVS logic, is given in Appendix C). The taxonomy starts with a definition of a *noisy* ACM and builds successive ACMs that give progressively stronger guarantees about their behaviour, with the final definition being that of L-atomicity.

First the model defines a basic ACM, which has a base type of values it can communicate (all of the possible values its registers can represent); a valid type, which consists of all of the user-defined values that are to be communicated by it and is a (possibly complete) subset of the base type (the definition of a valid type is not given here, because it is implementation dependent, although all members of the valid type must also be members of the base type); and a mapping from time to the particular value of the base type that the ACM contains at that time. The valid type is not used in the definition of the basic ACM, but is used in the remaining definitions

to reason about values that can be written or read.

$$Value = \mathsf{token};$$

$$Time = \mathbb{N};$$

$$\begin{aligned} ACM :: & \; baseType : Value\text{-}\mathsf{set} \\ & \; validType : Value\text{-}\mathsf{set} \\ & \; content : Time \xrightarrow{m} Value \end{aligned}$$

$$\mathsf{inv}\ \mathsf{mk}\text{-}ACM\,(bT, vT, c) \triangleq (vT \subseteq bT) \wedge (\mathsf{rng}\ c \subseteq bT);$$

The above definition is given in a VDM-SL like style, and the final line, starting with the VDM-SL keyword inv, is the invariant of the type. The VDM-SL keyword mk indicates the start of the type constructor, and mk-$ACM(bT, vT, c)$ constructs an ACM with a *baseType* called bT, a *validType* called vT and a *content* called c. These names can then be used in the definition of the invariant that follows to refer to the relevant fields of the type.

The formal definitions below make use of the following auxiliary boolean functions (all of the auxiliary functions in the taxonomy are boolean functions, because they are used in the definitions of invariants, axioms and theorems):

val: This auxiliary function takes a reader or writer, an occurrence number and an ACM as parameters, and relates the values written to the particular occurrence of the read or write. The signature of the function is either $val(r, i, v)$ or $val(w, i, v)$ to reason about values read and written respectively. This function does not need to refer to the time of the occurrence, since the time can be derived from the occurrence number.

access: This auxiliary function relates a reader, or writer, to an access of an ACM, and the signature of the function is $access(r, acm)$ or $access(w, acm)$ respectively. This function does not refer to the time or occurrence number of the event, since it is only used in the definitions of a basic ACM and *communicates* (which relates the accesses to occurrence numbers and times as described below), to relate the read and write events to a particular ACM.

communicates: This auxiliary function takes a reader (when it has the signature $communicates(r, i, t_1, t_2, v, acm)$) or writer (when its signature is $communicates(w, i, t_1, t_2, v, acm)$), an occurrence number, two

times, a value and an ACM as parameters, and defines what it means for the reader, or writer, to communicate with an ACM, in terms of the start and stop times, occurrence number and the value written or read.

The basic type of ACM, from which the others are constructed, in the extended taxonomy, could be called *noisy*. It has at least one reader and a single writer, and gives no guarantees about the value a reader will return as the result of a read, other than it will be a member of the base type of the ACM. The writer writes valid values to it, but they may be corrupted as they are written or while they are contained in the ACM, so it is possible that the reader will never read values that have been written. A valid value is, however, communicated to the ACM by the writer when the system is initialised. This type of ACM would not ideally be used in any implementation, but there might be occasions when noisy would be the best description of the particular communication mechanism that is being used, and it is necessary to reason about the behaviour of a particular protocol built on such mechanisms. For example an implementation that uses the TCP/IP protocol for communication over a network would need to allow for the possibility of part(s) of the message being lost, or corrupted, in transmission. In order to ensure correct communication of complete data items the writer would be required to check that all parts of the items had been received and retransmit missing part(s) as necessary. The formal definition of a basic ACM is:

$$
\begin{array}{l}
Basic\_ACM :: writer : Writer \\
\qquad\qquad\quad readers : Reader\text{-set} \\
\qquad\qquad\quad acm : ACM
\end{array}
$$

$$
\begin{array}{l}
\mathsf{inv}\ \mathsf{mk}\text{-}Basic\_ACM\ (w, r, acm) \triangleq \\
\quad \mathsf{card}\ r > 0 \wedge write\_val\_prop1(w, acm) \wedge init\_prop\_1(w, acm);
\end{array}
$$

$$
write\_val\_prop\_1 : Writer \times ACM \to \mathbb{B}
$$

$$
\begin{array}{l}
write\_val\_prop\_1\ (w, acm) \triangleq \\
\quad \forall\, i : occ, v : Value \cdot val(w, i, v) \wedge \\
\qquad\qquad\qquad access(w, acm) \ \Rightarrow\ v \in acm.validType;
\end{array}
$$

$$
init\_prop\_1 : Writer \times ACM \to \mathbb{B}
$$

$$
\begin{array}{l}
init\_prop\_1\ (w, acm) \triangleq \\
\quad \exists\, t, start : Time; v : Value \cdot t < start \wedge \\
\qquad\qquad\qquad\qquad communicates(w, 0, t, start, v, acm);
\end{array}
$$

The first useful type of ACM in the taxonomy also has properties weaker than those of a type-safe ACM. Many implementations of L-atomic ACMs, such as those given in [Sim90a, Sim97, HS94], use this type of ACM, which will be referred to as *persistent*, for communicating the data items between the writer and reader. It is called persistent because, at the end of a write the ACM contains the item written to it, and the item will remain constant (persist) until it is overwritten by the next write. A persistent ACM has the following properties:

1. When a read access to the ACM does not coincide with a write the reader will return the item that was last written to the ACM.

2. A read access that coincides with a write to the ACM may not return a complete item that has been written: the value of the item returned can be any value of the base type.

The set of values that can be returned by a read that clashes with a write is determined to an extent by the type of the data that is being communicated, and also by the implementation of the ACM. Two examples of implementations of a persistent ACM are:

- An important implementation is *dual port* memory, which is memory that the reader and writer can access concurrently (effectively it resides on two different data buses). If the data consists of a single word (whatever size that happens to be), the reader can return any value that the ACM can contain. For example if the word size is 8 bits, the reader can return any of the 256 combinations that the word can take. Some, or all, of these values may be valid values of the type that is being communicated, depending on the size of the data type. If the valid type contains 256 different values the ACM will behave in the same way as a type safe register - whatever value is returned it will be one of the valid values of the type. If there are less than 256 values in the type it is possible that invalid values may be returned by a read in these circumstances. Even then the ACM can be made to behave in the same way as a type safe one, by mapping more than one value that the register can take to some, or all, of the valid values of the type being communicated. A read to dual port memory can return part(s) of the old value and part(s) of the new one.

- Another implementation is where a large data structure is being communicated. In this case the writer may be given control of a large part of memory so that it can assemble the new data value to be communicated, which may consist of a number of values of smaller sub-types. The underlying hardware may make it impossible for the reader to

access anything smaller than a word, but clearly if the reader accesses this area of memory while the new value is being assembled, it may return part of the old value and part of the new one.

This type of ACM requires some mechanism to ensure that it is not accessed by the reader and writer at the same time. Implementations of the class of ACMs described in this thesis all use buffers of this persistent type to communicate the items of data between the reader and writer. Separate control variables are used to indicate which of the slots the reader and writer are accessing. The reader and writer each check which data slot the other is accessing, before starting their own access, and then chose a different data slot to read or write respectively. The definition of a persistent ACM makes use of the auxiliary boolean function *conflicting_read*, which returns true if an occurrence of a read occurs concurrently with one or more writes to the ACM. This function takes a reader and an occurrence number as parameters and has the following signature: $conflicting\_read(r, i)$. The definition also needs to distinguish between the start and end of writes to the ACM, and a the start of a write is referred to by the $start(w)$ event in the auxiliary functions. Similarly the end of a write could be referred to as $end(w)$ and the start and end of a read as $start(r)$ and $end(r)$, respectively. The formal definition of a persistent ACM is:

$$Persistent\_ACM :: b\_acm : Basic\_ACM$$

$$\begin{aligned}&\textsf{inv mk-}Persistent\_ACM\ (acm) \triangleq\\ &\quad write\_val\_prop2(acm) \wedge persistent\_acm1(acm) \wedge\\ &\quad persistent\_acm2(acm) \wedge persistent\_acm3(acm);\end{aligned}$$

$$write\_val\_prop2 : Basic\_ACM \to \mathbb{B}$$

$$\begin{aligned}&write\_val\_prop2\,(a) \triangleq\\ &\quad \textsf{let } w = a.writer,\\ &\qquad acm = a.acm \textsf{ in}\\ &\quad \forall\, i : Occ; v : Value; t_1, t_2 : Time \cdot\\ &\qquad communicates(w, i, t_1, t_2, v, acm) \Rightarrow\\ &\qquad acm.content(t_2) = v;\end{aligned}$$

$$
\begin{array}{l}
persistent\_acm1 : Basic\_ACM \rightarrow \mathbb{B} \\
persistent\_acm1\,(a) \triangleq \\
\quad \textsf{let } w = a.writer, \\
\qquad acm = a.acm \textsf{ in} \\
\quad \forall\, i : Occ; t_1, t_2 : Time; v : Value \cdot \\
\qquad communicates(w, i, t_1, t_2, v, acm) \;\Rightarrow \\
\quad (\exists\, t_3 : Time \cdot \Theta(start(w), i+1, t_3) \wedge \\
\qquad\qquad (\forall\, t : Time \cdot t_2 \leq t < t_3 \;\Rightarrow \\
\qquad\qquad\qquad acm.content(t) = v) \vee \\
\quad \neg (\exists\, t_3 : Time \cdot \Theta(start(w), i+1, t_3) \wedge \\
\qquad\qquad (\forall\, t : Time \cdot t_2 \leq t \;\Rightarrow\; acm.content(t) = v);
\end{array}
$$

$$
\begin{array}{l}
persistent\_acm2 : Basic\_ACM \rightarrow \mathbb{B} \\
persistent\_acm2\,(a) \triangleq \\
\quad \textsf{let } acm = a.acm \textsf{ in} \\
\quad \forall\, r \in a.readers, i : Occ; t_1, t_2 : Time; v : Value \cdot \\
\qquad communicates(r, i, t_1, t_2, v, acm) \wedge \\
\qquad \neg conflicting\_read(r, i) \;\Rightarrow\; acm.content(t_2) = v;
\end{array}
$$

$$
\begin{array}{l}
persistent\_acm3 : Basic\_ACM \rightarrow \mathbb{B} \\
persistent\_acm3\,(a) \triangleq \\
\quad \textsf{let } w = a.writer, \\
\qquad acm = a.acm \textsf{ in} \\
\quad \forall\, v : Value, t : Time \cdot acm.content(t) = v \wedge \\
\qquad\qquad \neg\, acm\_being\_written(acm, t) \;\Rightarrow \\
\quad (\exists\, t_1, t_2 : Time; i : Occ \cdot t_2 < t \wedge \\
\qquad\qquad communicates(w, i, t_1, t_2, v, acm) \wedge \\
\quad \neg (\exists\, t_3 : Time \cdot t_2 \leq t_3 < t \wedge \Theta(start(w), i+1, t_3)));
\end{array}
$$

(The auxiliary function *acm_being_written* is not given here, but defines what it means for a writer to be accessing the ACM in terms of the start and end times of the write access).

Lamport's type-safe ACM is the next in the taxonomy, where the reader is guaranteed to return a valid value as a result of a read as described in Section 2.2.2. It was originally thought that type-safeness was a sufficiently strong property so that a L-atomic ACM could be implemented using 4 type-safe single bit control variables. However, [HV01] gives an informal proof that 5 control variables are required in order to implement a L-atomic ACM from type-safe bits. This is an unpublished paper, but some preliminary results are published in [HS94] and [HV96]. Indeed [Rus02] shows that Simpson's 4-slot ACM is not L-atomic, but only regular, if its 4 control

variables are implemented using type-safe bits. The formal definition of a type-safe ACM is:

$$TypeSafe\_ACM :: p\_acm : Persistent\_ACM$$

$$\text{inv mk-}TypeSafe\_ACM\ (acm) \triangleq typesafe\_acm(acm);$$

$$typesafe\_acm : Persistent\_ACM \to \mathbb{B}$$

$$typesafe\_acm\ (a) \triangleq$$
$$\text{let } acm = a.b\_acm.acm \text{ in}$$
$$\forall r \in acm.readers; i : Occ; v : Value; t_1, t_2 : Time \cdot$$
$$communicates(r, i, t_1, t_2, v, acm) \Rightarrow v \in acm.validType;$$

The remaining additional type of ACM in the extended taxonomy is called semi-regular, which guarantees that a reader can only return values that have previously been written to it. This type of ACM is unlikely to be desirable in an implementation, because it is possible for the reader to always return the initial value as the result of any read. It is included in the hierarchy because its guarantees can be related to our requirement that an ACM should transmit valid data, as described later, in Section 2.5.

$$SemiRegular\_ACM :: s\_acm : TypeSafe\_ACM$$

$$\text{inv mk-}SemiRegular\_ACM\ (acm) \triangleq semiregular\_acm(acm);$$

$$semiregular\_acm : TypeSafe\_ACM \to \mathbb{B}$$

$$semiregular\_acm\ (a) \triangleq$$
$$\text{let } acm = a.p\_acm.b\_acm.acm,$$
$$w = a.p\_acm.b\_acm.writer \text{ in}$$
$$\forall r \in acm.readers; i : Occ; v : Value; t_1, t_2 : Time \cdot$$
$$communicates(r, i, t_1, t_2, v, acm) \Rightarrow$$
$$\exists j : Occ : t_3, t_4 : Time \cdot$$
$$t_3 \leq t_2 \wedge communicates(w, j, t_3, t_4, v, acm);$$

The final types of ACM in the taxonomy are the regular and L-atomic types, which were described in Section 2.2.2 above, respectively. The definition of *Regular_ACM* uses an auxiliary function, *conflicting_actions*, which takes a reader, a writer and two occurrence numbers (occurrence numbers of a read and writer respectively), and has the signature $conflicting\_actions(r, w, i, j)$. It returns true if a particular occurrence of a read to the ACM occurs concurrently with a particular occurrence of a write.

$Regular\_ACM :: sr\_acm : SemiRegular\_ACM$

inv mk-$Regular\_ACM\ (acm) \triangleq regular\_acm(acm)$;

$regular\_acm : SemiRegular\_ACM \rightarrow \mathbb{B}$

$regular\_acm\ (a) \triangleq$
  let $acm = a.s\_acm.p\_acm.b\_acm.acm$,
    $w = a.s\_acm.p\_acm.b\_acm.writer$ in
  $\forall r \in acm.readers; i : Occ; v : Value; t_1, t_2 : Time \cdot$
    $communicates(r, i, t_1, t_2, v, acm) \wedge$
    $conflicting\_read(r, i) \Rightarrow$
  $(\exists j : Occ; t_3, t_4 : Time \cdot$
    $communicates(w, j, t_3, t_4, v, acm) \wedge ((t_4 < t_1 \wedge$
    $\neg (\exists t_5, t_6 : Time; v_1 : Value \cdot$
      $communicates(w, j+1, t_5, t_6, v_1, acm) \wedge t_6 \leq t1)) \vee$
    $conflicting\_actions(r, w, i, j)))$;

The definition of a L-atomic ACM uses the following:

1. An auxiliary function *r_communicates* defines the set of values that are available to the reader during a read of a L-atomic ACM - the values from the set of items that were written by any conflicting writes, and the item written by the last write to end before the read started.

2. A new *DataItem* (item) type, which has an unique *Id* and a value, in addition to the name of the ACM to which it is written. The *Id*s start at zero and increase monotonically, so the occurrence number of the write that placed the item in the ACM is used for this purpose.

3. A new version of the *communicates* function that takes a *DataItem* instead of a *value* as a parameter (otherwise it is identical to the previous version), as well as the original version of the function.

The formal definitions of a L-atomic ACM and *r_communicates* are:

$L\text{-}Atomic\_ACM :: r\_acm : Regular\_ACM$

inv mk-$L\text{-}Atomic\_ACM\ (acm) \triangleq L\text{-}atomic\_acm(acm)$;

$$r\_communicates : Reader \times Occ \times Time \times Time \times DataItem \times ACM \times Writer \rightarrow \mathbb{B}$$

$$\begin{aligned}
&r_communicates\,(r, i, t_1, t_2, x, a, w) \triangleq \\
&\quad communicates(r, i, t_1, t_2, x.value, a) \wedge x.acm = a \wedge \\
&\quad x.id \in \{j : Occ \mid \exists\, t_3, t_4 : Time \cdot communicates(w, j, t_3, t_4, x, a) \wedge \\
&\quad (t_4 \leq t_1 \wedge \neg\, (\exists\, t_5, t_6 : Time; y : DataItem \cdot \\
&\qquad communicates(w, j+1, t_5, t_6, y, a) \wedge \\
&\qquad t_6 \leq t_1) \vee conflicting_actions(r, w, i, j))\};
\end{aligned}$$

$$L\text{-}atomic\_acm : Regular\_ACM \rightarrow \mathbb{B}$$

$$\begin{aligned}
&L\text{-}atomic_acm\,(a) \triangleq \\
&\quad \textsf{let } acm = a.sr_acm.s_acm.p_acm.b_acm.acm, \\
&\qquad w = a.sr_acm.s_acm.p_acm.b_acm.writer \textsf{ in} \\
&\quad \forall\, r \in acm.readers; i : Occ, t_3, t_4 : Time, x_2 : DataItem \cdot \\
&\qquad r_communicates(r, i+1, t_3, t_4, x_2, acm, w) \Rightarrow \\
&\quad \exists\, t_1, t_2 : Time, x_1 : DataItem \cdot \\
&\qquad r_communicates(r, i, t_1, t_2, x_1, acm, w) \wedge \\
&\qquad x_2.id \geq x_1.id;
\end{aligned}$$

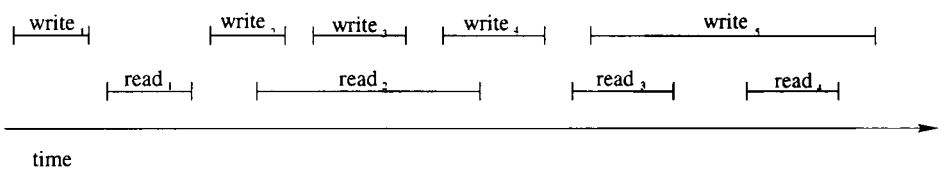

Figure 2.2: Example Read and Write Behaviour of a L-atomic ACM

The above possible behaviours are illustrated in Figure 2.2, which is explained as follows:

- $write_1$ will have overwritten the previous contents of the ACM. $read_1$ does not overlap with any writes, so the only item available to it is the one that was written by $write_1$.
- $read_2$ overlaps with $write_2$, $write_3$, and $write_4$, so it can return the item written by any of these writes, or the one written by the last write to completely finish before it started, $write_1$.
- $read_3$ can return the value written by $write_4$ or $write_5$, which it overlaps.
- similarly $read_4$ may return either of the items written by $write_4$ or $write_5$, unless $read_3$ returned the item written by $write_5$ in which case

this is the only item available to $read_4$[5].

Various properties have been verified to hold for the types of ACM defined in the taxonomy, in particular the following property has been verified of an L-atomic ACM[6]:

$$
\begin{array}{l}
\textit{L-atomic\_test\_th} : \textit{THEOREM} \\
\quad \forall r : \textit{Reader}, w : \textit{Writer}, i, j : \textit{Occ}, \\
\quad t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9, t_{10}, t_{11}, t_{12} : \textit{Time}. \\
\quad x_1, x_2, x_3, x_4 : \textit{DataItem}, v : \textit{Value}, \textit{acm} : \textit{L-Atomic\_ACM} \cdot \\
\qquad t_2 \geq t_9 \wedge t_{10} \geq t_3 \wedge t_{11} > t_4 \wedge \\
\qquad \textit{communicates}(w, i, t_5, t_6, x_1, \textit{acm}) \wedge \\
\qquad \textit{communicates}(w, i+1, t_7, t_8, x_2, \textit{acm}) \wedge \\
\qquad \textit{communicates}(w, i+2, t_9, t_{10}, x_3, \textit{acm}) \wedge \\
\qquad \textit{communicates}(w, i+3, t_{11}, t_{12}, x_4, \textit{acm}) \wedge \\
\qquad x_3.\textit{value} \neq x_2.\textit{value} \wedge x_3.\textit{value} \neq x_1.\textit{value} \wedge \\
\qquad \textit{communicates}(r, j, t_1, t_2, x_3.\textit{value}, \textit{acm}) \wedge \\
\qquad \textit{communicates}(r, j+1, t_3, t_4, v, \textit{acm}) \wedge t_1 \geq t_7 \Rightarrow \\
\qquad x_3.\textit{value} = v;
\end{array}
$$

This theorem verifies that, where two reads overlap with a write, and the second of those two reads has returned the latest item available to it, the later read cannot return an item that was written before this item. This behaviour is illustrated in Figure 2.3, which is described below:

1. It would be possible for the jth read to return any of the items written by writes i, i + 1 or i + 2.

2. If however, it returns the item written by write i + 2 then read j + 1 must return the same item. It cannot return the item written by write i + 3, because that write does not start until after the read ends.

3. It should be noted that this theorem distinguishes between values and items written. It specifically states that the value written by write i + 2 is not equal to the values written by writes i and i + 1. This is because it is possible the for the same value to be written by three consecutive writes, for example, so read j + 1 could return the value written by write i (the value of item x_1) if that value was the same as the one written by write i + 2 (the value of item x_3).

[5]The item written by $write_4$ must have already been overwritten before $read_3$ acquired the item it was going to read, since it returned the item written by $write_5$. Therefore, since $read_4$ occurs after $read_3$ the item written by $write_4$ is not available to it.

[6]The interested reader can download the PVS theory, and proof scripts, from http://homepages.cs.ncl.ac.uk/neil.henderson/fme2002/taxonomy.tgz.

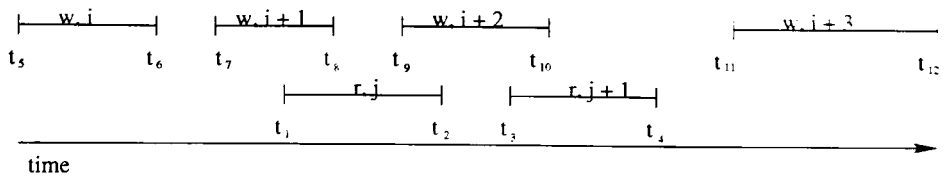

Figure 2.3: Proof of a Property of L-atomicity

2.5 Desirable Properties of ACM implementations

It is desirable that an ACM implementation of a shared variable will communicate coherent and fresh values between its writer(s) and reader(s). Definitions of coherence and two different versions of freshness, freshness with respect to an individual read (*local freshness*) and freshness respect to a sequence of reads (*global freshness*), are given below.

It is important that ACMs ensure coherent data is transmitted, even though the reader and writer are totally unconstrained as to when they access the mechanism. Coherence means that the reader of the ACM will read complete valid items, that have previously been written by the writer, when it accesses the mechanism. Semi-regular ACMs guarantee coherence.

The requirement for local freshness means that a reader will:

- Read the last item written prior to the start of a read, when the read does not overlap in time with a write.
- Read the last item written prior to the start of the read or one of the items written by an overlapping write, if the read does overlap in time with one or more writes.

This is the definition of freshness from [Sim04], and local freshness is guaranteed by regular ACMs (in addition to coherence).

The definition of freshness that is used in this thesis is that a read will return a globally fresh item of data, and this requires that items are locally fresh and additionally that they are read in the order that they have been written. Global freshness is guaranteed by L-atomic ACMs (in addition to coherence).

2.6 Using the taxonomy to Verify Properties of an Implementation

It was originally intended to model Simpson's 4-slot ACM using the extended taxonomy of ACMs described above: the model was to have been written in the PVS logic, using instances of particular types of ACM from the taxonomy to implement the control variables and slots, with a state machine to describe the algorithm. The model would consist of:

- 4 persistent slots;
- 4 single bit ACMs for its control variables. This would require an extension to the existing taxonomy, because the type of ACM required to implement the control variables is not currently included(it was originally assumed that the control variables were type-safe, but this is not the case, as discussed in Section 2.4); and
- an *algorithm* to describe how reads and writes are executed.

A proof theory would need to be devised to verify that the 4-slot implementation is L-atomic using this model, however there were a number of difficulties with this approach:

1. It was not clear how the model of the implementation could be verified to be equivalent to (a refinement of) the definition of L-atomicity from the taxonomy. The definition describes the behaviour of a L-atomic ACM in terms of complete reads and writes interleaving and overlapping. The implementation would, however, be described in terms of individual actions of the reader and writer, and these actions could interleave or overlap with each other.

2. It was not clear how to model the state machine to describe the algorithm in the PVS logic. The item of data that is available to the reader can depend on the order in which the reader and writer actions occur. In addition it is possible for an unbounded number of reader actions to occur between any two writer actions, and vice versa. In order to reason about the equivalence of the model and the implementation of an ACM, a means of encoding the (timed) ordering of the reads and writes in the model and the (untimed) ordering of the actions in the implementation, perhaps in the form of *traces* would be required. In addition a proof theory would also be devised in order to reason about the equivalence of traces of the model and implementation.

3. The model of the taxonomy is complex and difficult to follow: the taxonomy is hierarchical; each of the ACMs in the taxonomy inherits the behaviour of its parent and refines it by providing additional guarantees about its own behaviour. It is not possible to understand the behaviour of a L-atomic ACM from the taxonomy without understanding the behaviour of all of the other types. The model of the implementation would also be complex and difficult to understand: it would include 8 components ACMs, to model the control variables and buffers, plus a model of the algorithm. This complexity would make it difficult to understand and verify properties of the model.

Further work on the taxonomy was therefore deferred in favour of an approach with an abstract model of L-atomicity as its basis. This definition would then be available as an abstract introduction to the requirements for an ACM implementation, coherence and freshness, and could form the basis of any correctness proofs for particular implementations. Models of the implementation could be developed in a progressive manner, removing abstractions in the model, for example relaxing any assumptions about Hoare-atomicity of the actions of the reader and writer, with each iteration. This iterative approach can be continued until sufficient confidence is gained in the correctness of an implementation against its requirements. This would address the above shortcomings as follows:

- A known method could be used to show that the implementation is a refinement of the model: Nipkow's retrieve relation, [Nip86, Nip87]. This would, however, only partly address points 1 and 2, since it would be necessary to assume that groups of actions of the reader and writer are executed atomically, and this point is further discussed in Chapter 5.

- A means of relaxing the assumption about the atomicity of the reader and writer actions, using a compositional rely-guarantee method is introduced in Chapter 6. This further addresses points 1 and 2, by allowing the individual actions of the reader and writer to interleave in an unconstrained manner.

- The use of an iterative approach enables an understanding of the behaviour of the implementation to be built up as increasingly realistic models of the implementation are built in each iteration of the development process. This helps to address point 3.

- Points 1, 2 are finally addressed fully using CSP, with the FDR model checker, as described in Chapter 7, where the actions of the reader and writer are allowed to occur in a fully asynchronous manner.

2.7 Summary

This chapter describes a taxonomy of ACMs, which give increasing guarantees about their behaviour, together with a formal model of the taxonomy. ACMs in the hierarchy that give stronger guarantees can be implemented with instances of ACMs that give weaker guarantees. It then discusses the reasons why this approach was deferred in favour of an iterative approach, which starts with an abstract model of the requirements and verifies the correctness of an implementation against those requirements by verifying properties of increasingly realistic models of the algorithm. This revised approach allows an understanding of the behaviour of the implementation to be gained over time: lessons learned from verifying earlier models can prove valuable in creating and verifying later models. Chapter 3 first introduces a number of implementations of communication mechanisms that allow varying levels of asynchrony between their readers and writers, and then describes Simpson's fully asynchronous 4-slot ACM implementation and gives a formal model of this implementation. Chapter 4 then gives an abstract model of L-atomicity, and describes how the model has been verified to be equivalent to the definition in this chapter. Increasingly detailed models of the 4-slot are then given in the succeeding chapters, to explore properties of the implementation in an iterative manner and gain sufficient confidence in its correctness against its requirements.

Chapter 3

L-atomic ACMs

Chapter 2 described an extended taxonomy of ACMs, which give increasingly strong guarantees about their behaviour when their readers and writers access them. The strongest guarantee, L-atomicity, is a desirable property of any fully asynchronous (pure) ACM i.e. that the reader will always read globally fresh coherent data. Section 1.1.2 introduced an alternative means of classifying the behaviour of communication mechanisms in terms of a number of protocols that dictate the level of synchronisation that is required between their reader(s) and writer(s). One of the protocols, the pool, can be implemented using a pure ACM, for example see [Sim90a, Tro89, AG92, Sim97], in order to ensure that its reader(s) and writer(s) are never held up. However in a particular implementation absolute asynchrony may not be required and a classical method of implementing asynchronous communication is to use an n-place buffer between the reader and writer. The writer adds a new item to an empty place, and is only held up when the buffer is full (no places are available), and the reader removes items from the buffer and is only held up when the buffer is empty. Such buffers are often modelled as if they had an infinite number of places [JHJ89]. The developer of a system may require a means to reason about the behaviour of the different types of mechanism that are available, for example to trade performance against the resources that are used in an implementation. While it may be appropriate to use fully asynchronous communication between components where freshness of data is the overriding requirement, it may be less appropriate in other situations. For example where it is important that the reader processes every item of data it may be more appropriate to use a buffer between the communicating processes. The use of a buffer does, however, require the use of additional hardware resources, because of the potential need to store multiple items. Implementations of fully asynchronous mechanisms also require the use of a number of slots, so that the reader and writer can access different slots if they are reading and writing

concurrently, to ensure that the writer does not overwrite an item of data as the reader is reading it. It may be possible to trade absolute asynchrony, in some situations, in return for using less resources, for example where it is acceptable for the reader to be held up for a short time while the writer completes a write to the ACM. This chapter introduces a range of communication protocols and implementations of communication mechanisms that allow varying degrees of asynchrony between their readers and writers. The last of these is a fully asynchronous L-atomic ACM, Simpson's 4-slot, and it is shown how this can be implemented with instances of ACMs that are not themselves L-atomic. The 4-slot is the main vehicle for the investigations in this thesis.

The remainder of this chapter is organised as follows. A number of communication mechanisms that are implemented with fewer than 4 slots, but which either fail to be L-atomic or fully asynchronous are introduced in Section 3.1. The failure modes of these ACMs will be described and illustrated with examples. Section 3.2 describes an implementation classification scheme for ACMs that is used in academic literature; and introduces some impossibility results that give the minimum requirements for ACMs in terms of this classification scheme. Finally Section 3.3 describes Simpson's fully asynchronous ACM implementation, gives the algorithm for the ACM and introduces a formal model of the algorithm.

3.1 Communication Mechanism Implementations

[Sim90a] gives implementations of communication mechanisms that are implemented using 1, 2, 3, and 4-slots. The 1, 2 and 3-slot implementations, which are described below, can all fail to communicate coherent data if they are implemented in a fully asynchronous manner. An alternative 2-slot implementation is also given which may fail to communicate fresh data to its reader. These mechanisms may, however, allow a degree of asynchrony between their reader and writer and will be referred to as ACMs. For example in an implementation where the reader and writer both execute at approximately the same speed and the read and write actions are relatively short in relation to their overall algorithms, it may be perfectly acceptable to use a single slot shared variable for communication between them. This shared variable may then need a mechanism to ensure that the reader obtains correct data as the result of a read. For example it could be implemented using a mutual exclusion mechanism, such as a monitor, or the reader may check that correct data has been read (for example using a cyclic redundancy check), and re-read if it detects that the data is incorrect: this checking and

re-reading may have little cost for small data structures.

3.1.1 1-slot ACMs

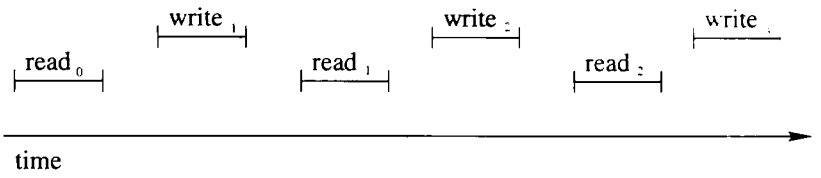

Figure 3.1: Accidental Synchronisation of a Reader and Writer

The 1-slot mechanism has a single area of shared memory that can be used for communicating data between the reader and writer. The reader and writer may accidentally avoid *interfering* with each other (Figure 3.1), or they may avoid interfering because they are implemented on a single processor, and the data structure that is being communicated is small enough to be written and read in a single atomic action. In general, however, non-interference can only be guaranteed if some type of synchronisation mechanism is used, for example a Hoare monitor [Hoa74].

3.1.2 2-slot ACMs

A 2-slot ACM implementation has two areas of shared memory that are used to communicate data between the reader and writer. If the reader and writer both access the ACM at the same time they should be directed to different slots of shared memory to ensure that the reader can read a coherent data item, and the writer can concurrently write a new value to the mechanism. The reader and writer use control variables to indicate the slot they are currently accessing, and they each check the control variable written by the other process at the start of an access. In this way the writer may choose to access the opposite slot to the reader and vice versa.

The 2-slot implementation from [Sim90a] is given in Table 3.1, and is described below (in the description one of the slots is initialised - the one that is initially available to the reader - and the value *nil* is used to indicate that the other slot is not initialised).

It should be noted that the local variable *index* in the write procedure of this algorithm could be omitted. Only the writer to the mechanism has write access to the *latest* control variable, so the *writerChoosesSlot* action could be omitted and the writer could simply access *latest* during the *write*

Table 3.1: A 2-slot ACM Implementation

```
mechanism two slot;
   type
   var data: array[bit] of Data := (init_item,nil);
       latest: bit := 0;

   procedure write (item: data);
      var index: bit;
   begin
      index := not latest;          (writerChoosesSlot)
      data[index] := item;          (write)
      latest := index;              (writerIndicatesSlot)
   end;

   function read: Data;
      var index: bit;
   begin
      index := latest;              (readerChoosesSlot)
      read := data[index];          (read)
   end;
end;
```

action. The local variable and *writerChoosesSlot* action are included here to faithfully reproduce Simpson's implementation.

The reader and writer local variables *index* are used to obtain pointers to the slots that are to be read and written. The global variable *latest* is used by the writer to indicate the slot it has written to. There is a write procedure consisting of three actions, and a read function consisting of two actions. The writer alternates between the two slots - this is the meaning of the *writerChoosesSlot* action, *index*: = not *latest* (the variable is a single bit and the writer *negates* the value each time it is used). Having chosen the new slot, the writer then writes the value and indicates the slot it has written to. The reader first chooses the slot to read from (the last slot that writer indicated it has written to) and reads the item from that slot. This implementation of a 2-slot ACM attempts to transmit the latest item of data to the reader at the possible expense of maintaining coherence.

For example, data coherence may be lost if the reads and writes to the ACM occur as illustrated in Figure 3.2. This diagram is explained as follows:

1. The reads and writes are indicated using 2 vertical lines denoting the start and end times of the actions, connected by a horizontal line

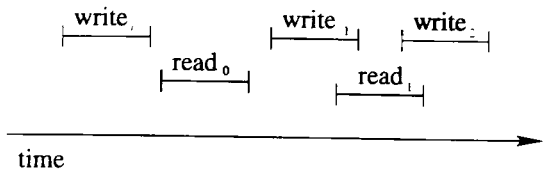

Figure 3.2: Incorrect Operation of a 2-slot ACM - 1

indicating the total time taken for the action. The type of action and instance number is indicated above the horizontal line. The instance numbers increase monotonically by one for each write or read, starting at zero.

2. While the individual actions of the reader and writer are ordered as shown in Table 3.1, the actual value may be read or written at any time during the respective actions. For example, the writer may start a write action by executing *writerChoosesSlot*, but the actual writing of data may take place at any time before the *writerIndicatesSlot* action is executed. Indeed the writer may be descheduled during the write to allow a higher priority process to run and the actual write action itself may be interrupted.

If the read and write operations occur as illustrated, the writer, when it starts $write_2$, will choose to overwrite the item in the slot that was used during $write_0$. The reader, at $read_1$, may have chosen to read either the item written by $write_0$, or that written by $write_1$. In the former case the writer may interfere with the reader, which may get part(s) of the items from $write_0$ and $write_2$.

In an alternative implementation of a 2-slot ACM, designed to maintain coherence at the expense of freshness, the writer could check which slot the reader is accessing before it starts a write access, and then write the new value to the slot that is not currently being accessed by the reader. Provided that the accesses to the control variables are atomic this should always ensure that the writer accesses a different slot from the reader[1]. In this implementation it is possible for the reader to always read an old value. For example consider the situation where the read and write accesses occur in the manner shown in Figure 3.3.

In this case the writer may choose to access the opposite slot to the reader each time a new write is started. The reader will also avoid the writer, and

[1] If the accesses to the control variables are not atomic the reader and writer may choose to access the same slot at the same time, because they clash on reading and writing one of the control variables.

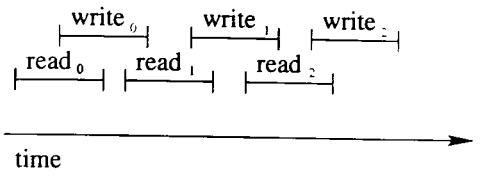

Figure 3.3: Incorrect Operation of a 2-slot ACM - 2

in doing so it may always access the same slot, and read old data. This pattern of read and write accesses could always occur from system start up, in which case the reader will always return the initial value written to the ACM.

3.1.3 3-slot ACMs

A 3-slot ACM, goes one step further in an attempt to keep the reader and writer apart by adding a third slot. The implementation from [Sim90a] is shown in Table 3.2 (similarly to the 2-slot description the slot initially available to the reader is initialised). This algorithm works in the following way:

1. There are two control variables: *latest*, which is used by the writer to indicate the last slot it has accessed, and *reading*, used by the reader to indicate the slot it is currently reading.

2. The reader follows the writer by choosing to read the slot last written (*readerChoosesSlot*), at *readerIndicatesSlot* it indicates the slot it is reading from and reads the item from the chosen slot (*read*).

3. The writer uses the array *differ* to avoid the slot it last wrote to, and also to attempt to avoid acquiring the slot that has been already been chosen by the reader. For example, if the writer last wrote to slot one, and the reader last indicated that it had chosen to read slot two, the writer would choose to access slot three at *writerChoosesSlot* (*index* : = *differ*[1, 2]). The writer then writes the new item to the chosen slot and indicates the slot it has written the item to at *writerIndicatesSlot*.

[Sim90a] states that there are two problems with this implementation. The first is that the reader and writer can access the same control variable at the same time, in which case the integrity of the value read from the control variable cannot be guaranteed. The solution recommended to overcome this flaw is to use 2-slot implementations for the control variables. The second flaw is that it is possible for coherence of the data being communicated to

Table 3.2: An Implementation of a 3-slot ACM

```
mechanism three slot;
 var data: array[1..3] of Data := (init_item, nil, nil);
     latest, reading: 1..3 := 1, 1;

 procedure write (item: data);
  const differ = ((2, 3, 2), (3, 3, 1), (2, 1, 1));
        var index: 1..3;
 begin
  index := differ[latest, reading];   (writerChoosesSlot)
  data[index] := item;                (write)
  latest := index;                    (writerIndicatesSlot)
 end;

 function read: Data;
  var index: 1..3;
 begin
  index := latest;                    (readerChoosesSlot)
  reading := index;                   (readerIndicatesSlot)
  read := data[index];                (read)
 end;
end;
```

be lost, if the read and write actions interleave in a particular manner. For example consider the interleaving of read and write actions and assignments to control variables shown in Table 3.3. This shows that if the writer executes the *writerIndicatesSlot* and *writerChoosesSlot* actions between the reader choosing and indicating the slot it is going to read, it is possible for the reader and writer to access the same slot at the same time. Chapter 6 shows how a formal model can be used to identify the precise ordering of actions and values of the control variables shown in this counter example.

[Sim90a] also gives an additional timing constraint which, if it can be guaranteed, makes the 3-slot behave in the same way as an ACM. The constraint is that

> ... the interval between control operations in the read function is always shorter than the interval between writes ...

In practice it may be difficult to guarantee that this timing constraint will always hold, and on its own it may be insufficient to guarantee that the reader and writer will not access the same slot at the same time. Figure 3.4 shows possible timings of the read and write actions that can lead to

Table 3.3: Assignments to the Control Variables

	latest	reading	write.index	read.index	
initial vals	2	1	3	1	
readerChoosesSlot	2	1	3	2	
writerIndicatesSlot	3	1	3	2	
writerChoosesSlot	3	1	2	2	
readerIndicatesSlot	3	2	2	2	*wr* and *rd* to slot 2

incorrect operation of the 3-slot ACM, even when this timing constraint is met, if the accesses to the control variables are not atomic. In this case the time between writes (the end of *writerIndicatesSlot* - *wis* - and the start of *writerChoosesSlot* - *wcs*) is greater than the time between the read control operations *rcs* (*readerChoosesSlot*) and *ris* (*readerIndicatesSlot*). However, because the *ris* and *wcs* actions overlap it is possible for the writer to read the *reading* control variable incorrectly during *wcs* and to choose to write to the same slot as the reader chooses to read[2].

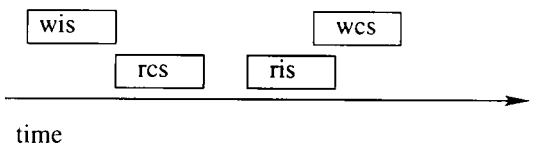

Figure 3.4: Incorrect Operation of the 3-slot ACM

In fact, if the read and write actions overlap as shown in Figure 3.4, the very solution that Simpson suggests for ensuring that the reader and writer return coherent values when accessing the control variables, which is to use 2-slot mechanisms for implementation of the control variables, may ensure that the reader and writer clash on reading and writing the same slot in the mechanism. For example, using the 2-slot implementation from Table 3.1, the writer may return the old value of the control variable *reading* at *wcs*, not the new value that is currently being written by the reader (by the *ris* action). This will ensure that, if the initial values of the control variables

[2]It should be noted, that since the control variables can take three values, the implementation would also need to ensure in some way that only valid values are returned when the variables are read. If a two bit variable is used to record the values this may mean mapping two different bit patterns to the same value e.g. $[0,0] \rightarrow 0, [0,1] \rightarrow 1, [1,0] \rightarrow 2, [1,1] \rightarrow 2$.

are as shown in Table 3.3, the incorrect behaviour shown will be guaranteed to occur. A stronger timing constraint that additionally ensures that the accesses to the control variables are atomic may be required to ensure the 3-slot implementation always behaves correctly. A revised 3-slot implementation is given in [XYIS02], where the control actions of the reader are effectively combined into a single operation[3], which maintains coherence provided that the accesses to the control variables are atomic. Correctness proofs for the two implementations: from [Sim90c], if the above timing constraint can be implemented, and from [XYIS02] are given in Appendix I.

The ACMs described above can allow a certain amount of asynchrony between their readers and writers, but cannot be implemented in a fully asynchronous manner. The next section introduces an alternative classification scheme for ACMs and describes some impossibility for ACMs results in terms of this scheme.

3.2 An Implementation Classification Scheme

This section introduces an alternative (implementation) classification scheme of ACMs that is widely used in academic literature, for example [HV01]. This scheme is used to classify Simpson's 4-slot implementation in Section 3.3.

ACMs that are designed to communicate data types with more than two values are referred to as *multivalued*, whereas ACMs that only communicate binary types are called *bits*, and ACMs with single writers (readers) are called 1-*writer* (1-*reader*) ACMs.

Implementations that use different variables, or memory locations, to communicate data and to co-ordinate read and write accesses to the data are called *buffer-based*. The variables used to communicate data are called *buffers*, although they are referred to as *slots* or *tracks* in particular algorithms. In a buffer-based ACM no co-ordination information is passed through the buffers, and data is not passed via the control variables.

A buffer-based shared variable where the reader and writer never access the same buffer at the same time (although they may access the same control variable concurrently) is called *conflict-free* [HS94], or *pure* [BP89a]. An ACM that is conflict-free can be implemented using persistent ACMs for its buffers.

A buffer-based shared variable where the read or write algorithm is only required to read or write once each time the reader or writer, respectively, access the ACM is referred to as *read-once* or *write-once*, respectively. A

[3]Rather than creating a local copy of the slot it has acquired and then indicating the chosen slot, the reader copies the chosen slot directly to the control variable in the mechanism.

non-conflict free ACM is unlikely to be read-once, since some re-reading will be necessary should a conflict occur during a read, in order for the reader to return coherent data. The write-once read-once properties are highly desirable in multivalued ACMs where large complex data structures are being communicated.

3.2.1 Impossibility Results for ACM Implementations

[Pet83] shows that buffer-based 1-Writer ACMs need $n+2$ buffers, where n is the number of readers, in order to be L-atomic. [BP89b] then shows that in order to be conflict-free an ACM needs at least $2n+2$ buffers. This result is consistent with the counter example in the last section and means that four buffers are the minimum requirement for an implementation of a 1-reader 1-writer conflict free ACM. [HV01] shows that it is impossible to realise a conflict-free write-once L-atomic variable from 4 buffers and 4 type-safe control variables. Simpson's 4-slot, which is described in the next section, is a conflict-free write-once ACM, which only uses 4-control variables. This thesis will show that the 4-slot is L-atomic provided it is implemented with control variables that have properties that are stronger than those given by type-safe ACMs.

3.3 Simpson's 4-slot ACM

[Sim90a] defines a fully asynchronous communication mechanism that maintains data-coherence and which uses only four *slots* to communicate the data between the reading and writing processes. The 4-slot, which is described in Section 3.3.1, can be seen as an implementation of a MASCOT pool. The implementation from [Sim90a] is given in Section 3.3.2, and a formal model of the implementation is introduced in Section 3.3.3.

3.3.1 Description of Simpson's 4-slot

Simpson's 1990 4-slot is an implementation of a MASCOT pool, and it is the intention that the 4-slot is L-atomic, although the precise item returned as a result of a read depends on how the actions of the reader and writer interleave.

Figure 3.5 shows an illustration of Simpson's 4-slot ACM, which is described as follows:

1. There are four slots for communication of the data between the writer and reader. The slots are organised into two pairs of two slots, (this organisation into two pairs is used to help ensure that the reader and

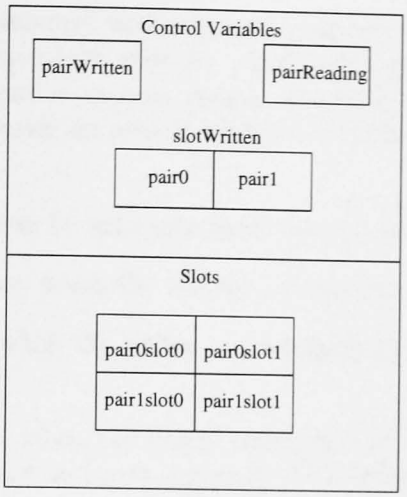

Figure 3.5: Simpson's 4-slot ACM

writer do not access the same slot at the same time). The precise nature of the 4-slot algorithm is described in Section 3.3.2. The ACM is designed to be conflict free, therefore the slots can be implemented using persistent ACMs

2. There are four single bit control variables:

   pairWritten: which is used by the writer to indicate the name of the pair, which contained the slot, that it last wrote to.

   pairReading: which is used by the reader to indicate the name of the pair, which contains the slot, that it is accessing.

   slotWritten: a two element array of binary slot indices, which is accessed by the reader to choose the slot to read from in the pair of slots it is currently accessing, and by the writer to choose the slot to write to in the current pair of slots it is accessing. The writer also uses this array to indicate the latest slot, in each pair of slots, that it has last accessed.

In some previous literature on the 4-slot (e.g. [CXYD98,Cla00,HP02a]) it was asserted or argued that its bit control variables can be implemented using type-safe ACMs. [Rus02] shows that this is not the case, and that the ACM is only regular if the control variables are implemented in this way. This point will be discussed further in Chapter 7, where the ACM is shown to be L-atomic provided the control vari-

ables are implemented with stronger properties than type-safeness[4]. This result is consistent with the proof in [HV01]. The formal models in Chapters 5 and 6 assume atomic access to the control variables, therefore the proofs associated with these models are not affected by this distinction.

The write action can be split into three distinct phases[5]:

- An acquire phase, when the writer acquires a slot to write to.
- A write phase, when the writer can assemble the new item of data in the chosen slot.
- A release phase, when the writer indicates the slot it has written to, by writing the slot and pair names to the relevant control variables in the mechanism.

and the read action can be split into two phases:

- An acquire phase, during which the reader chooses the slot it will read from, and also indicates the name of the pair it will read from, by writing the name to the relevant control variable in the mechanism.
- A read phase, during which time it can read the item from the chosen slot.

The default behaviour of the mechanism occurs when the complete reads and writes interleave with each other, as if those reads and writes are Hoare-atomic. In this case the reader will acquire the latest item that the writer has just released. The acquire and release actions are composed of a number of operations and if the individual acquire and release operations interleave with each other the precise slot chosen by the writer, or reader, may depend on the precise ordering of those operations. For example, when the writer changes pairs at the start of a write the reader cannot follow the writer to the new pair until after the end of the write release action. The reader will continue to read the item in the slot last accessed by the writer, in the opposite pair to the writer, until the writer indicates that it has changed pairs.

[4]The type of ACM required to implement the control variables is not currently included in the taxonomy in Chapter 2.

[5]Simpson prefers to consider the write as consisting of two phases. A write phase, followed by a release phase, when the writer also acquires the slot that it will next write to.

3.3.2 The 4-slot Algorithm

The four slot algorithm is deceptively simple, consisting of only five actions in the *write* operation and four actions in the *read* function, and is shown in Table 3.4.

The algorithm is described as follows:

1. the writer:
   - chooses the pair and the slot within that pair to which it will write the new value - *writerChoosesPair* and *writerChoosesSlot* in Table 3.4 (the write pre-sequence). It always chooses to write to the opposite pair to the one the reader last indicated it was reading from (this will be the pair the initial item was written to until the reader indicates the pair it is reading from for the first time), and the opposite slot in its chosen pair to the one it accessed during the last write;
   - writes the new item to the chosen slot - *write* in Table 3.4; and
   - indicates the slot and pair it has written the data to - *writerIndicatesSlot* and *writerIndicatesPair* in Table 3.4 (the write post-sequence).
2. the reader:
   - chooses to read from the pair of slots last written to (or the pair the initial value was written to if the first read occurs before the first write), indicates that it is reading from that pair, and then chooses to read from the latest slot in that pair that has had a value written to it - *readerChoosesPair*, *readerIndicatesPair* and *readerChoosesSlot* in Table 3.4 (the read pre-sequence); and
   - reads the item from the chosen slot - *read* in Table 3.4.

In terms of the classification scheme in Section 3.2 the 4-slot is a multivalued 1-writer 1-reader buffer-based conflict-free read-once write-once ACM.

3.3.3 A Formal Model of Simpson's 4-slot

This section describes a formal model of Simpson's 4-slot ACM, which is used in the formal proofs in succeeding chapters[6] (The full PVS encoding of the model is given in Appendix D). The formal description of the model is

[6]The only difference is that the proofs use different sets of auxiliary variables to record extra history state of the mechanism in order to verify that the ACM exhibits the desired properties.

Table 3.4: The 4-slot mechanism

```
mechanism four slot;
   type PairIndex = (p0, p1);
        SlotIndex = (s0, s1);
   var slots: array[PairIndex, SlotIndex] of Data :=
                ((init_item,nil),(nil,nil));
       slotWritten: array[PairIndex] of SlotIndex :=
                                                (s0,s0);
       pairWritten, pairReading: PairIndex := p0,p1;

   procedure write (item: data);
   var writerPair: PairIndex;
       writerSlot: SlotIndex;
   begin
      writerPair := not pairReading;          (writerChoosesPair)
      writerSlot :=
            not slotWritten[writerPair];      (writerChoosesSlot)
      slots[writerPair, writerSlot] :=
                                    item;     (write)
      slotWritten[writerPair] :=
                          writerSlot          (writerIndicatesSlot)
      pairWritten := writerPair;              (writerIndicatesPair)
   end;

   function read: Data;
   var readerPair: PairIndex;
       readerSlot: SlotIndex;
   begin
      readerPair := pairWritten;              (readerChoosesPair)
      pairReading := readerPair;              (readerIndicatesPair)
      readerSlot :=
            slotWritten[readerPair];          (readerChoosesSlot)
      read :=
          slots[readerPair, readerSlot];      (read)
   end;
end;
```

written in a VDM-like syntax, [ISO96], because this syntax is more readable. It deviates from VDM-SL in that it uses classical logic (to be compatible with the PVS logic [OSRSC99a]). The variable names are ***hooked*** where appropriate to indicate the values before an operation is executed.

First the basic types are introduced: the ACM communicates data items, which consist of an index number and a value (the index number is used to reason about the ordering writes to the ACM in the proofs of L-atomicity). Enumeration types are used to define the names of the pairs and slots, and the program counters which indicate the next instruction (action) to be executed by the reader and writer. Finally the writer and reader local states, which record the local state of the writer and reader of the ACM respectively (the pair, and slot in that pair, that they last accessed, or are currently accessing) are given.

$Val = token;$

$Data :: index : nat$
$\quad val : Val;$

$PairIndex = p0 \mid p1;$

$SlotIndex = s0 \mid s1;$

$NextReadInstruction = \text{RCP} \mid \text{RIP} \mid \text{RCS} \mid \text{RD};$

$NextWriteInstruction = \text{WCP} \mid \text{WCS} \mid \text{WR} \mid \text{WIS} \mid \text{WIP};$

$WriterState :: writerPair : PairIndex$
$\quad writerSlot : SlotIndex;$

$ReaderState :: readerPair : PairIndex$
$\quad readerSlot : SlotIndex;$

The ACM consists of:

1. two control variables, called *pairWritten* and *pairReading*, which record the pair the writer and reader have last accessed (or are accessing), respectively;

2. a two element array, called *slotWritten*, which the writer uses to indicate the slot it has last accessed, or is accessing, in each pair of slots, and which is used by the reader and writer to choose the slot they

are going to access in whichever pair they have chosen to read from or write to:

3. *slots*, which is a two dimensional array to represent the four data slots that are used for communicating data items between the reader and writer. One of the slots is initialised with an initial item of data (called *"initVal"* in the model), and the other slots are not initialised (initialised with the value *nil* in the model);

4. two variables called *nri* and *nwi*, which are used to model the program counter of the reader and writer of the ACM respectively. For example *nri* is of type *NextReadInstruction*, and it records the next operation that is to be executed by the reader;

5. a *writer* of type *WriterState* and a *reader* of type *ReaderState*.

$$
\begin{array}{l}
\textsf{state } \mathit{Conc\_State} \textsf{ of} \\
\quad \mathit{pairWritten} : \mathit{PairIndex} \\
\quad \mathit{slotWritten} : \mathit{PairIndex} \xrightarrow{m} \mathit{SlotIndex} \\
\quad \mathit{pairReading} : \mathit{PairIndex} \\
\quad \mathit{slots} : \mathit{PairIndex} \times \mathit{SlotIndex} \xrightarrow{m} \mathit{Data} \\
\quad \mathit{nri} : \mathit{NextReadInstruction} \\
\quad \mathit{nwi} : \mathit{NextWriteInstruction} \\
\quad \mathit{writer} : \mathit{WriterState} \\
\quad \mathit{reader} : \mathit{ReaderState} \\
\quad \textsf{init } s \triangleq s = \textsf{mk-}\mathit{Conc\_State}\,(p0, \{p0 \mapsto s0, p1 \mapsto s0\}, p1, \\
\qquad\qquad\qquad\qquad \{(p0, s0) \mapsto \textsf{mk-}\mathit{Data}\,(1, \textsf{mk-token}\,(\text{"}\mathit{initVal}\text{"})), \\
\qquad\qquad\qquad\qquad (p0, s1) \mapsto \mathit{nil}, (p1, s0) \mapsto \mathit{nil}, \\
\qquad\qquad\qquad\qquad (p1, s1) \mapsto \mathit{nil}\}, \mathit{rcp}, \mathit{wcp}, \\
\qquad\qquad\qquad\qquad \textsf{mk-}\mathit{WriterState}\,(p0, s0), \\
\qquad\qquad\qquad\qquad \textsf{mk-}\mathit{ReaderState}\,(p1, s1)) \\
\textsf{end}
\end{array}
$$

This model has five write operations and four read operations, each of which is equivalent to one write or read action, respectively, from the 4-slot implementation given in Section 3.3. The write operations are *writerChoosesPair*, *writerChoosesSlot*, *write*, *writerIndicatesSlot* and *writerIndicatesPair*, which are described as follows:

writerChoosesPair: which has the pre-condition that *nwi* (the next write instruction) is WCP. This operation chooses the pair the writer will access during the write operation, which is written to the local variable (*writerPair*) of the writer. It also changes the value of *nwi* to WCS.

$writerChoosesPair\ ()$
ext wr $nwi : nextWriteInstruction$
wr $writer.writerPair : PairIndex$
rd $pairReading : PairIndex$
pre $nwi = \text{WCP}$
post $nwi = \text{WCS} \wedge (\overleftarrow{pairReading} = p0 \Rightarrow writer.writerPair = p1) \wedge (\overleftarrow{pairReading} = p1 \Rightarrow writer.writerPair = p0)$;

writerChoosesSlot: this operation has the pre-condition that nwi is WCS. The writer chooses to acquire the opposite slot to the one it last accessed in its chosen pair[7], and writes the chosen slot to the local variable *writerSlot*. The operation also sets the value of nwi to WR.

$writerChoosesSlot\ ()$
ext wr $nwi : nextWriteInstruction$
wr $writer.writerSlot : SlotIndex$
rd $slotWritten : PairIndex \xrightarrow{m} SlotIndex$
pre $nwi = \text{WCS}$
post $nwi = \text{WR} \wedge (\overleftarrow{slotWritten}(\overleftarrow{writer.writerPair}) = s0) \Rightarrow writer.writerSlot = s1) \wedge (\overleftarrow{slotWritten}(\overleftarrow{writer.writerPair}) = s1 \Rightarrow writer.writerSlot = s0)$;

write: during this operation the writer writes the new item to the slot it has chosen to acquire. The pre-condition is that the value of nwi is WR and the operation sets it equal to WIS.

$write\ (v : Data)$
ext wr $nwi : nextWriteInstruction$
wr $slots : PairIndex \times SlotIndex \xrightarrow{m} Data$
pre $nwi = \text{WR}$
post $nwi = \text{WIS} \wedge \underleftarrow{slots} = slots \dagger \{(\overleftarrow{writer.writerPair}, \overleftarrow{writer.writerSlot}) \mapsto v\}$;

[7]Once again the writer is attempting to avoid the reader, because the reader may be reading from the slot that the writer last accessed in this pair.

writerIndicatesSlot: the pre-condition for this operation is that the value of *nwi* is WIS. The operation writes the name of the slot that the writer has accessed during this write to the appropriate element of the *slotWritten* array (it *indicates* the slot the writer has accessed), and sets the value of *nwi* to WIP.

writerIndicatesSlot ()
ext wr *nwi* : *nextWriteInstruction*
wr *slotWritten* : $PairIndex \xrightarrow{m} SlotIndex$
rd *writer.writerSlot* : *SlotIndex*
pre $nwi = \text{WIS}$
post $nwi = \text{WIP} \wedge slotWritten =$
$\overleftarrow{slotWritten} \dagger \{(\overleftarrow{writer.writerPair} \mapsto \overleftarrow{writer.writerSlot}\};$

writerIndicatesPair: the pre-condition of this operation is that the value of *nwi* is WIP. The operation indicates the pair that the writer has accessed, by writing the name of the pair to the *pairWritten* control variable in the mechanism, and changes the value of *nwi* to WCP.

writerIndicatesPair ()
ext wr *nwi* : *nextWriteInstruction*
wr *PairWritten* : *PairIndex*
rd *writer.writerPair* : *PairIndex*
pre $nwi = \text{WIP}$
post $nwi = \text{WCP} \wedge pairWritten = \overleftarrow{writer.writerPair};$

The four read operations are *readerChoosesPair*, *readerIndicatesPair*, *readerChoosesSlot* and *read*, which are described below:

readerChoosesPair: the pre-condition of this operation is that the value of *nri* is RCP. The operation chooses the pair for the reader to access, by copying the value of *pairWritten* to the reader local control variable *readerPair*, and sets the value of *nri* to RIP.

readerChoosesPair ()
ext wr *nri* : *nextReadInstruction*
wr *reader.readerPair* : *PairIndex*
rd *pairWritten* : *PairIndex*
pre $nri = \text{RCP}$
post $nri = \text{RIP} \wedge reader.readerPair = \overleftarrow{pairWritten};$

readerIndicatesPair: the pre-condition of this operation is that *nri* is equal to RIP. It indicates the pair that the reader has chosen to access, by copying the value of *readerPair* to the control variable *pairReading*, and sets the value of *nri* to RCS.

readerIndicatesPair ()
ext wr *nri* : *nextReadInstruction*
wr *pairReading* : *PairIndex*
rd *reader.readerPair* : *PairIndex*
pre $nri = \text{RIP}$
post $nri = \text{RCS} \wedge pairReading = \overleftarrow{reader.readerPair}$;

readerChoosesSlot: the pre-condition of this operation is that the value of *nri* is RCS. The operation sets the value of *readerSlot* to the name of the slot the reader is going to access, by copying the value from the element of the *slotWritten* array relating to the reader's chosen pair. It also sets the value of *nri* to RD.

readerChoosesSlot ()
ext wr *nri* : *nextReadInstruction*
wr *reader.readerSlot* : *SlotIndex*
rd $slotWritten : PairIndex \xrightarrow{m} SlotIndex$
pre $nri = \text{RCS}$
post $nri = \text{RD} \wedge reader.readerSlot = \overleftarrow{slotWritten}(\overleftarrow{reader.readerPair})$;

read: the pre-condition of this operation is that the value of *nri* is RD. It reads the item from the slot that the reader has chosen to acquire and sets the value of *nri* to RCP.

read () v : *Data*
ext wr *nri* : *nextReadInstruction*
rd $slots : PairIndex \times SlotIndex \xrightarrow{m} Data$
pre $nri = \text{RD}$
post $nri = \text{RCP} \wedge v = \overleftarrow{slots}(\overleftarrow{reader.readerPair}, \overleftarrow{reader.readerSlot})$;

3.4 Summary

This chapter describes a number of ACM implementations with less than 4 slots for communication of data between their reader(s) and writer, and shows that none of these implementations can be implemented in a fully asynchronous manner. It then introduces an alternative (implementation) classification scheme for ACMs and some results from related work, which prove that it is impossible to implement a single-reader, single-writer conflict free ACM with less than 4-slots, and that it is also impossible to implement such an ACM with fewer than 5 type-safe control variables. Simpson's fully asynchronous 4-slot ACM implementation, which uses 4 control variables, and a formal model of the 4-slot are described and the remainder of this thesis demonstrates how it is possible to verify that the 4-slot ACM is L-atomic, provided it can be implemented with control variables which give stronger guarantees than type-safeness. First Chapter 4 introduces an abstract model of L-atomicity, then Chapter 5 shows how Simpson's ACM can be shown to be a refinement of this model subject to certain assumptions about the atomicity of the actions of its reader and writer. Chapter 6 then uses a rely-guarantee proof method to verify that the implementation is L-atomic when these atomicity assumptions are relaxed, and the reader and writer actions can interleave in an unconstrained manner, and finally Chapter 7 describes some related work which verifies that realistic implementations of the 4-slot are L-atomic when the reader and writer actions are fully asynchronous, using models in CSP with the FDR model checker. This demonstrates how an understanding, and confidence in the correctness (with respect to its requirements), of asynchronous systems can be gained in an incremental manner, using a range of tools, to help reduce the amount of rework that is required when developing such systems.

Chapter 4

A Model of L-atomicity

This thesis demonstrates how it is possible to gain an understanding of the behaviour, and verify properties, of asynchronous systems in an incremental manner. The specification and development of asynchronous systems is difficult, because the specification is often complex, and components in fully asynchronous systems, with apparently simple specifications, may interact in unexpected ways. For these reasons it may be difficult to move directly to a model of the implementation, and to understand the model sufficiently well to be able to verify that it exhibits the desired properties. However, by starting with an abstract model of those properties, it is possible to gain valuable insights into the behaviour of the system by building and verifying more complex and realistic models as understanding increases, until sufficient confidence is gained in the correctness of the implementation. This process can also help to eliminate errors and ambiguities in the specification of the system that can be costly to correct in the later stages of development. For example errors often arise because the unexpected interactions of its components. Also the use of formal modelling techniques allows the developer to explore properties of the system to help to identify flaws in the specification. Identifying and fixing these errors may require extra effort in the earlier stages of development, but this extra effort can be recovered because of the reduction in the number of errors found in the later stages. This chapter describes the first part of the process, which is to build an abstract model of the system and verify that the model exhibits the properties that are required of the implementation. This model is then used in subsequent chapters to verify properties an ACM implementation, Simpson's 4-slot, which is used as a case study.

A desirable property of any ACM is that it will provide its reader with coherent fresh data as the result of a read: these are the properties of an L-atomic ACM (as described in Section 2.5). A formal definition of L-atomicity is given in Section 2.4, however Section 2.6 describes the difficulties in using

this formal definition directly in verifying properties of ACM implementations. for example the formal definition is difficult to understand. Chapter 3 described a number of ACM implementations, which allow different levels of asynchrony between their readers and writers, and a fully asynchronous implementation, Simpson's 4-slot, was described in Section 3.3. The remainder of this thesis describes how an incremental development method was used to verify that the 4-slot is L-atomic. The incremental approach uses a number of different tools to explore and verify properties of increasingly realistic models of the implementation. This chapter introduces a formal model of L-atomicity which forms the basis of these investigations, and provides an easier to understand model against which to verify properties of the implementation, thus overcoming the difficulty mentioned above, with the formal definition of L-atomicity.

This chapter is organised as follows. First Section 4.1 describes an abstract model of L-atomicity, and gives an informal proof that the model is equivalent to the formal definition of L-atomicity in Section 2.4. Section 4.2 then describes how the model has been verified to be L-atomic using an exhaustive proof method similar to that described in [Ash75].

4.1 The (Abstract) Model

The properties of L-atomic ACMs were described in Section 2.2.2 and can be summarised as follows: the reader will always read globally fresh data; and reads and writes appear to have occurred in a particular order (as if the entire read and write operations were Hoare atomic [Hoa71] and interleaved with each other).

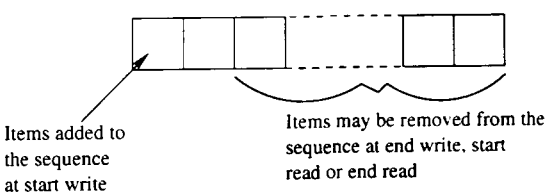

Figure 4.1: Sequence of items

This section describes an abstract model of L-atomicity, where the approach is taken of modelling the items that are written to the ACM as a sequence, which gives the order in which they were written. Items may be removed from the sequence because they are overwritten by a later write, or because a later item has been read. The presence of an item in the sequence models its availability to the reader, and there are four operations in

the model, *start_write*, *end_write*, *start_read* and *end_read*, which add items to and remove items from the sequence (as illustrated in Figure 4.1). The model is described below and an informal proof that it is equivalent to the L-atomic ACM as defined in Chapter 2 is given. As in the case of the formal model of the 4-slot in Section 3.3.3 the formal description of the model is written in a VDM-like syntax, and deviates from VDM-SL in that it uses classical logic (to be compatible with the PVS logic). The full model in the PVS logic is given in Appendix E and Appendix A describes the translation to the PVS logic.

Data items: the items transmitted between the reader and writer have unique serial numbers, starting at one and incrementing by one for each successive item written, which are recorded in the *index* field of the record. The data transmitted is represented by the *val* field; the type of the data is not important and is represented as a token.

$$Val = \mathsf{token};$$

$$Data :: index : \mathbb{N}$$
$$val : Val;$$

ACM State: the ACM itself is represented by a sequence of data items: the writer adds new items to the head end of the sequence and items are removed from the tail end when they are no longer available to be read. The sequence is initialised with a data item, sequence number one, so that an item is available if the first read occurs before the first write. There are a number of auxiliary variables in the model. Two booleans, called *readerAccess* and *writerAccess*, record whether the reader and/or writer are accessing the mechanism. *readerAccess* is set to true at *start_read* and false at *end_read*, and similarly *writerAccess* is set to true at *start_write* and false at *end_write*. These variables are also used in the pre-conditions of the operations, for example: *pre-start_read* $\triangleq \neg$*readerAccess*; and *pre-end_write* $\triangleq$ *writerAccess*. Further auxiliary variables *nextIndex*, *indexRead* and *firstIndex* record the indices of the next item to be written, the last item read and the first item available to be read during a read operation. These variables are used in the abstract model to ensure that the ACM modelled does behave in an L-atomic manner as described in Section 4.2 and to verify that Simpson's 4-slot implementation is a refinement of this model as described in Chapter 5:

- *firstIndex*, which is set equal to the index of the first item that is available to the reader, by *start_read* (the item at the tail end of the sequence after the operation is executed).

- *nextIndex* is the index number given to the next item to be written, which is incremented by one at *start_write*. The latest item available to the reader always has index number of one less that *nextIndex*.
- *indexRead*, which is set equal to the index of the item read at *end_read*.

Provided that *indexRead* is greater than or equal to *firstIndex* and less than *nextIndex* whenever *end_read* is executed the model guarantees L-atomicity.

state Abs_State of
  $vals : Data^+$
  $writerAccess : \mathbb{B}$
  $readerAccess : \mathbb{B}$
  $nextIndex : \mathbb{N}$
  $indexRead : \mathbb{N}$
  $firstIndex : \mathbb{N}$
  init $s \triangleq s =$ mk-Abs_State ([mk-$Data$ (1, mk-token ("$initItem$"))], false, false, 2, 0, 0)
end

Descriptions of the 4 operations in the model follow:

start_write: adds the new item, that is being written, to the head of the sequence. If the operation is executed a number of times during a single read a new item is added to the sequence on each occasion. This makes the new item(s) available to the reader.

$start\_write$ ()
ext wr $vals : Data^+$
  wr $writerAccess : \mathbb{B}$
  wr $nextIndex : \mathbb{N}$
pre $\neg\, writerAccess$
post let $newI =$ mk-$Data$ ($\overleftarrow{nextIndex}$, mk-token ("$newI$")) in
  $writerAccess \wedge vals = [newI] \curvearrowright \overleftarrow{vals} \wedge$
  $nextIndex = \overleftarrow{nextIndex} + 1$

end_write: if there is a read in progress at end write the sequence is left unchanged. If there is not a read in progress all of the items are removed from the sequence apart from the one just written.

end_write ()
ext wr $vals : Data^+$
wr $writerAccess : \mathbb{B}$
rd $readerAccess : \mathbb{B}$
pre $writerAccess$
post $\neg\, writerAccess \wedge (\neg\, readerAccess \Rightarrow vals = [\mathsf{hd}\ \overleftarrow{vals}]) \wedge$
$(readerAccess \Rightarrow vals = \overleftarrow{vals})$

start_read: if the sequence has more than one item removes all of the items that are not available to be read as follows: if there is a write in progress the sequence is shortened to contain only the last item written and the item being written by the current write (the first and second items in the sequence); and if there is no write in progress the sequence is shortened to contain only the last item written (the item at the head of the sequence). The operation also sets *firstIndex* equal to the index of the oldest item available to be read, which is the item that will be at the tail of the sequence after the operation has been executed. If the sequence only contains a single item it is left unchanged.

start_read ()
ext wr $vals : Data^+$
wr $readerAccess : \mathbb{B}$
wr $firstIndex : \mathbb{N}$
pre $\neg\, readerAccess$
post $readerAccess \wedge$
$(\mathsf{len}\ \overleftarrow{vals} = 1 \Rightarrow firstIndex = (\mathsf{hd}\ \overleftarrow{vals}).index) \wedge$
$(\mathsf{len}\ \overleftarrow{vals} > 1 \Rightarrow (\neg\, \overleftarrow{writerAccess} \Rightarrow vals = [\mathsf{hd}\ \overleftarrow{vals}] \wedge$
$firstIndex = (\mathsf{hd}\ \overleftarrow{vals}).index) \wedge$
$(\overleftarrow{writerAccess} \Rightarrow vals = \overleftarrow{vals}(1, \ldots, 2) \wedge$
$firstIndex = \overleftarrow{vals}(2).index))$

end_read: chooses an item to read, sets *indexRead* equal to the index of the item chosen, and removes all of the items from the sequence that are older than the one chosen.

end_read () $v : Val$
ext wr $vals : Data^+$
wr $readerAccess : \mathbb{B}$
wr $indexRead : \mathbb{N}$

pre *readerAccess*

post $\neg$ *readerAccess* $\wedge$ ($\exists$ $i \in$ inds $\overleftarrow{vals} \cdot v = \overleftarrow{vals}(i).val \wedge$
$indexRead = \overleftarrow{vals}(i).index \wedge vals = \overleftarrow{vals}(1, ..., i)$)

It is noted that in actual ACM implementations, subsequent writes always overwrite previous items in the ACM, but it is not practical to encode this property into the model. This is because, if a read is in progress when the *end_write* operation is executed there is no way of knowing which of the items in the sequence the reader has chosen to return as a result of the read. All of the items in the sequence at *end_write* must still be available when the read subsequently ends, therefore the sequence is not shortened at the end of the write in these circumstances.

An informal argument that the model is equivalent to the definition of L-atomicity in Section 2.4 is given below.

- The definition of L-atomicity uses the auxiliary function, *r_communicates*, which defines the items that are available to be read: those written by any writes that overlap with the read, and the item written by the last write that finished before the read started. Therefore if there are no overlapping writes only the item written immediately prior to the start of the read is available to be read. The definition of L-atomicity then states that the reader will read one of the available items, and the index of the item read will be greater than or equal to the index of the last item read.

- The formal model constructs a sequence equivalent to the set of items available to the reader as follows:

  1. At start write the writer adds the new item, which is going to be written, to the sequence. This ensures that any item written by a write that overlaps with a read is available to be read.
  2. At end write, if there is no read in progress the writer shortens the sequence to contain only the head item, the one that has just been written. This ensures that, if the next action is the start of a read, the only item available to be read is the one that has just been written. If there is a read in progress the writer leaves the sequence unchanged so that the item that has been written during the read is available to the reader, as well as any previous items (which include the item written immediately before the read starts). Each subsequent write that occurs while the read is in progress similarly adds an additional item to the sequence (the set of *DataItem ids* constructed by the *r_communicates* operation will contain all of the indices of the items in this sequence).

3. At start read, if there is only a single item in the sequence of items this is the only item available to be read at that time. The reader sets *firstIndex* equal to the index of this item.
4. If the length of the sequence is greater than 1 at start read there are two options. If there is no write in progress *firstIndex* is set equal to the index of the item at the head of the sequence, which is shortened to include only this item: this is the index of the last item written. If there is a write in progress the item at the head of the sequence is the one being written: *firstIndex* is set equal to the index of the second item on the sequence (the index of the item last written), and the sequence is shortened to include only the first two items. This ensures that, when a read starts, any items that are not available to the reader are discarded from the sequence.

- At end read the reader returns one of the items from the sequence constructed as above, and shortens the sequence to remove all items older than the one read. This ensures that the reader returns a fresh item, and that it cannot return an older item at the next read, so the items must be read in the order that they are written.□

A full formal proof of equivalence is not given for the following reasons:

- The definition of L-atomicity in the taxonomy is not self contained; it builds on the definitions of the the other ACMs in the taxonomy and adds the extra guarantee that items will be read in order. The proof would therefore need to relate to a number of different definitions in the taxonomy.
- The two models use different paradigms: the definition of L-atomicity in RTL is a *trace model.* It would be necessary to derive a trace semantics for the procedural model of L-atomicity in this chapter and define proof method in order to verify equivalence.

4.2 Verification of the Model of L-atomicity

The model given in the previous section has been verified to be L-atomic using an exhaustive proof method similar to that described in[Ash75]. Ashcroft's method used the same global invariant in each state to verify the correctness of parallel programs. Here different invariants are used, and correctness proofs are completed, for all locations in the state machine of the model, with PVS. The state space of the model is shown in Figure 4.2, which is described below.

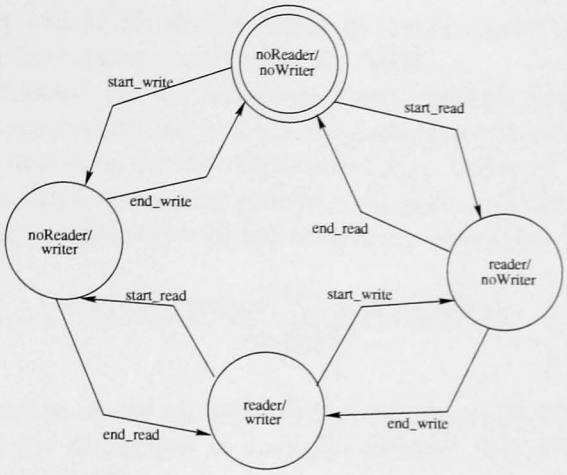

Figure 4.2: The State Space of the Model of L-atomicity

1. The start location of the model is the *noReader/noWriter* location, indicated by the double circle.

2. Each of the locations has two outgoing transitions from it: each transition is associated with an operation of one of the component processes (one of the outgoing transitions has a write operation associated with it, and the other has a read operation associated with it), and may have a guard. In this case the guard for each transition is *true* and is omitted.

3. Each of the locations has an assertion associated with it, composed of invariant properties that hold when the model is in that location. These assertions are used to verify properties of the model.

For example the assertions for the noReader/writer and reader/writer locations are:

$$\begin{aligned} noReader\_writer\_Assertion \triangleq\ & indexRead \leq nextIndex - \mathsf{len}\ vals \wedge \\ & firstIndex \leq nextIndex - \mathsf{len}\ vals \wedge \\ & vals(1).index = nextIndex - 1 \\ reader\_writer\_Assertion \triangleq\ & indexRead \leq nextIndex - \mathsf{len}\ vals \wedge \\ & firstIndex = nextIndex - \mathsf{len}\ vals \wedge \\ & vals(1).index = nextIndex - 1; \end{aligned}$$

It is interesting to note that there is a certain symmetry about the assertions in the locations of the model: the two assertions where the reader

is not accessing the ACM are the same; as are the assertions for the two locations where the reader is accessing the ACM.

Conjectures based on the following general scheme have been proved for each of the operations using PVS, to show that the operations do not invalidate the assertions in the respective target states of the transitions associated with those operations (where $Ass1$ and $Ass2$ are the assertions in the source and target states of the assertions respectively):

$$\frac{Ass1(\overleftarrow{\sigma}); pre\_op(\overleftarrow{\sigma}); post\_op(\overleftarrow{\sigma}, \sigma)}{Ass2(\sigma)}$$

The next section shows an interesting example proof: the remainder of the conjectures are discharged in a similar manner, and so the rest of the proofs are not described[1].

4.2.1 A Rigorous Proof for the end_read Operation

This section gives a rigorous example proof, for the *end_read* operation. The rigorous proof uses the natural deduction proof style [BFL+94] [Jon90]. While the structure of the rigorous proof is different from that of the formal proofs in PVS, they are included to illustrate the principles behind the formal proofs, and to help to increase confidence in the correctness of those proofs. The *end_read* operation can be executed from the states *reader/noWriter*, where *writerAccess* is false, and *reader/writer*, where *writerAccess* is true, but the assertions in the resultant states are identical. Therefore there is no need for a case distinction to discharge the proof. The conjecture is shown below, where $Ass1$ is the *reader_writer_Assertion* and $Ass2$ is the *noReader_writer_Assertion* from above, and the definition of the invariant of the model follows (the first conjunct of the invariant is given in the PVS model in Appendix E using a sub-type definition).

$$\frac{Ass1(\overleftarrow{\sigma}); pre\_end\_read(\overleftarrow{\sigma}); post\_end\_read(\overleftarrow{\sigma}, \sigma); inv(\overleftarrow{\sigma})}{Ass2(\sigma); inv(\sigma)}$$

$$inv \triangleq \mathsf{len}\ vals \geq 1 \wedge (\forall i \in \mathsf{inds}\ vals \cdot i < \mathsf{len}\ vals \Rightarrow vals(i).index = vals(i+1).index + 1);$$

For brevity only the names of the components of the state of the model are given in the proof, for example $\sigma.nextIndex$ is called *nextIndex*, and the values are *hooked*, where appropriate (to indicate the values before an

[1]The interested reader can download the PVS theory, and proof scripts, from http://homepages.cs.ncl.ac.uk/neil.henderson/fme2002/4slot.tgz.

operation is executed). For convenience the definition of *end_read* is repeated below:

end_read () v : *Val*
ext wr $vals : Data^+$
wr $readerAccess : \mathbb{B}$
wr $indexRead : \mathbb{N}$
pre $readerAccess$
post $\neg\, readerAccess \wedge (\exists\, i \in \mathsf{inds}\ \overleftarrow{vals} \cdot v = \overleftarrow{vals}(i).val \wedge$
$indexRead = \overleftarrow{vals}(i).index \wedge vals = \overleftarrow{vals}(1, ..., i))$

The proof relies on the lemmas given below (the names of the lemmas are shown in the boxes to their left), and rigorous proofs of the lemmas, that the invariant holds after the operation is executed, and of the conjecture follow:

$$\boxed{\text{seqIndsUnchanged}}\quad \frac{post\_end\_read(\overleftarrow{\sigma}, \sigma)}{\forall i \in \mathsf{inds}\ vals \cdot vals(i) = \overleftarrow{vals}(i)}$$

$$\boxed{\text{postRd}}\quad \frac{post\_end\_read(\overleftarrow{\sigma}, \sigma)}{vals = \overleftarrow{vals}(1, ..., i) \wedge indexRead = \overleftarrow{vals}(i).index}$$

from $post\_end\_read(\overleftarrow{\sigma}, \sigma)$
1 from $i \in \mathsf{inds}\ \overleftarrow{vals}; v = \overleftarrow{vals}(i) \wedge indexRead = \overleftarrow{vals}(i).index \wedge$
$vals = \overleftarrow{vals}(1, ..., i)$
1.1 $vals = \overleftarrow{vals}(1, ..., i)$ ∧-E-left(1.h2)
1.2 $vals(i) = \overleftarrow{vals}(i)$ sequences(1.1)
infer $\forall i \in \mathsf{inds}\ vals \cdot vals(i) = \overleftarrow{vals}(i)$ 1.h1,1.2
infer $\forall i \in \mathsf{inds}\ vals \cdot vals(i) = \overleftarrow{vals}(i)$ ∃-E(h1.1)

from $post\_end\_read(\overleftarrow{\sigma}, \sigma)$

1 from $i \in \text{inds } \overleftarrow{vals}$; $v = \overleftarrow{vals}(i) \wedge indexRead = \overleftarrow{vals}(i).index \wedge$
$vals = \overleftarrow{vals}(1, ..., i)$

1.1 $vals = \overleftarrow{vals}(1, ..., i)$ $\wedge$-E-left(1.h2)

1.2 $v = \overleftarrow{vals}(i) \wedge indexRead = \overleftarrow{vals}(i).index$ $\wedge$-E-right(1.h2)

1.3 $indexRead = \overleftarrow{vals}(i).index$ $\wedge$-E-left(1.2)

infer $vals = \overleftarrow{vals}(1, ..., i) \wedge indexRead = \overleftarrow{vals}(i).index$ $\wedge$-I(1.1,1.3)

infer $vals = \overleftarrow{vals}(1, ..., i) \wedge indexRead = \overleftarrow{vals}(i).index$ $\exists$-E(h1,1)

from $inv(\overleftarrow{\sigma})$

1 $vals = \overleftarrow{vals}(1, ..., i) \wedge indexRead = \overleftarrow{vals}(i).index$ lemma postRd

2 $vals = \overleftarrow{vals}(1, ..., i)$ ∧-E-right(1)

3 len $vals \geq 1$ 2.len

4 from $i \in$ inds $vals$

4.1 $\forall i \in \text{inds } \overleftarrow{vals} \cdot i < \text{len } \overleftarrow{vals} \Rightarrow \overleftarrow{vals}(i).index = \overleftarrow{vals}(i+1).index + 1$ ∀-E(h1)

4.2 from $i <$ len $vals$

4.2.1 $i <$ len $\overleftarrow{vals}$ 4.2.h1.2, len

4.2.2 $i \in$ inds $\overleftarrow{vals}$ 4.2.1, len

4.2.3 $i <$ len $\overleftarrow{vals} \Rightarrow \overleftarrow{vals}(i).index = \overleftarrow{vals}(i+1).index + 1$ ∀-E(4.1,4.2.2)

4.2.4 $\overleftarrow{vals}(i).index = \overleftarrow{vals}(i+1).index + 1$ ⇒-E(4.2.1,4.2.3)

4.2.5 $\forall i \in \text{inds } vals \cdot vals(i) = \overleftarrow{vals}(i)$ lemma seqIndsUnchanged

4.2.6 $vals(i) = \overleftarrow{vals}(i)$ 4.h1, 4.2.5

4.2.7 $vals(i+1) = \overleftarrow{vals}(i+1)$ 4.h1, 4.2.5, ℕ

infer $vals(i).index = vals(i+1).index + 1$ =-subs(4.2.6, 4.2.7, 4.2.4)

infer $i <$ len $vals \Rightarrow vals(i).index = vals(i+1).index + 1$ ⇒-I(4.2)

5 $\forall i \in \text{inds } vals \cdot i <$ len $vals \Rightarrow vals(i).index = vals(i+1).index + 1$ ∀-I(4)

infer $inv(\sigma)$ ∧-I(3,5)

from $Ass1(\overleftarrow{\sigma})$; $pre\_end\_read(\overleftarrow{\sigma})$; $post\_end\_read(\overleftarrow{\sigma}, \sigma)$	
1 $nextIndex = \overleftarrow{nextIndex}$	ext-post_end_read-defn
2 $vals = \overleftarrow{vals}(1, ..., i) \wedge indexRead = \overleftarrow{vals}(i).index$	lemma postRd
3 $vals = \overleftarrow{vals}(1, ..., i)$	∧-E-right(2)
4 len $vals = i$	3.len
5 $\overleftarrow{vals}(1).index = \overleftarrow{nextIndex} - 1$	∧-E(h1)
6 $vals(1).index = \overleftarrow{vals}(1).index$	inds (3)
7 $vals(1).index = nextIndex - 1$	=-subs(6,1,5)
8 $\overleftarrow{indexRead} \leq \overleftarrow{nextIndex}$ − len $\overleftarrow{vals}$	∧-E(h1)
9 $indexRead = \overleftarrow{vals}(i).index$	∧-E-left(2)
10 $indexRead = (\overleftarrow{vals}(1).index$ − len $vals) + 1$	3,9,ℕ
11 $indexRead = (\overleftarrow{nextIndex} - 1$ − len $vals) + 1$	=-subs(5,10)
12 $indexRead \leq nextIndex$ − len $vals$	=-subs (1,11),ℕ
13 $\overleftarrow{firstIndex} = \overleftarrow{nextIndex}$ − len $\overleftarrow{vals}$	∧-E(h1)
14 $firstIndex = \overleftarrow{firstIndex}$	ext-post_end_read-defn
15 len $vals \leq$ len $\overleftarrow{vals}$	3,4,ℕ
16 $firstIndex \leq nextIndex$ − len $vals$	=-subs(14,15,1,13),ℕ
infer $Ass2(\sigma)$	∧-I(7,12,16)

Verification of L-atomicity

Finally the model is verified to be L-atomic by showing that the following assertion always holds after the *end_read* operation is executed:

$$\begin{aligned} L\text{-}atomic(\overleftarrow{\sigma}, \sigma) \triangleq\ & \overleftarrow{indexRead} \leq indexRead \wedge \\ & firstIndex \leq indexRead \wedge \\ & nextIndex - 1 \geq indexRead; \end{aligned}$$

which ensures that the items are read from the sequence as required. The assertion is described as follows:

1. Each data item that is written to the mechanism is given an index number, starting at 1, and increasing each time a new item is written. New items are written to the head (index 1) of the sequence.

2. *firstIndex* gives the index number of the item at the tail of the sequence

after a read starts (the first item that is available to the reader for that read).

3. *indexRead* is the index number of the item that has been read.

The above assertion guarantees first that the item read has an index number greater than or equal to the number of the first item available at the start of the read, and less than the index to be used for the next item written. This ensures that the item read is fresh. Second it ensures that the index of the item read is greater than or equal to the index of the item read the previous time. This ensures that the items are read in order.

The following conjecture has been discharged to show that the model complies with the above assertion:

$$\frac{Ass1(\overleftarrow{\sigma}); pre\_end\_read(\overleftarrow{\sigma}); post\_end\_read(\overleftarrow{\sigma}, \sigma)}{L\text{-}atomic(\overleftarrow{\sigma}, \sigma)}$$

Where Ass1 is the assertion that holds in the states where *end_read* can be executed (*reader_writer_Assertion* from above). The proof relies on the *indexAfterEndRd* lemma which is given below. Rigorous proofs of the lemma and the proof obligation follow.

$$\boxed{\text{indexAfterEndRd}}\quad \frac{Ass1(\overleftarrow{\sigma}); pre\_end\_read(\overleftarrow{\sigma}); post\_end\_read(\overleftarrow{\sigma}, \sigma)}{indexRead = \overleftarrow{nextIndex} - \mathsf{len}\ vals}$$

from $Ass1(\overleftarrow{\sigma}); pre\_end\_read(\overleftarrow{\sigma}); post\_end\_read(\overleftarrow{\sigma}, \sigma)$	
1 $vals = \overleftarrow{vals}(1, ..., i) \wedge indexRead = \overleftarrow{vals}(i).index$	lemma postRd
2 $\overleftarrow{vals}(1).index = \overleftarrow{nextIndex} - 1$	$\wedge$-E(h1)
3 $indexRead = \overleftarrow{vals}(i).index$	$\wedge$-E-left(1)
4 $indexRead = (\overleftarrow{vals}(1).index - \mathsf{len}\ vals) + 1$	3.N
5 $indexRead = (\overleftarrow{nextIndex} - 1 - \mathsf{len}\ vals) + 1$	=-subs 2.4
infer $indexRead = \overleftarrow{nextIndex} - \mathsf{len}\ vals$	5.N

from $Ass1(\overleftarrow{\sigma})$; $pre\_end\_read(\overleftarrow{\sigma})$; $post\_end\_read(\overleftarrow{\sigma}, \sigma)$	
1 $nextIndex = \overleftarrow{nextIndex}$	ext-post-end-read-defn
2 $indexRead = \overleftarrow{indexRead}$	ext-post-end-read-defn
3 $firstIndex = \overleftarrow{firstIndex}$	ext-post-end-read-defn
4 $vals = \overleftarrow{vals}(1, ..., i) \wedge indexRead = \overleftarrow{vals}(i).index$	lemma postRd
5 $vals = \overleftarrow{vals}(1, ..., i)$	$\wedge$-E-right(4)
6 len $vals \leq$ len $\overleftarrow{vals}$	5,$\mathbb{N}$
7 $\overleftarrow{indexRead} \leq \overleftarrow{nextIndex} -$ len $\overleftarrow{vals}$	$\wedge$-E(h1)
8 $\overleftarrow{indexRead} = \overleftarrow{nextIndex} -$ len $vals$	lemma indexAfterEndRd
9 $\overleftarrow{indexRead} \leq indexRead$	=-subs(8,7,6),$\mathbb{N}$
10 $\overleftarrow{vals}(1).index = \overleftarrow{nextIndex} - 1$	$\wedge$-E(h1)
11 $indexRead = \overleftarrow{vals}(i).index$	$\wedge$-E-left(4)
12 $indexRead = (\overleftarrow{vals}(1).index -$ len $vals) + 1$	5,11,$\mathbb{N}$
13 $indexRead = (\overleftarrow{nextIndex} - 1 -$ len $vals) + 1$	=-subs(10,12)
14 len $vals \geq 1$	5, len
15 $nextIndex - 1 \geq indexRead$	=subs(1,13,14),$\mathbb{N}$
16 $\overleftarrow{firstIndex} \leq \overleftarrow{nextIndex} -$ len $\overleftarrow{vals}$	$\wedge$-E(h1)
17 $firstIndex \leq indexRead$	=-subs(16,3,13),$\mathbb{N}$
infer $L\text{-}atomic(\overleftarrow{\sigma}, \sigma)$	$\wedge$-I(9,15,17)

Some of the properties that are required to guarantee L-atomicity are encoded directly into the model, for example: when a read takes place all items earlier than that read are removed from the sequence to ensure that an older item cannot be read the next time. The atomicity of the operations ensure that it is not possible for the reader and writer to clash on accessing a particular item, so that coherence is guaranteed.

4.3 Summary

This chapter introduces an abstract model of L-atomicity, which specifies the properties that are required of ACMs in an abstract, but rigorous, manner, and gives details of the proofs that have been discharged to verify that the model exhibits the desired properties. Example rigorous proofs are given.

Verifying properties of asynchronous real-time systems is difficult and

this thesis shows how it is possible to build an understanding of the system in an incremental manner. Starting with an easy to understand abstract model that exhibits the properties that are required of the system, and building and verifying more realistic models to gain an understanding of the behaviour of the implementation. In this way it is possible to gain sufficient confidence that the implementation exhibits the required behaviour. The model given in this chapter is the formal basis of the investigations in the rest of the thesis, which explore the behaviour of Simpson's 4-slot ACM implementation and build confidence in its correctness with respect to the requirements (L-atomicity), in an incremental manner. The formal approach used helps to identify errors and ambiguities in the specification and models, gain a better understanding of the behaviour of the implementation and can help to make assumptions about system and its environment more explicit. This should help to ensure that those assumptions are not overlooked later in the development process. Taken together, the better understanding of the implementation, reduced number of errors and ambiguities and the more explicit assumptions should help to reduce the amount of rework due to flaws that are discovered in the later stages of the development process. The incremental method uses a number of tools to verify properties of increasingly realistic models of the implementation, until sufficient confidence is gained that the implementation has the required properties and exhibits the desired behaviour. Chapter 5 introduces the first of these tools, shows how the ACM implementation can be shown to be a refinement of the model, using Nipkow's retrieve rule, [Nip86, Nip87], and describes how this method can be used to improve understanding of the behaviour of the implementation, to assist in building later models.

Chapter 5

Using Refinement to Verify Properties of Simpson's 4-slot

This thesis shows how it is possible to use a range of tools to verify properties of asynchronous real-time systems and gain an understanding of the behaviour of those systems in an incremental manner. This increased understanding can help to identify and correct errors and ambiguities earlier in the development process and save on the amount of more costly rework due to those flaws. Section 3.3 gave a formal model of Simpson's 4-slot ACM implementation, and Chapter 4 described the first stage of the incremental development process by defining and verifying an abstract model of L-atomicity. This chapter introduces the next stage of the process by showing how it is possible to verify that the formal model of the implementation is a refinement of the model of L-atomicity, subject to an assumption about the atomicity of the operations in the implementation. In order to verify there is a refinement relation between the models it is necessary to assume that some of the operations in the implementation are combined into single atomic actions. While it is recognised that this is not a full correctness proof, since the operations are not combined in this way in actual implementations, discovering the retrieve relation between the models and discharging the proof obligations make it possible to explore properties of those implementations. This exploration gave an increased understanding of behaviour of the implementation which assisted in creating the later more realistic models and verifying properties of those models. An earlier version of the work in this chapter has previously been published in [HP02b].

This chapter is organised as follows: Section 5.1 introduces the notion of refinement. Section 5.2 explains why it is not possible to construct a retrieve function to describe the relation between the model of atomicity and the formal model of the implementation; Section 5.4 describes an outline, and gives details of part, of the retrieve relation between the models. The

proof obligations that are required to verify there is a refinement relation between the models, according to Nipkow's retrieve relation rule [Nip86, Nip87, Jon90], are given in Section 5.5, with a rigorous description of an interesting example proof.

5.1 Refinement

The notion of refinement dates from the stepwise refinement method for constructing programs [Dij71] [Wir71] and work on program correctness [Hoa69] [Hoa72]. For example [Dij71] introduced the notion of developing a sequential program in a stepwise manner, starting with a more abstract notion of what the program is trying to achieve and introducing more detail until the final executable program is completed. This stepwise approach can help with the development of complex programs, where the required algorithm is not known at the outset. The implementation can be completed in an incremental manner as understanding improves. Refinement is the process by which it is possible to verify that the behaviours of the later version of the program are a (possibly complete) sub-set of the behaviours of the earlier version. The refinement calculus provides a logical basis for these methods based on the weakest pre-condition approach to program correctness [Dij76]. It has been extended to the stepwise development of parallel programs and to the refinement of atomicity in parallel programs, e.g. [BvW03], [Bac89].

[Jon90] describes how the notion of refinement can be extended to VDM-SL models of systems. In order to verify that a more detailed model of an implementation is a refinement of a more abstract model it is necessary to verify: first that there is a relation between the the states in an abstract model and the states in a more concrete model; and second demonstrate that if it is possible to execute an operation in the concrete model to move from one state to another, it should be possible to execute an equivalent operation in the abstract model and move between equivalent states in that model. It may be possible to find a retrieve function between the models, or in the more general case, where there is a many to many relation between the states in the models it may be possible to verify the concrete model is a refinement of the abstract one by using a retrieve relation [Nip86, Nip87].

This chapter shows how a concrete model, the formal model of the 4-slot implementation, can be verified to be a refinement of an abstract model, the formal model of L-atomicity, using Nipkow's retrieve relation rule [Nip86, Nip87]. In order to discharge the proof obligations some of the actions of the reader and writer in the implementation need to be combined into single actions, that are *equivalent* to the operations of the abstract model, which are assumed to be executed in a Hoare-atomic manner. It is therefore

recognised that this is not a full correctness proof for the ACM, because these groups of actions are not atomic in actual implementations of the 4-slot. The individual actions can interleave without restriction, and in some (multi-processor or hardware) implementations it is possible for the individual actions of the reader and writer to be executed concurrently. The proofs are, therefore, insufficient to show that the 4-slot is Lamport-atomic when the reader and writer can access the mechanism in an asynchronous manner. A range of tools can be used to relax the assumption about the atomicity of actions of the component processes as is described in Chapters 6 and 7. Moving directly from the abstract model to realistic models of the implementation is a big step, and the exercise of constructing the retrieve relation and completing the refinement proofs is a useful stepping stone in the process. It allows the behaviour of the implementation to be explored in a more abstract manner than would otherwise be possible and helps to identify some of the potential behaviours of the ACM implementation. These lessons are useful when constructing the later, more detailed, models. The next section discusses the different types of relation that can exist between the models in more detail.

5.2 A Retrieve Function?

When a specification is interpreted to produce a design, a representation is chosen which reflects some of the requirements of the implementation. There is a relation between the two representations and it may be possible to use a retrieve function to map states in the concrete model to states in the abstract model (the material in this section is from [Jon90]).

However in some cases, for example where it is necessary to include *history information* in the abstract model that is not present in the implementation, it may not be possible to construct an abstract model of the implementation (or concrete model) so that a retrieve function can be found. Simpson's 4-slot ACM is an example of such an implementation. In the implementation, items are effectively overwritten by the writer recording the pair and slot, to which the latest item has been written, in the control variables. It is only possible for the mechanism to record a maximum of four items, and only one of those four items is available to the reader at any time, although it is possible for the available item to change while the read is in progress. In the abstract model of L-atomicity there may be more than four items in the sequence, all of which are available to the reader. For example if at the start of a read there was a single item available, and the read overlapped with five writes each successive write would add a new item to the sequence. At end read the specification of L-atomicity states that the reader

can read any one of these items, and the abstract model returns a random item from the sequence. These additional items are effectively history information that is not present in the implementation. The implementation is more deterministic because the reader chooses to access a particular slot and returns the item read from that slot.

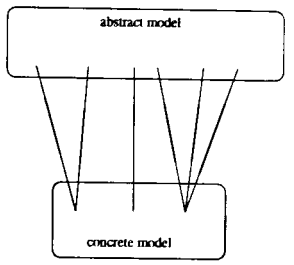

Figure 5.1: A one to many retrieve relation

This requirement for history information to be recorded can result in a *one to many relation* as illustrated in Figure 5.1 between the states in concrete model (or implementation) and the abstract model.

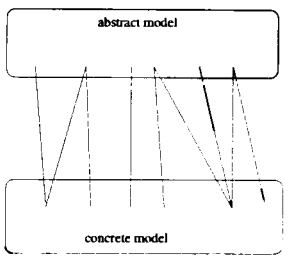

Figure 5.2: A many to many retrieve relation

In the more general case there may be a many to many relation between the specification and the implementation, as illustrated in Figure 5.2. In the case of the relation between the abstract model of L-atomicity and the 4-slot implementation this *implementation bias* occurs because the items that are in the slots in the implementation may be mapped to different items in the sequence in the abstract state, and in fact some of them may not be present in the sequence at all, depending on how the reader and writer interact with the ACM. For example, there may only be a single item available to the reader at the start of a read, because all of the previous items have been overwritten. There will, therefore, be a single item in the sequence in the abstract model, but this item can be in any one of the slots in the concrete

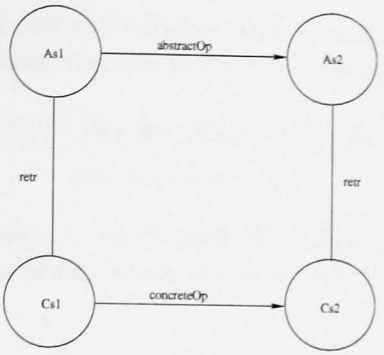

Figure 5.3: Modelling with a Retrieve Relation

model. In addition the implementation will still retain 3 items that were written previously, but these items have been removed from the sequence in the abstract model. There are therefore a number of different states in the implementation that are equivalent, and can be mapped, to the state in the abstract model that contains only a single item. There is, therefore, a many to many relation between the models: it is necessary to construct a retrieve relation between them as illustrated in Figure 5.3, and use Nipkow's rule to discharge the proofs. This requires the following refinement proof obligations to be discharged:

- where the retrieve relation holds between two states, and it is possible to execute an operation in the specification, it is also possible to execute the equivalent operation in the model of the implementation, and furthermore
- the retrieve relation holds between the states in the specification and model that are reached as a result of executing those operations.

These proof obligations are described formally in the next section.

5.3 Formal Definitions of the Proof Obligations

This section gives the formal definitions of the proof obligations that must be discharged to verify that there is a refinement relation between the concrete and abstract models using Nipkow's retrieve rule.

First it is necessary to discharge a *domain proof obligation* for each of the operations, which is the first of the proof obligations described above, as

follows (where R is the retrieve relation, and as and cs are arbitrary states in the abstract model and concrete model respectively):

$$\frac{R(\overleftarrow{as}, \overleftarrow{cs}); pre\_AbstractOp(\overleftarrow{as})}{pre\_concreteOp(\overleftarrow{cs})}$$

Second it is necessary to verify that the following *result proof obligation* holds for each operation, which is the second of the proof obligations described above:

$$\frac{R(\overleftarrow{as}, \overleftarrow{cs}); pre\_abstractOp(\overleftarrow{as}); post\_concreteOp(\overleftarrow{cs}, cs)}{\exists\, as : Abs\_State \cdot R(as, cs) \wedge post\_abstractOp(\overleftarrow{as}, as)}$$

5.4 A Retrieve Relation Between the Formal Models

This section gives an overview of how the retrieve relation between the models of L-atomicity and the implementation has been constructed. The complete retrieve relation, in the PVS logic, is given in Appendix F. The abstract model of L-atomicity given in Section 4 has four operations, *start_write*, *end_write*, *start_read* and *end_read*. The refinement notion requires that the operations in the concrete formal model given in Chapter 3.3.3 are combined into *equivalent* operations to those in the specification. The combination is described as follows:

startWr: the *start_write* operation in the abstract specification adds the a new item to the sequence of values that are available to be read. *startWr* combines *writerChoosesPair*, *writerChoosesSlot*, *write* and *writerIndicatesSlot* operations in the implementation to similarly make the item that has been written available to the reader, in some circumstances, before the write has been completed[1].

startWr (*conc* : *Conc_State*) *c* : *Conc_State*
pre *nwi* = WCP
post *c* = *writerIndicatesSlot*(
write(*writerChoosesSlot*(
writerChoosesPair(*conc*))))

[1] If the reader and writer access the same pair at the same time, and the writer indicates the slot that it has written the latest item to before the reader chooses the slot it is going to read, the reader will acquire, and read the item from, the slot that the writer has accessed.

endWr: this operation completes the write, by executing *writerIndicatesPair* operation in the implementation.

$endWr\ (conc : Conc\_State)\ c : Conc\_State$
pre $nwi = \text{WIP}$
post $c = writerIndicatesPair(conc)$

startRd: this operation combines the *readerChoosesPair*, *readerIndicatesPair* and *readerChoosesSlot* operations in the implementation, and acquires the slot that the reader will access[2].

$startRd\ (conc : Conc\_State)\ c : Conc\_State$
pre $nri = \text{RCP}$
post $c = readerChoosesSlot($
$readerIndicatesPair($
$readerChoosesPair(conc)))$

endRd: executes the *read* operation from the implementation to return the item that has been read.

$endRd\ (conc : Conc\_State)\ c : Conc\_State$
pre $nri = \text{RD}$
post $c = read(conc)$

It is necessary to find a retrieve relation between the abstract and concrete models. The relation between the abstract specification of atomicity and the model of the 4-slot is illustrated in Figure 5.4. The values of the *program counters* for the reader and writer (*nri* and *nwi* respectively) are mapped to the values of the booleans *readerAccess* and *writerAccess* as follows:

1. If $nwi = \text{WCP}$ there is not a write in progress so the writer is not accessing the ACM (*writerAccess* = false), whereas if $nwi = \text{WIP}$ there is a write in progress, so the writer is accessing the ACM (*writerAccess* = true).

2. Similarly for the reader, if $nri = \text{RCP}$ there is not a read in progress so the reader is not accessing the ACM (*readerAccess* = false), whereas

[2]Strictly speaking this is not equivalent to *start_read* in the abstract specification, which does not acquire the item to be read. The abstract specification could be changed so that the reader records the index of the item that it is going to read. This, however, would change the specification so that reader could not read any items that were written by writes that occurred during the read, which does not conform to the notion of atomicity.

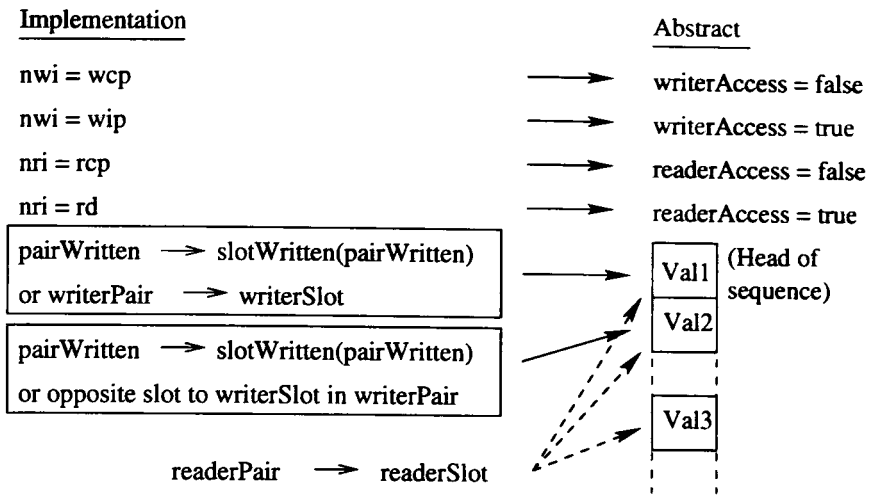

Figure 5.4: The retrieve relation between the concrete and abstract models

if *nri* = RD there is a read in progress, so the reader is accessing the ACM (*readerAccess* = true).

Constructing the retrieve relation requires the addition of an auxiliary boolean variable to the model of the implementation, *writerChangedPairNI*. This is used to record when the writer changes the pair of slots it is accessing, because, until it has completed the first write to the newly chosen pair, and indicated the pair it is accessing, the reader cannot access the same pair as the writer. This means it is not possible for the reader to read the item that has been written during the current write (this item was added to the head of the sequence in the abstract model by the start write model). The boolean is set to true by the *writerChoosesPair* operation, when the writer changes pairs, and is set to false by the *writerIndicatesPair* operation.

With respect to the sequence of values, it is possible to retrieve up to two values from the information recorded by the writer, and it may be possible to retrieve one value from the information recorded by the reader, as described below:

1. If the writer is accessing the ACM, and has written the new item to it, this item will be the head of the sequence in the specification and will be pointed to by the writer local variables *writerPair* and *writerSlot* in the implementation. Otherwise the item at the head of the sequence will be the one written by the last write, and will be pointed to by the control variables in the mechanism, *pairWritten* and *slotWritten(pairWritten)*.

2. If the writer is still accessing the mechanism and it has written the new item, then the item that is second on the sequence of values in the specification will be the one that was written by the last write. This will be pointed to either by the control variables in the mechanism, *pairWritten* and *slotWritten(pairWritten)*, if the writer changed pairs for the current write, otherwise it will be in the same pair, as the writer is currently accessing, but in the the opposite slot.

3. If the reader is accessing the mechanism and has acquired the slot it is going to access, that slot will be pointed to by its local control variables, *readerPair* and *readerSlot*, and it will contain one of the items in the sequence of values in the specification. This may be one of the items that can be accessed from the information recorded by the writer, or a third item, depending on how the read and write actions have interleaved with each other.

The retrieve relation is split into four cases, depending on whether the reader and writer are accessing the ACM, as follows:

$$
\begin{array}{l}
(\neg readerAccess \wedge \neg writerAccess \Rightarrow ....) \wedge \\
\quad (\neg readerAccess \wedge writerAccess \Rightarrow ....) \wedge \\
\quad\quad (readerAccess \wedge \neg writerAccess \Rightarrow ....) \wedge \\
\quad\quad\quad (readerAccess \wedge writerAccess \Rightarrow ....)
\end{array}
$$

The most interesting part of the retrieve relation is where the reader and writer are both accessing the ACM (the last conjunct above, where *readerAccess* and *writerAccess* are both true). The assertions in the relation need to be strong enough that it is possible to verify the reader can return the item in the abstract model that is equivalent to the item returned in the concrete model. The difficulty is that the item returned depends on the recent history of the ACM: i.e. on the precise interleaving of the actions of the reader and writer in the implementation. This part of the retrieve relation is described below.

The fact that the reader and writer are both accessing the ACM, is recorded in the concrete model as:

$$nri = \text{RD} \wedge nwi = \text{WIP}$$

The reader indicated the pair it is accessing during the *startRd* operation, and the writer indicated the slot it is accessing during *startWr*. Therefore the reader local variable *readerPair* and the writer variable *writerSlot* are equal to the relevant control variables, and the writer has added the item

just written to the slot pointed to by its local variables in the implementation and to the head of the sequence in the abstract model:

$$writer.writerSlot = slotWritten(writer.writerPair) \wedge$$
$$reader.readerPair = pairReading \wedge$$
$$slots(writer.writerPair, writer.writerSlot) = vals(1).val$$

The remainder of this part of the retrieve relation depends on the recent history of the ACM, in particular if the writer has changed pairs before starting the current write. The auxiliary variable, *writerChangedPairNI*, records whether the writer has changed pairs or not. The reader chooses the slot it is going to access at *startRd*, so the reader local variables will be pointing to the slot chosen, and the item it has chosen will be one of the items in the sequence of values in the abstract model. This enables the two cases to be defined in the retrieve relation as follows:

1. If the writer has changed pairs the boolean *writerChangedPairsNI* will be true, and the writer local variable *writerPair* will not be equal to the control variable *pairWritten*. In this case the reader and writer cannot be accessing the same pair of slots (as explained above), and the item at the head of the sequence is not available to the reader. The length of the sequence must therefore be strictly greater than 1, and the item written during the last write will be the second in the sequence. The reader will be accessing the pair of slots last accessed by the writer, so the control variables *pairReading* and *pairWritten* will be equal. This gives

$$writerChangedPairNI \Rightarrow$$
$$\quad \mathsf{len}\ vals > 1 \wedge$$
$$\quad pairReading = pairWritten \wedge$$
$$\quad pairWritten \neq writer.writerPair \wedge$$
$$\quad slots(pairWritten, slotWritten(pairWritten)) = vals(2).val \wedge$$
$$\quad\quad (\exists\, i \in \mathsf{inds}\ vals \cdot i > 1 \wedge$$
$$\quad\quad\quad slots(reader.readerPair, reader.readerSlot) = vals(i).val)$$

2. If the writer has not changed pairs for the current write the boolean *writerChangedPairsNI* will be false, and the writer local variable *writerPair* will be equal to the control variable *pairWritten*. In this case there is a possibility that the last read ended and the new read started during the current write, in which case the previous read may have returned the item written by the current write. The sequence of values in the abstract state will then have been shortened to length one and the item written during the previous write (the one pointed to by the control variables in the mechanism) will have been removed

from the sequence in the abstract model, and this item is not included in the retrieve relation. The single item remaining will be the one written during the current write and pointed to by the writer local variables. If a number of writes have occurred during the read, the items written will be in the sequence in the abstract model, and the reader may choose to read any one of these items. The sequence will then be shortened to include the item read and all later items. This gives:

$$
\begin{array}{l}
(\neg writerChangedPairNI \Rightarrow \\
\quad pairWritten = writer.writerPair \wedge \\
\quad\quad (\exists\, i \in \mathsf{inds}\ vals \cdot \\
\quad\quad\quad slots(reader.readerPair, reader.readerSlot) = \\
\quad\quad\quad\quad\quad\quad vals(i).val)) \wedge \\
\quad\quad \mathsf{len}\ vals \geq 1
\end{array}
$$

Combining the above completes this part of the retrieve relation:

$$
\begin{array}{l}
readerAccess \wedge writerAccess \Rightarrow \\
\quad nri = \mathrm{RD} \wedge nwi = \mathrm{WIP} \wedge \\
\quad writer.writerSlot = slotWritten(writer.writerPair) \wedge \\
\quad reader.readerPair = pairReading \wedge \\
\quad slots(writer.writerPair, writer.writerSlot) = vals(1).val \wedge \\
\quad\quad (writerChangedPairNI \Rightarrow \mathsf{len}\ vals > 1 \\
\quad\quad\quad pairReading = pairWritten \wedge \\
\quad\quad\quad pairWritten \neq writer.writerPair \wedge \\
\quad\quad\quad slots(pairWritten, slotWritten(pairWritten)) = vals(2.val) \wedge \\
\quad\quad\quad\quad (\exists\, i \in \mathsf{inds}\ vals \cdot i > 1 \wedge \\
\quad\quad\quad\quad\quad slots(reader.readerPair, reader.readerSlot) = vals(i).val)) \wedge \\
\quad\quad (\neg writerChangedPairNI \Rightarrow \\
\quad\quad\quad pairWritten = writer.writerPair \wedge \\
\quad\quad\quad\quad (\exists\, i \in \mathsf{inds}\ vals \cdot \\
\quad\quad\quad\quad\quad slots(reader.readerPair, reader.readerSlot) = vals(i).val)) \wedge \\
\quad \mathsf{len}\ vals \geq 1
\end{array}
$$

5.5 Discharging the Proof Obligations

This section describes the refinement proof obligations that have been discharged to show that the concrete formal model given here is a refinement of the abstract model.

First the domain proof obligation has been discharged for each of the operations as follows (where R is the retrieve relation, and as and cs are

arbitrary states in the abstract model and concrete model respectively). For example:

$$\frac{R(\overleftarrow{as}, \overleftarrow{cs}); pre\_start\_write(\overleftarrow{as})}{pre\_startWr(\overleftarrow{cs})}$$

Similar proof obligations must be discharged for the other operations.

These proof obligations are relatively trivial to discharge, because, for example in the case of *dom_start_write*, it is simply necessary to show that *writerAccess* = false when *nwi* = WCP (writerChoosesPair). This is the case, because the writer is not accessing the mechanism in both models. The only complication is that each of the proof obligations must be discharged by using a *case distinction*, because the reader may or may not be accessing the mechanism when the write operations are executed and vice versa.

The result proof obligations are more interesting, and the most interesting case, that for *endRd* is shown below[3]. The proof obligation is

$$\frac{R(\overleftarrow{as}, \overleftarrow{cs}); pre\_end\_read(\overleftarrow{as}); post\_endRd(\overleftarrow{cs}, cs)}{\exists\, as : Abs\_State \cdot R(as, cs) \wedge post\_end\_read(\overleftarrow{as}, as)}$$

pre_end_read expands to *readerAccess* = true and the post conditions of the operations are:

$$\texttt{post\_endRd} \triangleq nri = \text{RCP} \wedge v = \overleftarrow{slots}(\overleftarrow{reader.readerPair}, \overleftarrow{reader.readerSlot})$$

$$\texttt{post\_end\_read} \triangleq \neg readerAccess \wedge \exists\, i \cdot \in \text{inds } \overleftarrow{vals} \cdot v = \overleftarrow{vals}(i).val \wedge indexRead = \overleftarrow{vals}(i).index \wedge vals = \overleftarrow{vals}(1, ..., i)$$

A witness value (*as*:*Abs_State*) can now be found to satisfy the conclusion of the proof obligation, which must satisfy *post_end_read* and the retrieve relation. The *end_read* operation shortens the sequence of values to remove items that are older than the one read, sets *indexRead* equal to the index of the item read and also sets *readerAccess* to false, the other component parts of the record are unchanged. This following can therefore be used as the witness value:

$$\text{as} = \text{mk-Abs\_State}(\overleftarrow{vals}(1,...,\text{i}), \overleftarrow{writerAccess}, \text{false}, \overleftarrow{nextIndex}, \overleftarrow{vals}(\text{i}).\text{index}, \overleftarrow{firstIndex})$$

However $\overleftarrow{writerAccess}$ can take two possible values, false and true, and the proof needs to proceed by case distinction as follows (in the outline below the notation "by ???" in the justification of proof lines one and two is used to indicate that the sub-proofs are still to be completed):

[3]The interested reader can download the PVS theory, and the proof scripts from http://homepages.cs.ncl.ac.uk/neil.henderson/fme2002/4slot.tgz.

from $R(\overleftarrow{as}, \overleftarrow{cs}); pre\_end\_read(\overleftarrow{as}); post\_endRd(\overleftarrow{cs}, cs)$
1 from mk-*Abs_State* $(\overleftarrow{vals}\,(1,\ldots,i), \mathsf{false}, \mathsf{false}, ...) : Abs\_State$
infer $\exists\, a : Abs\_State \cdot R(as, cs); post\_end\_read(\overleftarrow{as}, as)$ by ???
2 from mk-*Abs_State* $(\overleftarrow{vals}\,(1,\ldots,i), \mathsf{true}, \mathsf{false}, ...) : Abs\_State$
infer $\exists\, a : Abs\_State \cdot R(as, cs) \wedge post\_end\_read(\overleftarrow{as}, as)$ by ???
infer $\exists\, a : Abs\_State \cdot R(as, cs) \wedge post\_end\_read(\overleftarrow{as}, as)$
case-distinction 1,2

The two sub-proofs are completed in similar ways, and a rigorous proof of sub-proof 2 follows. Three of the conjuncts of the retrieve relation follow immediately by $\Rightarrow$-I-right-vac, since the antecedent of the implication is false in each case as a result of the witness value used. This leaves the fourth conjunct of the retrieve relation to be shown to hold, and it is necessary to show that the witness value satisfies *post_end_read*.

from $R(\overleftarrow{as}, \overleftarrow{cs}); pre\_end\_read(\overleftarrow{as}); post\_endRd(\overleftarrow{cs}, cs)$
1 from mk-*Abs_State* $(\overleftarrow{vals}\,(1,\ldots,i), \mathsf{false}, \mathsf{false}, ...) : Abs\_State$
...
infer $\exists\, a : Abs\_State \cdot R(as, cs); post\_end\_read(\overleftarrow{as}, as)$ $\exists$-I(...)
2 from mk-*Abs_State* $(\overleftarrow{vals}\,(1,\ldots,i), \mathsf{true}, \mathsf{false}, ...) : Abs\_State$
2.1 $\neg\, readerAccess \wedge \neg\, writerAccess \Rightarrow ...$ $\Rightarrow$-I-right-vac(2.h1)
2.2 $readerAccess \wedge \neg\, writerAccess \Rightarrow ...$ $\Rightarrow$-I-right-vac(2.h1)
2.3 $readerAccess \wedge writerAccess \Rightarrow ...$ $\Rightarrow$-I-right-vac(2.h1)
2.4 from $\neg\, readerAccess \wedge writerAccess$
...
infer $\neg\, readerAccess \wedge writerAccess \Rightarrow ...$ by ???
2.5 $R(as, cs)$ $\wedge$-I(2.1,2.2,2.3,2.4)
2.6 from $i \in \mathsf{inds}\ \overleftarrow{vals}$
...
infer $post\_end\_read(\overleftarrow{as}, as)$ by ???
2.7 $R(as, cs) \wedge post\_end\_read(\overleftarrow{as}, as)$ $\wedge$-I(2.5,2.6)
infer $\exists\, a : Abs\_State \cdot R(as, cs) \wedge post\_end\_read(\overleftarrow{as}, as)$
$\exists$-I(2.h1,2.7)
infer $\exists\, a : Abs\_State \cdot R(as, cs) \wedge post\_end\_read(\overleftarrow{as}, as)$
case-distinction 1,2

Considering the completion of sub-proof 2.6 first, $R(\overleftarrow{as}, \overleftarrow{cs})$ gives the following:

$$
\begin{array}{l}
(writerChangedPairNI \Rightarrow \mathsf{len}\ vals > 1 \\
\qquad \ldots \\
\qquad (\exists\, i \in \mathsf{inds}\ vals \cdot i > 1 \wedge \\
\qquad\quad slots(reader.readerPair, reader.readerSlot) = vals(\boldsymbol{i}).\boldsymbol{val})) \wedge \\
(\neg writerChangedPairNI \Rightarrow \\
\qquad \ldots \\
\qquad (\exists\, i \in \mathsf{inds}\ vals \cdot \\
\qquad\quad slots(reader.readerPair, reader.readerSlot) = vals(i).\boldsymbol{val}))\ldots
\end{array}
$$

post_endRd states that the reader, in the implementation, returns the item from $\overleftarrow{slots(reader.readerPair, reader.readerSlot)}$ which allows the sub-proof to be completed. Strictly the sub-proof should be discharged by case distinction on the value of *writerChangedPairNI*, but the two cases are almost identical and will be combined for the purposes of this rigorous proof:

from $R(\overleftarrow{as}, \overleftarrow{cs}); pre\_end\_read(\overleftarrow{as}); post\_endRd(\overleftarrow{cs}, cs)$

1 from mk-Abs_State $(\overleftarrow{vals}\,(1, \ldots, i), \mathsf{false}, \mathsf{false}, \ldots) : Abs\_State$

...

infer $\exists\, a : Abs\_State \cdot R(as, cs)$: $post\_end\_read(\overleftarrow{as}, as)$ $\exists$-I(...)

2 from mk-Abs_State $(\overleftarrow{vals}\,(1, \ldots, i), \mathsf{true}, \mathsf{false}, \ldots) : Abs\_State$

...

2.6 from $i \in \mathsf{inds}\ \overleftarrow{vals}$

2.6.1 $v = \overleftarrow{slots}(\overleftarrow{reader.readerPair}, \overleftarrow{reader.readerSlot})$ post_endRd-defn

2.6.2 $v = \overleftarrow{vals}(i).val$ =-subst(h1,2.6.1)

2.6.3 $indexRead = \overleftarrow{vals}(i).index$ 2.h1

2.6.4 $\neg\, readerAccess$ 2.h1

2.6.5 $vals = \overleftarrow{vals}(1, \ldots, i)$ 2.h1

2.6.6 $\neg\, readerAccess \wedge v = vals(i).val...$ $\wedge$-I(2.6.2,2.6.3,2.6.4,2.6.5)

infer $post\_end\_read(\overleftarrow{as}, as)$ $\exists$-I(2.6.h1,2.6.5)

2.7 $R(as, cs) \wedge post\_end\_read(\overleftarrow{as}, as)$ $\wedge$-I(2.5,2.6)

infer $\exists\, a : Abs\_State \cdot R(as, cs) \wedge post\_end\_read(\overleftarrow{as}, as)$ $\exists$-I(2.h1,2.7)

infer $\exists\, a : Abs\_State \cdot R(as, cs) \wedge post\_end\_read(\overleftarrow{as}, as)$ case-distinction 1,2

The remaining conjunct of the retrieve relation that must be shown to hold (sub-proof 2.4) is:

$$
\begin{aligned}
&\neg readerAccess \wedge writerAccess \Rightarrow \\
&\quad nri = \text{RCP} \wedge nwi = \text{WIP} \wedge \\
&\quad writer.writerSlot = slotWritten(writer.writerPair) \wedge \\
&\quad reader.readerPair = pairReading \wedge \\
&\quad slots(writer.writerPair, writer.writerSlot) = vals(1).val \wedge \\
&\qquad (writerChangedPairNI \Rightarrow \mathsf{len}\ vals > 1 \wedge \\
&\qquad\ pairReading = pairWritten \wedge \\
&\qquad\ pairWritten \neq writer.writerPair \wedge \\
&\qquad\ slots(pairWritten, slotWritten(pairWritten)) = \\
&\qquad\qquad vals(2).val) \wedge \\
&\qquad (\neg writerChangedPairNI \Rightarrow \\
&\qquad\ pairWritten = writer.writerPair) \wedge
\end{aligned}
$$

$$\mathsf{len}\ vals \geq 1$$

Most of the above follows directly from h1 ($R(\overleftarrow{as}, \overleftarrow{cs})$) since:

1. The values of the control variables and writer and reader local variables are not changed by *post_endRd*, therefore the equality, or otherwise, of the control variables and the local variables is unchanged in the conclusion of the proof. Similarly the writer program counter remains unchanged, since the writer has not executed an operation (nwi = WCP).

2. No new items are written to the ACM, and the sequence in the abstract model always contains at least one item, therefore $slots(writer.writerPair, writer.writerSlot) = vals(1).val$ will hold in the conclusion.

It only remains to prove that the sequence will be of the correct length, depending on the value of the auxiliary variable *writerChangedPairNI*. (If the sequence in the abstract model is of length greater than one after end read the relation $slots(pairWritten, slotWritten(pairWritten)) = vals(2).val$ will automatically follow and complete the proof). $R(\overleftarrow{as}, \overleftarrow{cs})$ states that the sequence in the abstract model has at least one item in it before end read is executed, therefore the case where *writerChangedPairNI* is false follows trivially, since a subsequence of a non empty sequence must contain at least one item. The case where the auxiliary variable is true is proved by case-distinction on the value of i. If $i = 1$ the proof follows by contradiction, since $R(\overleftarrow{as}, \overleftarrow{cs})$ states:

$$\exists\, i \in \mathsf{inds}\ vals \cdot i > 1 \wedge slots(reader.readerPair, reader.readerSlot) = vals(i).val$$

and *post_endRd* states that the reader in the implementation returns the value from $slots(reader.readerPair, reader.readerSlot)$. The value of i cannot, therefore, be equal to 1. If $i > 1$ the proof follows from the definitions of the retrieve relation and *post_endRd*[4].

[4]Strictly speaking sub-proof 2.4 verifies that the consequent of this conjunct of the retrieve relation holds (using $\wedge$-I), and this sub-proof can then be used with the $\Rightarrow$-I rule to establish $\neg\, readerAccess \wedge writerAccess \Rightarrow$ .... This would, however, add an additional line to the proof and the remaining lines would need to be renumbered. Therefore, in order to keep the line numbers consistent with the outlines on the preceding pages, this extra step has been omitted.

from $R(\overleftarrow{as}, \overleftarrow{cs}); pre\_end\_read(\overleftarrow{as}); post\_endRd(\overleftarrow{cs}, cs)$

1 from mk-Abs_State $(\overleftarrow{vals}\,(1, \ldots, i), \text{false}, \text{false}, \ldots) : Abs\_State$

...

infer $\exists\, a : Abs\_State \cdot R(as, cs); post\_end\_read(\overleftarrow{as}, as)$ $\exists$-I(...)

2 from mk-Abs_State $(\overleftarrow{vals}\,(1, \ldots, i), \text{true}, \text{false}, \ldots) : Abs\_State$

...

2.4 from $\neg\, readerAccess \wedge writerAccess$

2.4.1 $nri = \text{RCP}$ post_endRd-defn

2.4.2 from $writerChangedPairNI$

2.4.2.1 from $i = 1$

infer len $vals > 1$

contradiction(h1,2.h1,post_endRd-defn)

2.4.2.2 from $i > 1$

infer len $vals > 1$ (h1,2.h1,post_endRd-defn)

infer len $vals > 1$ case-distinction(2.4.2.1, 2.4.2.2)

2.4.3 from $\neg\, writerChangedPairNI$

infer len $vals \geq 1$ 2.h1, len

2.4.4 len $vals \geq 1$ case-distinction(2.4.2.2.4.3)

...

infer $\neg\, readerAccess \wedge writerAccess \Rightarrow$ ...

$\Rightarrow$-I(2.4.h1,2.4.4,...)

2.5 $R(as, cs)$ $\wedge$-I(2.1,2.2,2.3,2.4)

...

2.7 $R(as, cs) \wedge post\_end\_read(\overleftarrow{as}, as)$ $\wedge$-I(2.5,2.6)

infer $\exists\, a : Abs\_State \cdot R(as, cs) \wedge post\_end\_read(\overleftarrow{as}, as)$

$\exists$-I(2.h1,2.7)

infer $\exists\, a : Abs\_State \cdot R(as, cs) \wedge post\_end\_read(\overleftarrow{as}, as)$

case-distinction 1,2

5.6 Summary

This chapter introduces the first of the tools, that has been used in the incremental development process described in this thesis to verify properties of the 4-slot, which can be used to gain an improved understanding of the behaviour of the implementation to help to reduce errors and ambiguities in the specification earlier in the development process. It describes method of demonstrating that the 4-slot implementation is a refinement of the abstract model of L-atomicity from Chapter 3. Discovering the retrieve relation and

discharging the proof obligations gave increased confidence in its correctness, and helped in improving our understanding of the behaviour of the implementation by identifying the following behaviour:

1. If the writer has changed pairs and has not indicated that it has changed when a read starts, the slot the reader will access is effectively chosen at start read, when the reader executes the *readerChoosesPair* operation. The reader will access the pair of slots the writer previously accessed, and any new values will be written to the pair the writer is now accessing. The reader will continue to access the pair the writer was previously accessing, until after the end of the current write, and the values of the control variables *pairReading* and *pairWritten* will remain equal until then.

2. There are two points within the writer algorithm when the item that is being written can be released and made available to the reader. If the reader chooses to access a different pair to the writer, the item was effectively released by the *writerIndicatesPair* operation at the end of the last write to the pair the reader has chosen. If the reader accesses the same pair as the writer, the point that the item is released is dependent on the ordering of the *readerChoosesSlot* and *writerIndicatesSlot* operations.

3. There are effectively a maximum of three different items that the reader can return as a result of a read. The exact interleaving of the actions of the reader and writer, and the recent history of the interleaving of those actions, determine which of the slots the reader will access during a read and which of these three items it will return.

Unfortunately the proofs rely on an unrealistic assumption about the atomicity of the actions of the reader and writer of the ACM. In order to discharge the proof obligations some of the actions of the reader and writer in the implementation need to be combined into single actions, that are equivalent to the operations of the abstract model, which are assumed to be executed in a Hoare-atomic manner. It is therefore recognised that this is not a full correctness proof for the ACM, because these groups of actions are not atomic in actual implementations of the 4-slot: the proofs are insufficient to show that the 4-slot implementation is L-atomic when the reader and writer can access the mechanism in a totally asynchronous manner. Unfortunately it is not possible to relax the atomicity assumptions and use refinement to verify properties of the implementation for two reasons. First, each of the individual reader and writer actions either accesses a control variable in the mechanism, or one of the slots. It is possible for an unbounded number of

reader actions to occur between any two writer actions. Similarly it is possible for an unbounded number of writer actions to occur between any two reader actions. It is therefore possible for any of the writer actions to interfere with the operation of the reader, for example. Discovering a relation between the models would, therefore, be very difficult. Second, it became apparent, when discovering the retrieve relation and discharging the proof obligations that the writer may effectively release the item it has written at different points in its algorithm. If the reader accesses the same pair as the writer the item is available to it as soon as *writerIndicatesSlot* has been executed, however, if the reader accesses the opposite pair to the writer the item it is going to acquire was released by the last *writerIndicatesPair* operation. Despite this shortcoming, the effort was considered worthwhile, because of the increased understanding of the behaviour of the system, and the increased confidence in the correctness of the system, that was gained. Recent work, [BvW03], has extended action systems by adding a guarantee condition to each process, but it may not be possible to find suitable guarantee conditions for the processes, to use action refinement, to verify the implementation is a refinement of the abstract model.

The exhaustive proof method used to verify the abstract model of atomicity could be used to verify the implementation is Lamport atomic, when the individual actions of the reader and writer are themselves atomic. This, however, would require an exploration of the entire state space of the 4-slot. This state space is not simply the cross product of the number of read and write operations, because, for example, the behaviour of the mechanism can change if a read occurs when the writer has changed pairs but has not indicated it has changed. It would be a non trivial task to ensure that the entire state space is explored correctly, and verification proofs would need to be discharged for each of the states in the entire state space. Therefore this is not considered to be a practical solution, and it was necessary to explore other proof methods to relax the assumption about the atomicity of the actions of the reader and writer and verify that the 4-slot implementation is L-atomic. Chapter 6 describes such a method, using an assertional rely-guarantee proof method for interleaved shared variable concurrency and the lessons about the behaviour of the implementation described above assisted in devising assertions that are required for this method. This method has the advantage that it may be possible to use the rely-guarantee conditions of the ACM, with a model of its behaviour, to verify properties of larger systems, where the 4-slot is itself used as a component.

Chapter 6

Applying a Compositional Proof Method

This thesis describes an incremental approach to system development. Starting with an abstract model of the required properties of the system it shows how an understanding of its behaviour can be gained over time, by verifying properties of increasingly realistic models of the implementation. Chapter 4 described an abstract model of L-atomicity that has been used as the basis of the incremental approach. Chapter 5 then described a refinement method that has been used to verify that the 4-slot implementation is a refinement of an abstract specification of atomicity. The proof that a refinement relation exists between the models relies on an unrealistic assumption about the (Hoare) atomicity of actions of the reader and writer to the ACM: some actions of the reader (and writer) in the implementation are grouped into single atomic actions in order to discharge the proof obligations. In implementations the individual actions of the reader and writer can interleave in an unconstrained manner, and in fully asynchronous implementations it is possible for the actions to be executed concurrently. Verifying the refinement relation helped to build an understanding of the behaviour of the implementation, however a means to relax the atomicity assumptions is required.

This chapter describes the next stage in the process: how a rely-guarantee method for interleaved shared variable concurrency can be used to verify properties of systems when the individual actions of the individual processes can interleave in an unconstrained manner. The method also overcomes the a second deficiency of the refinement approach, because it allows the verification of systems where one of the components can execute an unbounded number of actions in between any two actions of another component. This makes it possible to verify properties of some actual implementations, for example where the system is implemented on a single processor. First a proof of L-atomicity is given for Simpson's 4-slot (this work has previously

been published in [Hen03]). Then an incorrectness proof is briefly described to demonstrate how the method can be used to identify errors in proposed implementations. This shows that a 3-slot ACM implementation may allow the reader and writer to access the same slot at the same time, so the reader may return invalid data as the result of a read. The chapter is organised as follows. First Section 6.1 introduces the rely-guarantee method. Section 6.2 introduces the compositional method used to verify properties of the 4-slot and describes the proof obligations that need to be discharged. Section 6.3 describes how the method has been used to verify that the 4-slot implementation is L-atomic; and Section 6.4 briefly describes how the method has been used to verify the incorrect operation of a 3-slot ACM implementation. All of the proofs in this chapter have been discharged using the PVS theorem prover. The model of the 4-slot implementation is that described in Section 3.3.3, with additional auxiliary variables that record history information about the behaviour of the implementation.

6.1 Rely-Guarantee

The rely-guarantee proof method [Jon81, Jon83] was developed to give a precise means of specifying interference between parallel programs. Formal languages, such as VDM-SL[ISO96] can be used to give specifications of programs a precise meaning, so that properties of those programs can be verified in a rigorous manner. Such languages, however, assume that operations are executed atomically: in VDM-SL pre- and post-conditions are given for operations, that specify the state of the program before and after the operation is executed. It is assumed that nothing will occur while the operation is being executed to interfere with the result and make the post-condition invalid. In implementations where components are implemented in parallel it is possible for the components to interfere with each other, for example the writer of a shared variable may be able to overwrite the value stored while the variable is being read.

The rely-guarantee method allows the specification of additional properties of interfering programs:

A Rely Condition: that specifies the maximum amount of interference that a process or program can tolerate from its environment.

A Guarantee Condition: that specifies what guarantees a process or program provides about its behaviour, for example the maximum amount of interference that it will generate.

For example in the 4-slot implementation the reader of the mechanism relies on the fact that the writer, once it has chosen a slot, will access that

chosen slot. In this way the reader can choose to read a different slot, safe in the knowledge that the writer will not *interfere* with it, and it can read a coherent value from its chosen slot. The complete implementation guarantees that the reader will read fresh, coherent, data i.e. that it is L-atomic.

An exhaustive proof method could be used to verify that the 4-slot implementation is L-atomic, when the actions of the reader and writer can interleave in an unconstrained manner. This would, however, require an exhaustive exploration of the entire state space of the implementation. Verification proofs for each of the states in this entire state space would then need to be discharged. It would be a non trivial task to ensure that the state space was explored correctly, particularly since the behaviour of the mechanism changes in certain circumstances, for example when the writer changes pairs at the start of a write[1]. For this reason an exhaustive method is not considered to be a practical solution, and it was necessary to explore other methods to verify that the 4-slot is L-atomic. This chapter describes such a method, which can be used to verify properties of systems where the components communicate via shared variables.

6.2 A Proof Method for Shared Variable Concurrency

This section describes a rely-guarantee method, from [dR+01], that can be used to verify properties of systems where the components communicate using shared variables, and the actions of those components can interleave in an unconstrained manner. The method assumes that the individual actions of those components are atomic, and therefore they cannot occur concurrently.

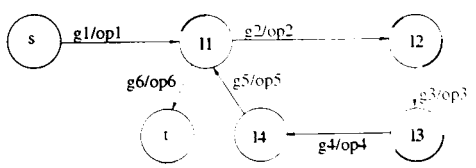

Figure 6.1: An Example Assertion Network

The method is based on the inductive assertion method from [Flo67], generalised to include the additional rules required for rely-guarantee formulae. *Assertion networks* are produced for the individual processes in the system, and the additional rules are used to verify: first that those processes

[1]It is not then possible for the reader to access the same pair of slots as the writer until after the writer has completed an entire write, and indicated it has changed pairs.

meet their individual guarantee conditions on their actions provided their rely conditions are met; and second that the complete system, which is a composed from those individual processes, meets its guarantee conditions on its actions, provided their rely conditions are met. An example assertion network is shown in Figure 6.1 which is constructed as follows:

1. Each network is based on a state transition diagram that describes the operation of the component. The diagram is a quadruple (L, T, s, t), where L is a finite set of locations, T is a finite set of labelled transitions between those locations, there is a unique start location s, and a unique final location t ($s \neq t$ and $\{s, t\} \in L$).

2. The labels on the transitions consist of a guard and an operation. The guard is a predicate over the state of the system and the transition is enabled whenever the guard evaluates to true. The transition can be taken when the component is in the start location of the transition and the guard evaluates to true, and the associated operation is then executed.

3. Each of the locations has an assertion associated with it that must hold at all times when the component process is in that location. These assertions must satisfy the guarantee conditions for the actions of each of the components as described below, and in general encode information about the values of the shared variables and history information about the system. For example, in order to prove mutually exclusive access to a shared resource, the assertions would encode details about the values of the shared variables that control access to the resource.

The following proof obligations must be discharged for each transition (action) in the assertion networks of the components:

- That if the rely condition holds and the assertion in the start state holds, that the assertion in the target location also holds if the operation associated with the transition is executed.

- In addition, because the system uses shared variables for communication between its components it is necessary to verify that the operations associated with the transitions in a network do not interfere with the assertions in any of the locations of the assertion networks for other components of the system (the Aczel semantics [Acz83] described in [dR+01]). This is because, in general, the assertions of the components will include statements about the values of the shared variables. Since the operations of the components can interleave in an unconstrained manner, it must be shown that, if one of the components relies on a

shared variable taking a particular value, that value cannot be changed by a transition taken by one of the other components.

- When the assertion in the start location of the transition holds and the transition is enabled (its guard is true): that the state of the system meets the guarantee condition for the action is satisfied ($\sigma \models guar$ and $op(\sigma) \models guar$, where σ is the current state of the component, including any relevant history information, in the model).

A parallel composition rule is then used to show that the system meets the guarantee conditions on its actions. The system is composed of n components, $C_1 ... C_n$. It is necessary to discharge the following proof obligations for every transition in the assertion networks the components:

1. Since every transition of component C_i, and every transition of the environment of the system is seen as an environment transition by every other component C_j, $i \neq j$, it is necessary to show that the rely condition on the actions of the component C_j is satisfied by the rely condition on the actions of the composed system on the environment and the guarantee conditions on the actions of all of the other components.

2. Every transition of the components, $C_1 ... C_n$, is a transition of the composed system so it is necessary to show that the guarantee conditions on the actions of the components satisfy the guarantee condition of the actions of the composed system i.e. $guar_{C_1} \vee ..... \vee guar_{C_n} \models guar$.

The advantages of this method are:

1. It is not necessary to identify the complete state space of the composed system. This is difficult for relatively small systems and may not be tractable for larger systems.

2. It is only necessary to discharge proof obligations for each of the transitions in the assertion networks of the components, rather than proofs for each of the transitions in the full state space of the composed system.

The disadvantage is that the proofs for the transitions in the assertion networks may be more complex than the proofs for the transitions in the composed system, because of the need to prove non-interference between the components, but this disadvantage is outweighed by the above advantages. The identification of the state space of the composed system would be error prone, and it is anticipated that the number of proofs required for the 4-slot, for example, would be more than double the number required by this method.

6.3 Verifying L-Atomicity of the 4-slot Implementation

This section shows how the rely-guarantee method can be used to verify that Simpson's 4-slot ACM is L-atomic. The assertion networks for the reader and writer are described in Section 6.3.1. The verification proof has been split in to two parts: first a proof that the implementation guarantees to transmit coherent data between the reader and writer is described in Section 6.3.3; then a proof that it communicates globally fresh data is briefly introduced in Section 6.3.4. These properties together are sufficient for L-atomicity. All of the proofs are based on the formal model of the 4-slot which is given in Section 3.3.3.

6.3.1 Assertion Networks for the Component Processes

The assertion networks for the reader and writer processes of the 4-slot are shown Figure 6.2 and Figure 6.3 respectively.

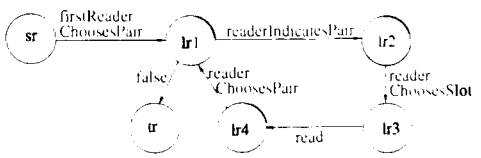

Figure 6.2: Assertion Network for the Reader

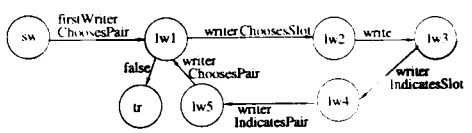

Figure 6.3: Assertion Network for the Writer

The assertion networks are briefly described as follows:

1. The networks both contain a transition labelled *false*, which leads to their respective termination locations. This transition is included only to explicitly indicate that the reader and writer algorithms do not terminate once they have started (inclusion of this transition follows the style used in [dR+01]).

2. There are no guards on the transitions in the networks, because the guards are all *true*, which means that the outgoing transition from a location can be taken at any time when the process is in that location[2].

3. Each of the transitions is labelled with the operation that is executed when it is taken.

The assertions that are associated with the locations in the assertion networks of the reader and writer and the verification proofs that show that the 4-slot is L-atomic are described in Sections 6.3.3 (the coherence proof) and 6.3.4 (the freshness proof). First Section 6.3.2 gives formal descriptions of the proof obligations from Section 6.2.

6.3.2 Formal Descriptions of the Proof Obligations

The proof obligations from Section 6.2 are described below, using a VDM-SL like notation as with the models in the previous chapters, and the variables (or state) are hooked to indicate the value before an operation is executed, where appropriate, as before.

- In order to discharge the first proof obligation it is necessary to show for each operation in the reader and writer assertion networks:

$$\frac{pre\_Op(\overleftarrow{\sigma}); startState\_Assertion(\overleftarrow{\sigma}); post\_Op(\overleftarrow{\sigma}, \sigma)}{targetState\_Assertion(\sigma)}$$

- The second proof obligation is to show non interference between the reader operations and assertions in the writer network and vice versa. This involves showing, for each write operation:

  Similarly for each read operation:

  In the model these two proof obligations are combined into a single *consistency proof* for each operation.

- In addition a well-formedness proof (called a TCC by PVS) needs to be discharged for each of the operations. A witness value must be provided for each operation to show that there exists a state of the ACM such that the operation can be executed.

[2]There is a pre-condition in each of the operations in the model, however this relates to the value of a *program counter*, which is not present in the implementation. This program counter is simply used to record the next operation that can be executed by the component process, and is analogous to the process being in the location where that operation can be executed in the assertion network.

$$\frac{\begin{array}{c} firstReaderChoosesPair\_Assertion(\overleftarrow{\sigma}); \\ readerChoosesPair\_Assertion(\overleftarrow{\sigma}) : readerIndicatesPair\_Assertion(\overleftarrow{\sigma}) \\ readerChoosesSlot\_Assertion(\overleftarrow{\sigma}); read\_Assertion(\overleftarrow{\sigma}); \\ post\_writer\_Op(\overleftarrow{\sigma}, \sigma) \end{array}}{\begin{array}{c} firstReaderChoosesPair\_Assertion(\sigma) \wedge \\ readerChoosesPair\_Assertion(\sigma) \wedge readerIndicatesPair\_Assertion(\sigma) \wedge \\ readerChoosesSlot\_Assertion(\sigma) \wedge read\_Assertion(\sigma) \end{array}}$$

$$\frac{\begin{array}{c} firstWriterChoosesPair\_Assertion(\overleftarrow{\sigma}); \\ writerChoosesPair\_Assertion(\overleftarrow{\sigma}); writerChoosesSlot\_Assertion(\overleftarrow{\sigma}); \\ write\_Assertion(\overleftarrow{\sigma}); writerIndicatesSlot\_Assertion(\overleftarrow{\sigma}); \\ writerIndicatesPair\_Assertion(\overleftarrow{\sigma}); post\_reader\_Op(\overleftarrow{\sigma}, \sigma) \end{array}}{\begin{array}{c} firstWriterChoosesPair\_Assertion(\sigma) \wedge \\ writerChoosesPair\_Assertion(\sigma) \wedge \\ writerChoosesSlot\_Assertion(\sigma) \wedge write\_Assertion(\sigma) \\ writerIndicatesSlot\_Assertion(\sigma) \wedge writerIndicatesPair\_Assertion(\sigma) \end{array}}$$

- In order to show that the guarantee condition of the components holds there are two proof obligations for each operation:

  1. To show that the guarantee condition holds before the operation is executed. For example (for the writer):

$$\frac{\begin{array}{c} pre\_writer\_Op(\overleftarrow{\sigma}); startState\_Assertion(\overleftarrow{\sigma}); \\ firstReaderChoosesPair\_Assertion(\overleftarrow{\sigma}); \\ readerChoosesPair\_Assertion(\overleftarrow{\sigma}); \\ readerIndicatesPair\_Assertion(\overleftarrow{\sigma}); \\ readerChoosesSlot\_Assertion(\overleftarrow{\sigma}); read\_Assertion(\overleftarrow{\sigma}) \end{array}}{guar(\overleftarrow{\sigma})}$$

  2. To show that the guarantee condition will still hold in the target location of the transition after the associated operation is executed, which (again for the writer) is given as:

- The rely condition of the composed system, the ACM, on its environment is that the underlying hardware will be fault free. For example that items written to the buffers, and values in the control variables will remain until they are overwritten, and that reads to the buffers and control variables will return the values stored.

$$\frac{\begin{array}{c} pre\_writer\_Op(\overleftarrow{\sigma});\ startState\_Assertion(\overleftarrow{\sigma}): \\ firstReaderChoosesPair\_Assertion(\overleftarrow{\sigma}); \\ readerChoosesPair\_Assertion(\overleftarrow{\sigma}): \\ readerIndicatesPair\_Assertion(\overleftarrow{\sigma}); \\ readerChoosesSlot\_Assertion(\overleftarrow{\sigma});\ read\_Assertion(\overleftarrow{\sigma}): \\ post\_writer\_Op(\overleftarrow{\sigma}, \sigma) \end{array}}{guar(\sigma)}$$

- The property that has been shown to hold for each of the locations in the individual networks is the same as the property that is required of the composed system, and, provided the rely condition on the environment is met, the composition proof follows immediately. Therefore it is not proved separately.

6.3.3 The Coherence Proof

The coherence proof shows that the reader and writer to the ACM cannot access the same slot in the mechanism at the same time, and that the implementation therefore only communicates coherent data. Assertions give the relationship between the local copies of the control variables in the reader and writer and the values of those control variables in the mechanism itself. The most interesting assertion is that for location *lr*3 in the reader assertion network, when the reader is about to execute the *read* operation. A description of this assertion follows.

The assertion makes use two auxiliary variables. First, *wisOccurred* which is set to true by the *writerIndicatesSlot* operation and to false by the *writerIndicatesPair* operation. It is therefore true whenever the writer has already indicated the slot it is accessing during the current write. Second, *rcsSinceWis*, which is set to true by the *readerChoosesSlot* operation and false by the *writerIndicatesSlot* operation. It is therefore true whenever *readerChoosesSlot* has been executed since *writerIndicatesSlot*.

When the reader is about to read the data from a buffer in the ACM it has previously indicated the pair it is accessing, during the *readerIndicatesPair* operation, and the local variable *readerPair* is therefore equal to the control variable *pairReading*. The reader has also chosen the slot it is going to read from, when it executed the *readerChoosesSlot* operation. However it is not always possible to relate the value of the slot chosen directly to the control variables in the mechanism itself, because the writer has write access to the *slotWritten* array. If the reader and writer are accessing different pairs they are by definition accessing different slots. However if the reader and writer are accessing the same pair two different *cases* for this value need to be

considered in the assertion:

1. The writer has not got as far as indicating the slot it is writing to in the current write (the auxiliary boolean variable *wisOccurred* is false). In this case the reader's copy of the control variable will record the same value as the control variable itself.

2. The writer has indicated the slot it is writing to (*wisOccurred* is true) when *rcsSinceWis*, is used to reason about whether:

   - The reader chose the slot to read from after the writer had indicated the new slot it had written to: the readers local copy of the value will be the same as the appropriate element of the control variable *slotWritten* relating to the pair the reader is accessing, and the auxiliary variable *rcsSinceWis* will be true.
   - The reader chose its slot before the writer indicated the new slot it had written to, in which case the reader will access the opposite slot in the pair to the writer, and *rcsSinceWis* will be false.

   Once the reader is reading from a slot, it has previously indicated the pair it is reading from (at *readerIndicatesPair*), so the writer will change pairs at the next start write, and cannot access the same pair in the next write.

The assertion is given as:

$$
\begin{array}{l}
read\_Assertion \triangleq nri = \mathrm{RD} \wedge \Rightarrow \\
\quad pairReading = reader.readerPair \wedge \\
\quad\quad (reader.readerPair = writer.writerPair \Rightarrow \\
\quad\quad\quad (\neg wisOccurred \Rightarrow \\
\quad\quad\quad\quad\quad reader.readerSlot = slotWritten(reader.readerPair)) \wedge \\
\quad\quad\quad (wisOcurred \Rightarrow (rcsSinceWis \Rightarrow reader.readerSlot = \\
\quad\quad\quad\quad\quad\quad\quad\quad slotWritten(reader.readerPair)) \wedge \\
\quad\quad\quad\quad\quad\quad (\neg rcsSinceWis \Rightarrow reader.readerSlot \neq \\
\quad\quad\quad\quad\quad\quad\quad\quad slotWritten(reader.readerPair))))
\end{array}
$$

It is not necessary to make any assertions in the coherence proof for the reader network locations $sr.lr1$ and $lr4$ (when the reader is about to execute *firstReaderChoosesPair*, *readerIndicatesPair*, and *readerChoosesPair* respectively) and the writer network location $lw5$ (when the writer is about to execute *writerChoosesPair*).

The Coherence Proof Obligations

This section describes how the proof obligations described in Section 6.3.2 are discharged in order to verify that the 4-slot implementation preserves

coherence of data. This requires assertions to be discovered that are sufficient to meet the required property. Rather than describe the complete model and all of the proofs and overview is given of the proofs for one of the transitions. The remaining proof obligations are discharged in a similar manner and the complete model is given in Appendix G[3]. The most interesting consistency proof obligation relates to the *readerChoosesSlot* operation, which verifies that the above *read_Assertion* holds after the operation is executed. The proof obligation is:

$$\frac{\begin{array}{c} pre\text{-}readerChoosesSlot(\overleftarrow{\sigma});\ readerChoosesSlot\_Assertion(\overleftarrow{\sigma});\\ firstWriterChoosesPair\_Assertion(\overleftarrow{\sigma});\\ writerChoosesPair\_Assertion(\overleftarrow{\sigma});\ writerChoosesSlot\_Assertion(\overleftarrow{\sigma});\\ write\_Assertion(\overleftarrow{\sigma});\ writerIndicatesSlot\_Assertion(\overleftarrow{\sigma});\\ writerIndicatesPair\_Assertion(\overleftarrow{\sigma});\ post\_readerChoosesSlot(\overleftarrow{\sigma},\sigma) \end{array}}{\begin{array}{c} read\_Assertion(\sigma) \wedge firstWriterChoosesPair\_Assertion(\sigma) \wedge\\ writerChoosesPair\_Assertion(\sigma) \wedge\\ writerChoosesSlot\_Assertion(\sigma) \wedge write\_Assertion(\sigma) \wedge\\ writerIndicatesSlot\_Assertion(\sigma) \wedge writerIndicatesPair\_Assertion(\sigma); \end{array}}$$

The *readerChoosesSlot* assertion is given below, for convenience the *readerChoosesSlot* operation is also repeated, and an outline proof follows:

readerChoosesSlot_Assertion $\triangleq$
$nri = \text{RCS} \wedge pairReading = reader.readerPair$

readerChoosesSlot ()
ext wr nri : *nextReadInstruction*
wr *reader.readerSlot* : *SlotIndex*
rd *slotWritten* : $PairIndex \xrightarrow{m} SlotIndex$
pre $nri = \text{RCS}$
post $nri = \text{RD} \wedge reader.readerSlot = \overleftarrow{slotWritten}(\overleftarrow{reader.readerPair})$;

[3]The interested reader can download the PVS theory, and proof scripts, from http://homepages.cs.ncl.ac.uk/neil.henderson/fme2003/coherent.tgz.

from $pre\text{-}readerChoosesSlot(\overleftarrow{\sigma})$; $readerChoosesSlot\_Assertion(\overleftarrow{\sigma})$;
$writerChoosesPair\_Assertion(\overleftarrow{\sigma})$;
$writerChoosesSlot\_Assertion(\overleftarrow{\sigma})$
$write\_Assertion(\overleftarrow{\sigma})$; $writerIndicatesSlot\_Assertion(\overleftarrow{\sigma})$;
$writerIndicatesPair\_Assertion(\overleftarrow{\sigma})$; $readerChoosesSlot(\overleftarrow{\sigma}, \sigma)$
1 nri = RD post_readerChoosesSlot-defn
2 pairReading = reader.readerPair h2, post_readerChoosesSlot-defn
3 from $\overleftarrow{reader.readerPair} = \overleftarrow{writer.writerPair}$
...
infer $reader.readerPair = writer.writerPair \Rightarrow$
$(\neg wisOccurred \Rightarrow ...) \wedge$
$(wisOccurred \Rightarrow ...)$???
4 from $\overleftarrow{reader.readerPair} \neq \overleftarrow{writer.writerPair}$
infer $reader.readerPair = writer.writerPair \Rightarrow$
$(\neg wisOccurred \Rightarrow ...) \wedge$
$(wisOccurred \Rightarrow ...)$
readerChoosesSlot-defn,⇒-I-right-vac(4.h1)
infer $read\_Assertion(\sigma)$ 1,2,case-distinction(3,4)

Sub proof 3 above is discharged by case distinction on the value of the auxiliary variable *nwi*, which establishes which of the assertions for the writer network holds for each particular case, and therefore the value of the auxiliary variable *wisOccurred*. Rather than give a full rigorous description of sub-proof the following outline uses a case distinction based on the value of *wisOccurred*:

```
from pre-readerChoosesSlot(←σ); readerChoosesSlot_Assertion(←σ);
     ...; σ = readerChoosesSlot(←σ)
...
3    from reader.readerPair = writer.writerPair
3.1      from ¬wisOccurred
3.1.1        reader.readerSlot =
                 slotWritten(reader.readerPair))
                                          post_readerChoosesSlot-defn
         infer ¬wisOccurred ⇒                              h3.1, 3.1.1
                 reader.readerSlot =
                     slotWritten(reader.readerPair)
3.2      from wisOccurred
3.2.1        rcsSinceWis                  post_readerChoosesSlot-defn
3.2.2        reader.readerSlot =
                 slotWritten(reader.readerPair))
                                          post_readerChoosesSlot-defn
3.2.3        rcsSinceWis ⇒ reader.readerSlot =
                 slotWritten(reader.readerPair))     ⇒-I(3.2.1,3.2.2)
3.2.4        ¬rcsSinceWis ⇒ reader.readerSlot =
                 slotWritten(reader.readerPair))
                                          ⇒-I-right-vac(3.2.1)
         infer wisOccurred ⇒
         (rcsSinceWis ⇒
            reader.readerSlot = slotWritten(reader.readerPair))∧
         (¬rcsSinceWis ⇒
            reader.readerSlot ≠ slotWritten(reader.readerPair))
                                          ∧-I(3.2.3,3.2.4)
     infer reader.readerPair = writer.writerPair ⇒
         (¬wisOccurred ⇒ ...)∧
         (wisOccurred ⇒ ...)              case-distinction 3.1,3.2
...
infer read_Assertion(σ)                   1.2.case-distinction(3.4)
```

The proof of non-interference with the assertions in the writer network follows directly from the definitions of the assertions themselves. For example, the *writerChoosesPair_Assertion* is:

$$writerChoosesPair\_Assertion \triangleq \neg wisOccurred \wedge writer.writerPair = pairWritten$$

The auxiliary variable *wisOccurred* is only assigned to by the writer operations and the writer assertions refer to control variables that the only writer has write access to. Discharging the consistency proof obligations establishes that the reader and writer networks are inductive assertion networks. It remains to verify that the locations in the networks establish the required guarantee condition (that the reader and writer do not access the same slot in the mechanism at the same time):

$$nri = \text{RD} \wedge nwi = \text{WR} \Rightarrow ((reader.readerPair \neq writer.writerPair) \vee (reader.readerSlot \neq writer.writerSlot))$$

The interesting proofs are those to show that the guarantee condition holds after executing *readerChoosesSlot* and *writerChoosesSlot* and before executing *read* and *write*. The first of these proof obligations is:

$$\frac{\begin{array}{c} pre\_readerChoosesSlot(\overleftarrow{\sigma}); readerChoosesSlot\_Assertion(\overleftarrow{\sigma}); \\ firstWriterChoosesPair\_Assertion(\overleftarrow{\sigma}); \\ writerChoosesPair\_Assertion(\overleftarrow{\sigma}); writerChoosesSlot\_Assertion(\overleftarrow{\sigma}); \\ write\_Assertion(\overleftarrow{\sigma}); writerIndicatesSlot\_Assertion; \\ writerIndicatesPair\_Assertion(\overleftarrow{\sigma}); post\_readerChoosesSlot(\overleftarrow{\sigma}, \sigma) \end{array}}{\begin{array}{c} nri = \text{RD} \wedge nwi = \text{WR} \Rightarrow \\ ((reader.readerPair \neq writer.writerPair) \vee \\ (reader.readerSlot \neq writer.writerSlot)) \end{array}}$$

A rigorous proof is shown below. The complete proof is discharged by case-distinction on the value of *nwi*, although the only interesting part of the proof is where $nwi = \text{WR}$, where *write_Assertion* (which is h5 in the proof) expands to:

$$nwi = \text{WR} \Rightarrow \neg wisOccurred \wedge writer.writerSlot \neq slotWritten(writer.writerPair)$$

The composition proof obligations have not been separately discharged, because:

- The property that is required to hold of the composed system is the same as the property that has been shown to hold for the individual locations of the components..
- The rely condition of the system is that $rely = id$, in other words that no transition of the environment of the ACM affects the state of the composed ACM. The reader only relies on the writer accessing its chosen slot, and vice versa, which follows from the above rely condition.

from $pre\_readerChoosesSlot(\overleftarrow{\sigma})$; $readerChoosesSlot\_Assertion(\overleftarrow{\sigma})$
...; $write\_Assertion(\overleftarrow{\sigma})(h5)$: ...; $readerChoosesSlot(\overleftarrow{\sigma}.\sigma)$
1 from $reader.readerPair \neq writer.writerPair$
1.1 $(reader.readerPair \neq writer.writerPair) \vee$
$(reader.readerSlot \neq writer.writerSlot)$ $\vee$-I-right(1.h1)
infer $(nri = \text{RD} \wedge nwi = \text{WR} \Rightarrow$
$(reader.readerPair \neq writer.writerPair) \vee$
$(reader.readerSlot \neq writer.writerSlot)$ $\Rightarrow$-I-left(1.1)
2 from $reader.readerPair = writer.writerPair$
2.1 from $nwi = \text{WCP}$
infer $nri = \text{RD} \wedge nwi = \text{WR} \Rightarrow ...$ $\Rightarrow$-I-right-vac(2.1.h1)
2.2 from $nwi = \text{WCS}$
infer $nri = \text{RD} \wedge nwi = \text{WR} \Rightarrow ...$ $\Rightarrow$-I-right-vac(2.2.h1)
2.3 from $nwi = \text{WR}$
2.3.1 $reader.readerSlot =$
$slotWritten(reader.readerPair)$
post_readerChoosesSlot-defn
2.3.2 $\neg\, wisOccurred \wedge$
$\neg\, writer.writerSlot =$
$slotWritten(writer.writerPair)$ $\Rightarrow$-E-left(2.3.h1.h5)
2.3.3 $writer.writerSlot \neq$
$slotWritten(writer.writerPair)$ $\wedge$-E-left(2.3.2)
2.3.4 $reader.readerSlot \neq$
$writer.writerSlot$ 2.h1.2.3.1.2.3.3
2.3.5 $(reader.readerPair \neq writer.writerPair) \vee$
$(reader.readerSlot \neq writer.writerSlot)$
$\vee$-I-left(2.3.4)
infer $nri = \text{RD} \wedge nwi = \text{WR} \Rightarrow$
$((reader.readerPair \neq writer.writerPair) \vee$
$(reader.readerSlot \neq writer.writerSlot))$
$\Rightarrow$-I-left(2.3.5)
2.4 from $nwi = \text{WIS}$
infer $nri = \text{RD} \wedge nwi = \text{WR} \Rightarrow ...$ $\Rightarrow$-I-right-vac(2.4.h1)
2.5 from $nwi = \text{WIP}$
infer $nri = \text{RD} \wedge nwi = \text{WR} \Rightarrow ...$ $\Rightarrow$-I-right-vac(2.5.h1)
infer $nri = \text{RD} \wedge nwi = \text{WR} \Rightarrow$
$((reader.readerPair \neq writer.writerPair) \vee$
$(reader.readerSlot \neq writer.writerSlot))$
case-distinction(2.1.2.2.2.3,2.4.2.5)
infer $nri = \text{RD} \wedge nwi = \text{WR} \Rightarrow$
$((reader.readerPair \neq writer.writerPair) \vee$
$(reader.readerSlot \neq writer.writerSlot))$
case-distinction(1.2)

It should be noted that the property that has been verified to hold of the composed ACM in this section is not a guarantee-condition, since it does not relate the input state of the ACM to its output state (the property is that the reader and writer will not access the same slot at the same time). In each case assertions about the state of the ACM, and the required property, have been shown to hold both before and after each transition is taken in the assertion networks of the components. Therefore, while the proof method used is based on the rely-guarantee method described in Section 6.2, the proofs have been effectively been discharged using the proof method of Owicki-Gries [OG76c,OG76a], with an additional explicit test to verify non-interference between the individual components.

The verification proof to show that the 4-slot maintains (global) freshness of data is described in the next section (this property is a guarantee condition of a read action to the ACM).

6.3.4 The Freshness Proof

The freshness proof verifies that the 4-slot transmits globally fresh data between the reader and writer processes. The proof uses auxiliary variables to record extra history information about the data items that are available to the reader in a similar manner to the exhaustive proof for the abstract specification given in Section 4.2. The extra variables are:

newMaxFresh: Incremented by the writer at start write, to record the index of the new data item to be written to the ACM.

maxFresh: Used by the writer to indicate the index of the latest data item written to the ACM. This variable is set equal to *newMaxFresh* by the *writerIndicatesPair* operation.

minFresh: Used by the reader to record the index of the latest item available to be read, at the start of a read (by the *readerIndicatesPair* operation).

indexRead: Used by the reader to record the index of the data item it has chosen to read (by the *readerChoosesSlot* operation).

lastIndexRead: Used by the reader to record the value of *indexRead*, before it is updated to record the index of the item read during the current read.

These auxiliary variables are used in the guarantee condition to ensure that items read by the reader are (globally) fresh. This guarantee condition, when combined with the guarantee of data coherence, gives the required

property that the ACM is L-atomic. In the refinement proof the items read by the reader are related directly to items in the abstract sequence and equivalent behaviour is encoded into the model used in this proof using the above variables. An informal proof of this assertion is as follows:

1. When a write starts in the abstract model the item being written is added to the sequence as it is potentially available to the reader. This item potentially becomes available in the implementation once the writer has executed *writerIndicatesSlot* to indicate the slot it has accessed. The *start_write* operation in the abstract model includes the *writerIndicatesSlot* operation, so the new item becomes available at the same time.

2. If the reader and writer access the ACM at the same time, and the reader manages to read the item that was written by the write that is in progress, the sequence in the abstract model is shortened to only contain that item, and its index will be equal to *newMaxFresh*. In the implementation the reader can read the item that has been written, but not fully released and *minFresh* is set equal to *newMaxFresh* at the start of the next read if the writer has not executed *writerIndicatesPair* and updated *maxFresh*. In all other cases the oldest item in the sequence in the abstract model will be the one written by the last complete write to finish before the read starts. The equivalent item is pointed to by the control variables in the mechanism and *minFresh* is set equal to the index of this item at start read.

3. The retrieve relation maps the item read in the implementation to an item in the sequence in the abstract model. This model ensures that *indexRead* is greater than or equal to *minFresh* and less than or equal to *newMaxFresh* during the *read* operation. Therefore the item read is one that would be in the sequence in the abstract model at end read.

4. The index of the previous item read is recorded as *lastIndexRead* by the reader, and *indexRead* is verified to be greater than or equal to this value to verify the items are read in the order they are written. This property is guaranteed by the removal of all items older than the one read from the sequence in the abstract model.

The relationship between the above auxiliary variables depends on the recent history of the ACM, which can be in one of four states as shown in Figure 6.4. Each state is shown as a double rectangle: the left hand rectangle shows the relationship between the variables *pairReading*, *pairWritten*, *writerPair* and *readerPair*; and the right hand rectangle briefly describes

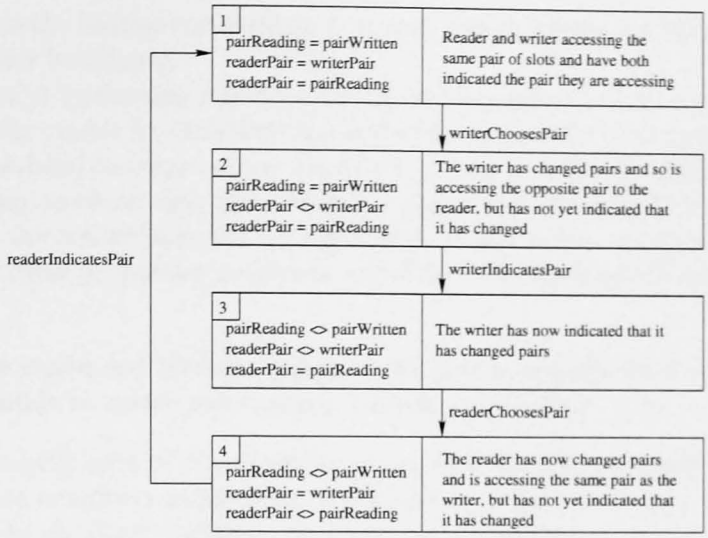

Figure 6.4: Relationship Between the Control Variables

the recent history of the behaviour of the ACM. The transitions between the states are labelled with the operations of the writer and reader that affect the relationship between the variables (and therefore affect the history of the ACM). The state where all of the variables are equal has arbitrarily been chosen as the start state[4]. This relationship evolves as follows:

1. When the writer next executes the *writerChoosesPair* operation it changes pair, and the value of the *writerPair* variable changes so that it is no longer equal to the value recorded by *readerPair*.

2. At the next *writerIndicatesPair* operation, the writer changes the value of *pairWritten* so that it is no longer equal to *pairReading*.

3. Once the writer has indicated it has changed pairs, the reader will follow it to the new pair when it next executes *readerChoosesPair*, and the value of *readerPair* will then no longer be equal to *pairReading*.

4. This situation will remain until the reader executes *readerIndicatesPair*, when all of the variables will again be equal, and the cycle starts again.

The only location in the reader network where it is possible that $readerPair \neq pairReading$, is after the reader has executed *readerChoosesPair* (location $lr1$) and it has changed pairs, and has not yet executed *readerIndicatesPair*

[4]It would be equally valid to record the relationship between the *pairWritten* and *writerPair* variables, rather than *readerPair* and *writerPair*.

to indicate the change (only states 1, 2, and 3 from Figure 6.4 are accessible in the other locations).

The most interesting assertion for the reader network is for location *lr3*, because the reader has acquired the slot to read, and the *read* operation (of the actual data) can start at any time ($nri = \text{RD}$). It is in this location that it is necessary to check that the reader is going to read acceptably fresh data[5], and that the writer does not interfere with it (the coherence proof above). Together these properties guarantee atomicity. The assertion is described as follows:

- The reader has already indicated the pair it is going to access so it is possible to assert $pairReading = reader.readerPair$.
- The next part of the assertion is required to establish that the guarantee condition holds when a read takes place and relates the values of the auxiliary variables. In most cases this states:

  $minFresh \leq maxFresh \wedge indexRead \leq maxFresh \wedge$
  $indexRead \geq minFresh \wedge lastIndexRead \leq indexRead$

  If, however the reader and writer are accessing the same pair ($readerPair = writerPair$), a write is in progress and the writer has indicated the slot it is accessing ($wisOccurred = \text{true}$), and the reader subsequently chose the slot it is going to access ($rcsSinceWis = \text{true}$), it is possible for the reader to read the item that the writer has written during the current write:

  $minFresh \leq newMaxFresh \wedge indexRead \leq newMaxFresh \wedge$
  $indexRead \geq minFresh \wedge lastIndexRead \leq indexRead$

- It is also necessary to relate the value of $minFresh$ to the index of an item in one of the slots in the ACM. There are two cases to consider:

  1. It the reader and writer are accessing the same pair of slots, or if the reader is accessing the opposite pair to the writer and the writer has not yet completed the first write to the new pair (state 2 above), the first item that is available to be read is at least as old as the last item fully released by the writer:

     $minFresh \leq slots(pairWritten, slotWritten(pairWritten)).index$

  2. The other case is where the reader and writer are accessing different pairs and the writer has completed the first write to its

[5]There is an underlying assumption that the reader will receive a fair share of its processor's resources: i.e. that it will not be held up for a long period of time when it reaches this stage of its execution cycle, so that it can complete the read.

current pair. In this case the first item available to the reader is at least as old as the last one written to the pair it is accessing:
$(pairWritten = p0 \Rightarrow minFresh \leq slotWritten(p1).index) \wedge$
$(pairWritten = p1 \Rightarrow minFresh \leq slotWritten(p0).index)$

The complete assertion is:
$nri = \text{RD} \Rightarrow reader.readerPair = pairReading \wedge$
$(pairReading = pairWritten \wedge reader.readerPair = writer.writerPair \wedge$
$reader.readerPair = pairReading \Rightarrow$
$(\neg wisOccurred \Rightarrow$
$minFresh \leq maxFresh \wedge indexRead \leq maxFresh \wedge$
$indexRead \geq minFresh \wedge lastIndexRead \leq indexRead) \wedge$
$(wisOccurred \Rightarrow$
$(\neg rcsSinceWis \Rightarrow$
$minFresh \leq maxFresh \wedge indexRead \leq maxFresh \wedge$
$indexRead \geq minFresh \wedge lastIndexRead \leq indexRead) \wedge$
$(rcsSinceWis \Rightarrow$
$minFresh \leq newMaxFresh \wedge indexRead \leq newMaxFresh \wedge$
$indexRead \geq minFresh \wedge lastIndexRead \leq indexRead)) \wedge$
$minFresh \leq slots(pairWritten, slotWritten(pairWritten)).index) \wedge$
$(pairReading = pairWritten \wedge reader.readerPair \neq writer.writerPair \wedge$
$reader.readerPair = pairReading \Rightarrow$
$minFresh \leq maxFresh \wedge indexRead \leq maxFresh \wedge$
$indexRead \geq minFresh \wedge lastIndexRead \leq indexRead \wedge$
$minFresh \leq slots(pairWritten, slotWritten(pairWritten)).index) \wedge$
$(pairReading \neq pairWritten \wedge reader.readerPair \neq writer.writerPair \wedge$
$reader.readerPair = pairReading \Rightarrow$
$minFresh \leq maxFresh \wedge indexRead \leq maxFresh \wedge$
$indexRead \geq minFresh \wedge lastIndexRead \leq indexRead \wedge$
$(pairWritten = p_0 \Rightarrow minFresh \leq slots(p_1, slotWritten(p_1)).index) \wedge$
$(pairWritten = p_1 \Rightarrow minFresh \leq slots(p_0, slotWritten(p_0)).index))$

The assertions for the locations in the writer network are all very similar. They relate the values of the writer local variables to the values of the control variables, and keep track of the value of the *maxFresh* auxiliary variable. In addition they encode the relationship between the indices of the items that are currently in the 4 slots in the mechanism. For example that for location $lw3$, when the writer is about to execute the *writerIndicatesSlot* operation is explained below:

1. The first three conjuncts relate to values of auxiliary variables: the program counter *nwi* is equal to WIS; the auxiliary variable *wisOccurred* is equal to false (since the writer has not yet indicated which slot it

is accessing) ; and *newMaxFresh* was incremented at the start of the write so it is equal to *maxFresh* + 1:

$nwi = \text{WIS} \Rightarrow \neg wisOccurred \wedge maxFresh = newMaxFresh - 1$

2. When the mechanism is in states 3 and 4 in Figure 6.4 the writer has changed pairs and completed a write, by executing *writerIndicatesPair*, but the reader has not yet changed pairs and indicated that it has changed. The *pairWritten* and *pairReading* control variables are therefore not equal and the writer local variable *writerPair* is equal to the *pairWritten* control variable:

   $pairWritten \neq pairReading \Rightarrow pairWritten = writer.writerPair$

3. The item the writer has just written during the *write* operation is in the slot pointed to by the writer local variables, and its index is equal to *newMaxFresh*. The item written during the last write is pointed to by the control variables in the ACM and its index is equal to *maxFresh*:

   $maxFresh = slots(pairWritten, slotWritten(pairWritten)).index \wedge$
   $newMaxFresh = slots(writer.writerPair, writer.writerSlot).index$

4. The remainder of the assertion relates the indices of the remaining slots to the value of *maxFresh*. This encodes the order that the items were written (this encoding is equivalent to the ordering of the items in the sequence in the abstract model). If the writer did not change pairs at start write this is stated as:

   $writer.writerPair = pairWritten \Rightarrow$
   $(pairWritten = p_0 \Rightarrow slots(p_1, s_0).index \leq maxFresh - 1 \wedge$
   $slots(p_1, s_1).index \leq maxFresh - 1) \wedge$
   $(pairWritten = p_1 \Rightarrow slots(p_0, s_0).index \leq maxFresh - 1 \wedge$
   $slots(p_0, s_1).index \leq maxFresh - 1)$

   If the writer did change pairs following holds:

   $(writer.writerPair \neq pairWritten \Rightarrow$
   $(slotWritten(pairWritten) = s_0 \Rightarrow$
   $slots(pairWritten, s_1).index \leq maxFresh - 1) \wedge$
   $(slotWritten(pairWritten) = s_1 \Rightarrow$
   $slots(pairWritten, s_0).index \leq maxFresh - 1) \wedge$
   $(writer.writerSlot = s_0 \Rightarrow$
   $slots(writer.writerPair, s_1).index \leq maxFresh - 1) \wedge$
   $(writer.writerSlot = s_1 \Rightarrow$
   $slots(writer.writerPair, s_0).index \leq maxFresh - 1))$

Putting this together gives the complete assertion:

$$
\begin{array}{l}
nwi = wis \Rightarrow \neg wisOccurred \wedge maxFresh = newMaxFresh - 1 \wedge \\
\quad (pairWritten \neq pairReading \Rightarrow pairWritten = writer.writerPair) \wedge \\
\qquad writer.writerSlot \neq slotWritten(writer.writerPair) \wedge \\
\qquad maxFresh = slots(pairWritten, slotWritten(pairWritten)).index \wedge \\
\qquad newMaxFresh = slots(writer.writerPair, writer.writerSlot).index \wedge \\
\qquad (writer.writerPair = pairWritten \Rightarrow \\
\qquad\quad (pairWritten = p_0 \Rightarrow slots(p_1, s_0).index \leq maxFresh - 1 \wedge \\
\qquad\qquad\qquad\qquad slots(p_1, s_1).index \leq maxFresh - 1) \wedge \\
\qquad\quad (pairWritten = p_1 \Rightarrow slots(p_0, s_0).index \leq maxFresh - 1 \wedge \\
\qquad\qquad\qquad\qquad slots(p_0, s_1).index \leq maxFresh - 1)) \wedge \\
\qquad (writer.writerPair \neq pairWritten \Rightarrow \\
\qquad\quad (slotWritten(pairWritten) = s_0 \Rightarrow \\
\qquad\qquad slots(pairWritten, s_1).index \leq maxFresh - 1) \wedge \\
\qquad\quad (slotWritten(pairWritten) = s_1 \Rightarrow \\
\qquad\qquad slots(pairWritten, s_0).index \leq maxFresh - 1) \wedge \\
\qquad\quad (writer.writerSlot = s_0 \Rightarrow \\
\qquad\qquad slots(writer.writerPair, s_1).index \leq maxFresh - 1) \wedge \\
\qquad\quad (writer.writerSlot = s_1 \Rightarrow \\
\qquad\qquad slots(writer.writerPair, s_0).index \leq maxFresh - 1))
\end{array}
$$

The Freshness Proof Obligations

It is necessary to discharge identical proof obligations for each transition, in order to verify that the ACM maintains freshness of data, as were necessary to verify it communicated coherent data between its reader and writer. First to discharge the consistency proofs to show that the reader and writer networks are inductive assertion networks, and that the reader and writer do not interfere with each other; and then to show that the individual components meet their guarantee conditions. Once again it is not necessary to explicitly discharge the proof obligations required by the composition rule, since the guarantee conditions of the components are identical to the guarantee condition of the ACM, and the rely condition of the composed system on its environment is $Rely \models id$.

The proofs for the reader and writer networks are discharged in the same manner as for the freshness proofs, although they are more complicated to discharge. This is because it is necessary to use case distinctions to discharge the proofs for the possible different states of the ACM, and the different possible values of the control variables. The required guarantee condition, which is established directly by *read_Assertion*, and guarantees that the reader will return a fresh item as the result of a read as described earlier, is:

$$
\begin{array}{l}
nri = \text{RD} \Rightarrow minFresh \leq newMaxFresh \wedge indexRead \leq newMaxFresh \wedge
\end{array}
$$

$$indexRead \geq minFresh \wedge lastIndexRead \leq indexRead.$$

Having shown how this proof method can be used to verify the correct operation of an ACM the next section shows how the method can be used to identify and correct defects in an ACM, using a 3-slot ACM implementation as an example. The complete model described in this section is given in Appendix H in the PVS logic[6]. The proof of L-atomicity described in this section was completed in the style described in [OG76b, OG76c], in the same way as the proof of coherence given in Section 6.3, however in this case the property verified is a guarantee condition of a complete read to the ACM.

6.4 Identifying and Correcting Defects in a 3-slot ACM Implementation

This section first briefly describes a proof that the 3-slot implementation given in Chapter 3 does not guarantee coherence of the data items communicated, because it is possible for the reader and writer to access the same slot in the mechanism at the same time. It then shows how the method can be used to identify the sequence of actions that lead to this incorrect behaviour.

The assertion networks for the reader and writer for the 3-slot implementation are given in Figure 6.5 and Figure 6.6, respectively.

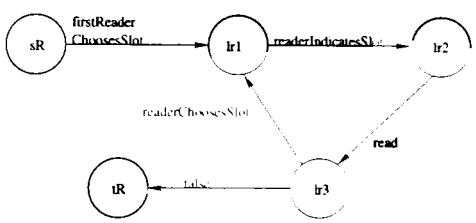

Figure 6.5: Assertion Network for the Reader to the 3-slot

The complete model of this implementation, assertions for the locations in the reader and writer assertion networks, and proof obligations are given in Appendix I. The guarantee condition required for coherence is:

$nwi = \text{WR} \wedge nri = \text{RD} \Rightarrow readerSlot \neq writerSlot$
where *writerSlot* is the slot the writer has acquired and *readerSlot* is the

[6]The interested reader can download the PVS theory, and proof scripts, from http://homepages.cs.ncl.ac.uk/neil.henderson/fme2003/atomic.tgz.

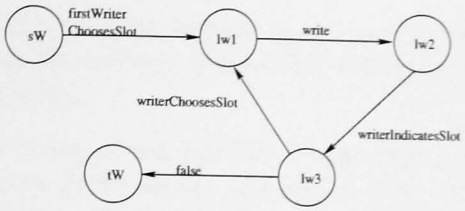

Figure 6.6: Assertion Network for the Writer to the 3-slot

slot the reader has acquired (*slotWritten* is the control variable used by the writer to indicate the last slot it accessed, and *slotReading* is the control variable used by the reader to indicate the last slot it chose to access in this model).

Using PVS, it is possible to identify a number of sets of witness values of these variables for which it can be shown that the guarantee condition will not hold when the *readerIndicatesSlot* operation is executed. For example: $nri = ris, nwi = \text{WR}, slotWritten = s2, writerSlot = s1, readerSlot = s1$[7]. It is possible to confirm that this satisfies the property that the reader and writer access the same slot at the same time by proving the following conjecture, instantiating ACM state in the existential quantifier with the appropriate values as above:

$$\frac{\begin{array}{c}\exists\, \overleftarrow{\sigma} : Conc\_State \cdot pre\_readerIndicatesSlot(\overleftarrow{\sigma}) \wedge \\ writerChoosesSlot\_Assertion(\overleftarrow{\sigma}) \wedge write\_Assertion(\overleftarrow{\sigma}) \wedge \\ writerIndicatesSlot\_Assertion(\sigma) \wedge readerIndicatesSlot(\overleftarrow{\sigma}, \sigma);\end{array}}{nwi = \text{WR} \wedge nri = \text{RD} \wedge reader.readerSlot = writer.writerSlot}$$

The chain of events leading to this situation can be identified as follows:

1. It is already known that the last operation that was executed was *readerIndicatesSlot*.

2. the writer is currently writing the data item, so the last write operation must have been *writerChoosesSlot*.

3. The *readerChoosesSlot* operation, which occurred at the start of the read, set *readerSlot* equal to *slotWritten*. Therefore at this stage *slotWritten* must also have been equal to $s1$, which means *readerChoosesSlot* occurred before *writerIndicatesSlot*, which then set *slotWritten* to $s2$.

[7]This example is taken from [Sim90a].

4. The order of the operations must therefore have been *readerChoosesSlot*, *writerIndicatesSlot*, *writerChoosesSlot*, *readerIndicatesSlot*.

Table 6.1 shows values of the variables at the different stages of the above interleaving of actions of the reader and writer (the value of *slotReading* is not important until after *readerIndicatesSlot* has been executed).

	slotWritten	slotReading	writerSlot	readerSlot
initial values	s1	s0	s2	-
readerChoosesSlot	s1	s0	s2	s1
writerIndicatesSlot	s2	s0	s2	s1
writerChoosesSlot	s2	s0	s1	s1
readerIndicatesSlot	s2	s1	s1	s1

Table 6.1: Incorrect Operation of the 3-slot ACM

The full model of the 3-slot, with details of the proof obligations is given in Appendix I. A proof is also given that shows the implementation is correct provided that a timing constraint, from [Sim90a], which ensures that the above interleaving of actions of the reader and writer cannot occur, and a proof of correctness for a revised 3-slot implementation from [XYIS02].

6.5 Summary

This work described in this chapter has shown that it is possible to verify the correctness of asynchronous networks of processes using rely-guarantee, even where the component processes give few or no guarantees about their own behaviour. Rather the correctness of the *composed system* is a result of the behaviour that emerges from the asynchronous operation of the components, and in the particular example the individual components provide guarantees that are equivalent to the guarantee condition required of the complete system. The verification of correctness of the system described requires proofs to be completed with respect to an infinite state space (an unbounded number of reads can occur concurrently with a single write, and vice versa). It may be possible to represent the 4-slot implementation as a finite state model, since the number of buffers that can contain data is finite, and the implementation can only contain four data items at any one time. This may, however, require some form of data abstraction, or a means to prove that particular states are *equivalent* (bi-similar), even though they contain different values, from the data type that is being communicated, in the individual slots.

The advantages of the method are that:

1. It is not necessary to identify the entire state space of the system.
2. A reduced number of proofs need to be discharged than would be required for an exhaustive proof of the entire state space.
3. It may be possible to use the rely-guarantee conditions that have been shown to hold for the 4-slot in verifying properties of systems where the ACM is itself used as a component.

It may be easier to verify the required properties hold of a system by model checking, but the method described here provides much greater insight into the behaviour of the system. The requirement to give assertions about the behaviour of the reader and writer requires a greater understanding of the behaviour of the system than may be gained from model checking, and gaining this extra understanding may be considered to be worth the extra effort associated with the method. For example discovering the assertions for the locations in the reader and writer networks and discharging the proof obligations identified that the are different points at which the reader can effectively acquire the item it is going to read. If the reader accesses the opposite pair of slots to the writer the item is acquired when *readerChoosesPair* is executed, because the writer cannot do anything to interfere with the choice of slot in the chosen pair. If, however the reader is accessing the same pair as the writer the slot acquired will depend on the ordering of the *readerChoosesSlot* and *writerIndicatesSlot* operations. Discharging the consistency proof obligations to show that the reader and writer networks are inductive assertion networks can also help to identify inconsistencies in the model, and increase confidence in the correctness of the proofs. For example an inconsistency may make it impossible to verify non-interference between the reader and writer.

The disadvantage of this method is that the individual proofs may be more complex than the individual proofs for an exhaustive proof, but the advantages above outweigh this disadvantage. It should be borne in mind that this method is only applicable where state machines can be constructed to represent the behaviour of the components, and sufficiently strong assertions can be found to ensure that the guarantee condition of the system can be met. In the case study described in this thesis the guarantee condition was met by the individual components, although in general it may be met by a composition of the guarantee conditions of those components.

It has also been demonstrated that the method can be used identify erroneous behaviour of incorrect specifications, in Section 6.4, using a 3-slot implementation as a case study. Once the erroneous behaviour has been identified it is may then be possible to identify corrections to the specification. Verifying the correctness of the revised specification may be made

easier by the understanding that has already been gained about the interactions of the components.

The verification proofs described in this chapter hold for interleaved concurrent implementations, where the individual operations of the component processes are atomic. The proofs may not hold for fully asynchronous systems and in the case of the 4-slot it must be recognised that, in any implementation where the above atomicity assumption cannot be guaranteed to hold, asynchronous accesses to control variables in the mechanism could result in an attempt to read one of those variables when it is being overwritten. This could result in the reader of the variable not returning the value that was written. In fact the value returned may not even be a valid one. It may be possible to extend the models described in this chapter to verify properties of the 4-slot implementation (and other fully asynchronous systems) when the above assumption is relaxed, but the assertions may be significantly more complex, and the proofs may be daunting, if not intractable. Chapter 7 shows how model checking can be used to prove properties of fully asynchronous implementations of the 4-slot and a full comparison of the relative advantages and disadvantages of the two methods is given in Chapter 8.

Chapter 7

Model Checking Simpson's 4-slot ACM

Chapters 5 and 6 have shown how a range of formal tools can be used to explore the behaviour of asynchronous real-time systems. Starting with an abstract model of the required properties of the system it is possible to gain an understanding of its behaviour in an incremental manner and verify properties of the system as understanding improves. The rely-guarantee method, described in Chapter 6, can be used to verify properties of implementations where the actions of the individual components can interleave in an unconstrained manner i.e. those actions are Hoare atomic. Simpson's 4-slot uses control variables to direct the reader and writer to different buffers, if they access the ACM concurrently, so that the reader can read coherent fresh data. However, it is possible for the reader and writer to access the same control variable at the same time in fully asynchronous implementations, and the reader may attempt to read the variable when it is changing value. The control variables are single bits, but there is no guarantee that the reader will return the value that is being written in these circumstances. It is also possible that reading a control variable while its value is changing will cause the reader to become *metastable*, in which case it may take an arbitrarily long period of time to decide whether it has read a zero or a one. Theoretically there is a very small probability that the reader of the control variable will never decide on the precise value it has read.

This chapter describes joint work by the author, Paynter and Armstrong to model the 4-slot, that recognises that the reader and writer can clash on accesses to control variables[PHA04]. The models, which are due to Paynter, are in CSP, [Ros98], and this work has shown, using the FDR model checker, [FSE96], that the ACM is L-atomic even when such clashes occur, provided measures are taken to contain the effects of metastability. The chapter is organised as follows. Section 7.1 describes the fundamental prob-

lem of metastability. Section 7.2 briefly introduces CSP[1] and the FDR model checker and Section 7.3 describes an series of increasingly sophisticated models to describe the behaviour of the (single bit) control variables that are used by the 4-slot implementation. Section 7.4 gives a CSP model of the 4-slot ACM implementation and Section 7.5 describes how the models have been checked with FDR to show that the 4-slot is L-atomic. Section 7.6 briefly introduces some recent work which has modelled the effects of metastability on a number of other ACM implementations, and Section 7.7 describes the conclusions from this work.

7.1 Metastability

Metastability is a fundamental problem of systems that have two or more stable states, which respond to inputs that are *connected inputs*: that is inputs that are either continuous in time (for example hardware latches), continuous in value, or both. The state space of the system is divided into stable states, *regions* of unstable states which must lead to stable states and metastable states between the unstable states. The unstable region (and hence stable state) which will be entered from a metastable state, and the length of time that it takes to enter such a region is undetermined. It has been shown, however, that the probability that a system remains in a metastable state decreases exponentially with time, [Män88, KBY02].

An important class of systems that can exhibit metastability are digital electronic circuits that synchronise asynchronous inputs, [CM73, HEC89]. The two binary states of the circuit are the stable states, but it is possible for the asynchronous input to a latch to occur arbitrarily close to the synchronising (latching) clock pulse, causing the device to read a changing input, which will not have a clear binary value. The synchroniser, or latch, then enters a metastable state, where its value lingers indefinitely (potentially infinitely) between the two valid, stable, states. It is possible for this metastable value, while it is an invalid digital value, to induce metastability on a circuit that reads it, and thus propagate metastability further through the system. Note that metastability is not induced in the input (writing) circuit.

A number of approaches can be adopted to minimise the effects of metastability in practice[Män88]. One solution, from [Cha87], is to use a detector to detect when a value read is metastable, and to pause the reader's clock until the value settles to a valid value. This requires the ability to stop a system's clock for an arbitrary length of time, so the system has to be able to cope

[1]The interested reader may refer to [Ros98] for a more complete description of the CSP language.

with arbitrary pauses in its operation. This is not the same as causing the system to skip clock cycles, because the *end of metastability* signal is itself asynchronous, and could itself be the cause of further metastability in the system. In this solution the clock cycle, on resumption, may not be *in phase* with its cycle prior to the clock being stopped.

A more common solution, and one that is adopted in many implementations is to accept that an asynchronous read of a value may cause the reader to become metastable, and to ensure that the system waits for a period of time after latching values before they are used, to allow the value to settle to a binary one. A suitably long *wait duration* can ensure that the probability of using a metastable value is reduced as far as is needed. This is a particularly practical solution where the reading processor is implemented in software, because processor clock speeds are typically slow compared to the time it takes for metastable values to settle to a stable state. In such systems it is feasible to engineer circuits so that the expected mean time between failure due to using a metastable value is vanishingly small[2], and insignificant compared to other sources of failure.

In fully asynchronous ACM implementations it is possible for the reader and writer to access any of the control variables at the same time. Such concurrent (clashing) accesses can lead to the reader returning a metastable value, and one of the engineering solutions described above must be used to reduce the possibility of this happening to an acceptably low probability.

This chapter describes models of the 4-slot: where the assumption about the atomicity of actions has been removed; and which explore the different behaviour of the ACM when metastability occurs, and different methods are employed to contain the effects of metastability. Models that recognise the reader and writer can clash on accesses to control variables, and take account the methods that can be employed to contain the effects of metastability, have been produced using CSP. It has then been shown, using FDR, that the 4-slot is still L-atomic, and preserves coherence of data, even when metastability occurs. This work has also shown that, if the effects of metastability and the different methods for its containment are not modelled correctly, any results obtained from model checking may be suspect.

7.2 CSP and the FDR Model Checker

CSP is a process algebra, which can be used to specify the behaviour of concurrent systems that are composed of a number of communicating pro-

[2]For example, it is claimed to be possible to design circuits which have a mean time between failures due to metastability of 10^{204} or 10^{420} years, [Gin03]. (The age of the universe is thought to be in the order of 10^{10} years.)

cesses. Each process is specified separately, for example a simple process that carries out two actions, a followed by b, could be described as:

$$proc1 : a \rightarrow b \rightarrow STOP \tag{7.1}$$

A second process that carries out an action c and then synchronises with *proc1* on action b could be specified as

$$proc2 : c \rightarrow b \rightarrow STOP \tag{7.2}$$

Specifying processes in this way only allows synchronous communication between components, although the processes are otherwise asynchronous. This limitation can be overcome by defining actions using start and end events, thus it is possible to model concurrent accesses to communication mechanisms by the component processes. The specification of a system defines a set of traces of the actions of these component processes.

The FDR tool verifies properties of systems, that are specified using CSP_M (the machine readable version of CSP), by analysing all of the possible traces that are allowed by the CSP model. For example the tool is able to verify that a model of an implementation is a refinement of a model of a specification by verifying that their traces are equivalent (it may be necessary to *hide* the internal actions of the implementation, so that both models only execute equivalent visible actions).

7.3 Modelling Bit Control Variables

[PHA04] describes a series of increasingly sophisticated models of bit variables, in CSP, that model the effects of metastability and take account of constraints that need to be observed in implementations to cope with it. This section describes a selection of these models in detail and briefly introduces the remainder of the models[3]. Full details of the CSP models from [PHA04] are given in Appendix J.

First the basic definitions are introduced and described below:

```
max_no_of_values = ...
data_values = {1..max_no_of_values}
datatype bit_values = b0 | b1 | d
datatype slot_index = s1 | s2 | s12
datatype pair_index = p1 | p2 | p12
```

[3]The models are given in CSP_M, the machine readable version of CSP,which is used by the FDR tool.

```
bs(b0) = s1  -- convert bit values to slot indices
bs(b1) = s2
bs(d) = s12
bp(b0) = p1  -- convert bit values to pair indices
bp(b1) = p2
bp(d) = p12
sb(s1) = b0  -- convert slot indexes to bit values
sb(s2) = b1
sb(s12) = d
pb(p1) = b0  -- convert pair indexes to bit values
pb(p2) = b1
pb(p12) = d
toggle(b0) = b1  -- toggle (invert) bit values
toggle(b1) = b0
toggle(d) = d
```

1. *max_no_of_values* defines the maximum number of different values that can be communicated between the writer and reader in the model, and this value needs to be kept reasonably small (2 or 3) in most cases for model checking purposes. When model checking that data items are read in the order they are written (a crucial property for atomic ACMs), however, where the number of values may be more crucial, this figure has been increased to 10 by Paynter.

2. A three valued data type called *bit_values* is used to model the possible values that can be returned by a reader of the variable: the values $b0$ and $b1$ represent the valid values, 0 and 1, and the third value d represents the metastable (*dithering*) value that a reader may return in some of the models if a read occurs concurrently with a write to the variable.

3. The above *bit_values* are converted into pair indices and slot indices, as appropriate, to index into the four slot mechanism, and there are data values to represent the slot indices and pair indices. Once again these have three values: $s0$ ($p0$) and $s1$ ($p1$) to represent the valid values and $s12$ ($p12$) to represent the metastable value.

4. There are *functions* to convert between pair and slot indices and bit values, and also a *function* to toggle the bit values[4] (the dithering value is left unchanged by this function).

[4]This toggle function is used, for example, when the writer chooses the pair it is going to write to at *writerChoosesPair*. It chooses to write to the opposite (*toggle* of the) pair that the reader last indicated it is accessing.

The following section introduces the various models of the bit variables, and describes one of the models in detail.

7.3.1 Models of the BIT variables

The different models of the BIT variables are summarised as follows:

BIT0: The basic model, which is of a Hoare atomic bit variable, to reflect the assumption made in Chapter 6, that accesses to the control variables in the 4-slot are atomic. This model is included for completeness.

BIT1: is a type-safe Bit Variable.

BIT2: This model is a revised type-safe model, where the value recorded by the bit variable does not flicker when it is being overwritten with the same value. This means that a read that clashes with a write in these circumstances returns the value being written.

BIT3: The previous models, BIT0 to BIT2, fail to capture the behaviour when metastability occurs in the reader of the bit variable, or to model the measures that can be taken to contain the effects of metastability. Chapiro's solution, [Cha87], which was described in Section 7.1, is captured by this model where a metastability detector is used to delay the system clock until metastability is resolved. The model can diverge, since there is a possibility that metastability will never be resolved. The behaviour of this model is not, therefore, that of an type-safe ACM, even though the value returned where metastability does resolve itself is decided by internal non-determinism as in the type-safe model (BIT1).

BIT4: The first to model the possible consequences of metastability. The control variables an return an extra *dithering* value, d. This dithering value can be used to model the situation where the reader and writer of a control variable clash and the reader returns a metastable value, which is then copied into a local variable. In this model, which is given in full below, the reader can access the control variable any number of times while it is being written.

BIT5: Digital circuits have maximum speeds at which they can be operated, because the components (latches) from which they are built have minimum set up and hold times that must be observed, if they are to operate within their specifications. In addition to the above restriction, digital electronic circuit implementations of bit variables have maximum switching (or propagation) times, which are much (orders

of magnitude) shorter than the clock speeds of processor clocks. Taken together the above timing constraints limit the number of times that a reader can access a bit variable while it is switching value. The BIT5 model modifies the BIT4 process to reflect these timing constraints.

BIT6: This is a modification of BIT5, where the value recorded by the bit variable can be disturbed when it is overwritten with the same value. It is therefore possible for any read that clashes with a write to return a valid value, or a dithering one.

BIT4 - A Bit Variable That Can Return Metastable Values

A realistic way of modelling the effects of metastability on the 4-slot implementation in CSP is to extend the alphabet of the *sw* and *er* channels of the bit variables that are used to model the control variables with an extra *dithering* value, *d*. This dithering value can be used to model the situation where the reader and writer of a control variable clash and the reader returns a metastable value, which is then copied into a local variable. The first parameter of the *BIT4* process, *var_name*, is used to instantiate instances of the process with the names of the local variables that are being modelled, and *sw* and *ew*, and *sr* and *er*, model the start and end of a write and start and end of the read, respectively, to the variable.

The CSP model of the bit (control) variables is:

```
BIT4(var_name, val) =
     var_name.sw?x ->
        (if x == val then BIT4_w_stable(var_name, val)
         else BIT4_w(var_name, val, x))
     [] var_name.sr -> BIT4_r(var_name, val)

BIT4_w(var_name, val, x) =
         var_name.ew -> BIT4(var_name, x) []
         var_name.sr -> BIT4_wr(var_name, val, x)

BIT4_r(var_name, val) =
     var_name.sw?x ->
        (if x == val then BIT4_wr_stable(var_name, val)
         else BIT4_wr(var_name, val, x))
     [] var_name.er!val -> BIT4(var_name, val)

BIT4_wr(var_name, val, x) =
      var_name.ew ->
          BIT4_r_clashed(var_name, x) []
```

```
            (var_name.er!b0 -> BIT4_w(var_name, val, x) |~|
             var_name.er!b1 -> BIT4_w(var_name, val, x) |~|
             var_name.er!d -> BIT4_w(var_name, val, x))

BIT4_r_clashed(var_name, val) =
      var_name.sw?x ->
         BIT4_wr(var_name, val, x) []
           (var_name.er!b0 -> BIT4(var_name, val) |~|
            var_name.er!b1 -> BIT4(var_name, val) |~|
            var_name.er!d -> BIT4(var_name, val))

BIT4_w_stable(var_name, val) =
      var_name.ew -> BIT4(var_name, val) []
      var_name.sr -> BIT4_wr_stable(var_name, val)

BIT4_wr_stable(var_name, val) =
      var_name.ew -> BIT4_r(var_name, val) []
      var_name.er!val -> BIT4_w_stable(var_name, val)
```

The introduction of metastability into the models of the bit variables requires a change to the processes that use them, so that, if the reader of a variable accesses it at the same time as the writer, and returns a metastable value, this metastable value has the chance to settle to a stable one before it is used. The approach that has been taken is to introduce an additional process to model the local copy of the variable, and to model the measures that can be taken to contain the effects of metastability. There are two variants of this additional process, which are described in Section 7.3.2.

7.3.2 LB1 and LB2 - Local Copies of the Control Variables

The *BIT*0 to *BIT*3 models of bit variables above assume that the reader of the bit variable will return a valid value as the result of a read, or, in the case of the *BIT*3 model, the reader may never decide on the value returned and so may diverge. A simple example reader process that reads a control variable modelled by one of these processes, and uses the value returned twice, could be modelled as:

```
READER1 = sr -> er?x -> use1(x) -> use2(x) -> READER1
```

where use1(x) and use2(x) are two arbitrary uses of the value.

The remaining models, *BIT*4 to *BIT*6, allow the reader of the variable to return a metastable value, which may, but is not guaranteed to, decay to

a stable value at some future time. The value returned must therefore be given the opportunity to settle to such a stable value before it is used. The approach that has been taken is to add a process to model the local copy of the variable, and an example process makes use of the local variable could be (in this and the remaining definitions the *set* and *get* processes are used to write values to and read values from the local variables, respectively):

```
READER2 = sr -> er?x -> set!x -> get?x -> use1(x) -> get?x ->
          use2(x) -> READER2
```

This process first reads the control variable and copies the value read to the local variable (potentially this value can be the metastable value, d). The value is then read back from the local variable on each occasion, immediately before it is used, to allow the value to settle to a binary one, should the original read return a metastable value. In general the reader must re-read the local variable on each occasion, before its value is used, to allow for any metastability to be resolved.

A CSP process to model the behaviour of the local variable could be:

```
LB1(val) =  if val == d then
               (LB1(b0) |~| LB1(b1) |~| (set?x -> LB1(x) []
                                         get!val -> LB1(val)))
            else (set?x -> LB1(x) [] get!val -> LB1(val))
```

This process allows the value of a local bit to be *set*, and resolves metastable values non-deterministically into binary ones each time they are read. This model allows multiple reads of a variable while it is metastable, which is theoretically possible (bearing in mind it is possible for metastable values to take an infinite amount of time to resolve to binary ones), however, as was stated in Section 7.1, it is possible to engineer digital circuits so that the reader waits for a short period of time, before making use of any value returned as the result of a read. This reduces the chance that a value will still be metastable when it is used to a very small probability. A model of a local bit variable that is engineered in this way is:

```
LB2(val) = set?x -> (if x == d then LB2(b0) |~| LB2(b1)
                     else LB2(x)) [] get!val -> LB2(val))
```

This process resolves metastable values to stable ones as they are set, so that only stable values can be used by the reading process (subsequent reads of the value from the local variable get the same non-deterministically chosen value after it has been set to the metastable one (d))[5].

[5]While this ignores the theoretical possibility that the metastable value will not have settled to a stable one before it is used, the probability of this occurring is very small as described in Section 7.1.

7.4 A CSP Model of the 4-slot

The CSP model of the 4-slot, is built on the unpublished model, due to White [Whi01] and this section gives the definitions of the reader and writer processes that use the local copies of the control variables. The processes used in the remaining definitions model relate to the variables in the ACM implementation given in Section 3.3 as follows:

1. *reading* is equivalent to *pairReading*.
2. *latest* is equivalent to *pairWritten* and *not_pair_written* is used to indicate that the value needs to be *toggled* (negated) when it is saved to the local variable.
3. *LB_write_pair* and *LB_write_slot* are equivalent to the local variables *writerPair* and *writerSlot* respectively.
4. *writers_slots* represents the *slotWritten* array, and *not_slot_written* is again used to indicate that the value needs to be negated before it is saved to the local variable.
5. *LB_read_pair* and *LB_read_slot* represent the reader local variables *readerPair* and *readerSlot* respectively.

```
Fourslot_Writer_LB =
    start_write?val -> reading.sr ->
    reading.er?not_pair_written ->
    LB_write_pair.set!toggle(not_pair_written) ->
    LB_write_pair.get?pair_written ->
    writers_slots.bp(pair_written).sr ->
    writers_slots.bp(pair_written).er?not_slot_written ->
    LB_write_slot.set!toggle(not_slot_written) ->
    LB_write_slot.get?slot_written ->
    LB_write_pair.get?pair_written ->
    start_write_slots -> slot_written_pair!bp(pair_written) ->
    slot_written_slot!bs(slot_written) ->
    slot_written_val!val ->
    end_write_slots -> LB_write_pair.get?pair_written ->
    LB_write_slot.get?slot_written ->
    writers_slots.bp(pair_written).sw!slot_written ->
    writers_slots.bp(pair_written).ew ->
    LB_write_pair.get?pair_written ->
    latest.sw!pair_written -> latest.ew -> end_write ->
    Fourslot_Writer_LB
```

```
Writer_LB1 =
    Fourslot_Writer_LB [| {| LB_write_pair, LB_write_slot |} |]
          the_writers_local_bits1
              \ {| LB_write_pair, LB_write_slot |}

Writer_LB2 =
    Fourslot_Writer_LB [| {| LB_write_pair, LB_write_slot |} |]
          the_writers_local_bits2
              \ {| LB_write_pair, LB_write_slot |}

Fourslot_Reader_LB =
    start_read -> latest.sr -> latest.er?read_pair ->
    LB_read_pair.set!read_pair -> LB_read_pair.get?read_pair ->
    reading.sw!read_pair -> reading.ew ->
    LB_read_pair.get?read_pair ->
    writers_slots.bp(read_pair).sr ->
    writers_slots.bp(read_pair).er?read_slot ->
    LB_read_slot.set!read_slot -> LB_read_slot.get?read_slot ->
    LB_read_pair.get?read_pair -> start_read_slots ->
    read_slot_pair!bp(read_pair) ->
    read_slot_slot!bs(read_slot) ->
    read_slot_val?val -> end_read_slots -> end_read!val ->
    Fourslot_Reader_LB

Reader_LB1 =
    Fourslot_Reader_LB [| {| LB_read_pair, LB_read_slot |} |]
         the_readers_local_bits1
             \ {| LB_read_pair, LB_read_slot |}

Reader_LB2 =
    Fourslot_Reader_LB [| {| LB_read_pair, LB_read_slot |} |]
         the_readers_local_bits2
             \ {| LB_read_pair, LB_read_slot |}
```

The definitions of the *bp*, *bs* and *toggle* functions were given in Section 7.3. These definitions include the extra *get* events, that allow metastable values to settle to stable ones, before each use of the value stored by the local bit variables.

It should be noted that these models assume that the value being used to access a slot will not change while the slot is being accessed. This assumption may not hold in some implementations when a metastable value is being used

to index into the slots. In some hardware implementations it is possible that the reader (or writer) may interpret a metastable value as a one, and the value may then settle to a zero while the reader (writer) is accessing the slot. In such cases the reader (writer) may start to access one slot and then change to another slot in the ACM during the read (write). This assumption does not, however, effect the results of the analysis given in Section 7.5, since the 4-slot fails all of the model checking tests when metastable values fail to resolve before they are used.

7.5 Model Checking the 4-slot ACM using CSP and FDR

The models of the 4-slot have been model checked for the following properties:

Data-coherence: by checking if the model refines an *Incoherence-Spec*, which engages in a *clash-bang* event and stops if the reader and writer access the same slot a the same time. These results were confirmed by model checking against a semi-regular process (*SemiRegularACM*), which records the complete set of values that have been written and ensures that the reader only returns one of those values as the result of a read.

Local Freshness: by checking the model against a specification of a regular ACM (*RegACM*). This specification creates a set which contains the value written before a read starts and the values written while the read is in progress: the reader then returns one of those values as the result of a read.

L-atomicity: by checking that the model refines a *monotonic-Spec* which writes a monotonically increasing sequence of values to the ACM and produces a visible *order-bang* event if the reader reads the items out of order. This combined with a check for local freshness is sufficient to verify the model is L-atomic. Alternatively it is possible to check if the ACM behaves like a Hoare-atomic variable with an asynchronous writer and this specification is shown below. In the definitions that follow *wr_op* and *rd_op* represent complete reads and writes of items of data to the *ACM*.

```
H_Atomic_Var(var_name, val) =
     var_name.wr_op?x -> H_Atomic_Var(var_name, x) []
     var_name.rd_op!val -> H_Atomic_Var(var_name, val)
```

```
Read = start_read -> pool.rd_op?val -> end_read!val -> Read

Write = start_write?val -> pool.wr_op!val -> end_write -> Write

H_Atomic_state = H_Atomic_Var(pool, 1)

LAtomicACM = (((Read ||| Write) [| {| pool |} |]
                  H_Atomic_State) \ {| pool |})
```

The results of model checking the 4-slot against the above specifications are given in Section 7.5.2.

7.5.1 Relationship Between the Specifications

Some validation of the specifications in this section has been achieved by showing that *SemiRegularACM* is trace refined by *RegACM*, which in turn is trace refined by *LAtomicACM*. In addition, by composing *LAtomicACM* with a writer that writes a monotonically increasing sequence of values, and a reader which fails if it does not read a weakly increasing monotonic set of values: it has been shown that a specification is L-atomic, when its reader does not fail.

7.5.2 Results and Analysis

This section presents the results of model-checking the various models of Simpson's 4-slot: a summary of the different models is given in Table 7.1, and Table 7.2 summarises the results[6].

The models have been checked for data-coherence by ensuring that they did not refine the *Incoherence_Spec* and that they did refine the specification of a semi-regular ACM. It can also be observed that the result of the check for global freshness (against the *H-atomic* specification above - BIT0) agrees with the conjunction of the checks for local freshness and sequencing, as expected.

These results confirm the following for the 4-slot:

1. The results presented in Chapter 6, that the 4-slot is L-atomic in implementations that guarantee that access to the control variables is Hoare-atomic.

[6]The interested reader can download the PVS theory, and proof scripts, from http://homepages.cs.ncl.ac.uk/neil.henderson/CSP/CSP.tgz.

Table 7.1: The Descriptions of the Different Bit Models

Model	Model Description
BIT0	H-atomic, mutually exclusive atomic access to each bit
BIT1	Type-safe. Allows arbitrary clashes. No model of metastability.
BIT2	As BIT1, except remains stable when over-written with same value
BIT3	As BIT2, except metastability causes arbitrary clock stretching
BIT4 LB1	As BIT2, except has metastable values, which may be re-read
BIT4 LB2	As BIT4 LB1, except metastable values cannot be re-read
BIT5 LB1	As BIT4 LB1, except timing constraints prevent multiple clashes
BIT5 LB2	As BIT4 LB2, except timing constraints prevent multiple clashes
BIT6 LB1	As BIT5 LB1, except flickers when over-written with same value
BIT6 LB2	As BIT5 LB2, except flickers when over-written with same value

2. The ACM is not L-atomic if it is implemented with control variables that behave in a type-safe manner (which was shown by Rushby in [Rus02]).

3. The ACM is L-atomic provided the value recorded by a bit variable does not flicker if it is overwritten with the same value; that the reader of a control variable executes sufficiently slowly to allow a metastable value to resolve to a valid one before it is used, and it is only possible for a read to clash with a single write. The *BIT6* models show that the 4-slot is not L-atomic if the values recorded by the control variable flicker if overwritten with the same value. If this flickering behaviour can occur the implementation needs to be changed so that the reader and writer keep copies of the last values written to the control variables. They can then compare the new value with the old one, and only write the value to the control variable if it is different from the previous one.

It can be seen, from Table 7.2, that there are quite different results from modelling the 4-slot in different ways: from those models that do not directly model the effects of metastability to those that model realistic implementations and the engineering solutions that are used to mitigate its effects. The 4-slot will not preserve data-coherence if it is implemented in such a way

Table 7.2: 4-Slot Coherence, Sequencing and Freshness Results

1987 4-Slot with all the control bits modelled as:	Data-Coherence (Semi-Regular)	L-Regular (Local Freshness)	Sequencing	L-Atomic (Global Freshness)
BIT0 (H-Atomic)	√	√	√	√
BIT1 (L-Safe)	√	×	×	×
BIT2	√	√	×	×
BIT3	√	√	×	×
BIT4 LB1	×	×	×	×
BIT4 LB2	√	√	×	×
BIT5 LB1	×	×	×	×
BIT5 LB2	√	√	√	√
BIT6 LB1	×	×	×	×
BIT6 LB2	√	×	×	×

that control variables can be re-read before a metastable value has resolved to a stable one, but it does preserve data-coherence if it is implemented with type-safe bit control variables. This seems to indicate that formal models of ACMs that assume the bit control variables act in a type-safe manner may incorrectly verify that those ACMs have certain properties, such as data coherence. Some ACMs that are assumed to be implemented with type-safe control variables are described briefly in Section 7.6, and the results described in this chapter may challenge some of the proofs of correctness of these ACMs.

7.6 Further Work

Much academic literature (for example [HS94, HV96, Tro89] has assumed that single bit ACMs are *type-safe*[Lam86b], when they are implemented in a fully asynchronous manner, even when metastability can occur. The work described in this chapter indicates that this may be a dangerous assumption. [HV01] describes a proof that shows it is impossible to realise a conflict-free write-once L-atomic ACM from only 4 buffers and 4 type-safe control variables[7]. This result applies to the 4-slot, and [Rus02] establishes that it fails to be L-atomic when implemented with 4 type-safe control bits. However, with models of control bits that are arguably more realistic than the type-safe model (because they take into account the effects of metastability and measures to contain those effects), the 4-slot is L-atomic.

Further joint work by the author, Paynter and Armstrong [PHA05] mod-

[7]It is a pity that such an important result is unpublished.

els the effects of metastability on a number of proposed ACM implementations. These implementations are more complex (and may be less efficient) than the 4-slot and are:

1. Tromp's Four Track ACM [Tro89], which uses 12 safe bits to implement its four control variables.

2. Tromp's Efficient Four Track ACM [Tro89], which uses 8 single bit control variables.

3. A 1-reader 1-writer ACM from [HS94], which uses 4 control variables like the 4-slot, but which writes the same item twice, to 2 different buffers, in some circumstances. This implementation may be particularly inefficient if a large data structure is being communicated.

4. An *atomic* ACM from [KKV87]. This particularly complex implementation uses three 13 valued and three 2 valued control variables, but (if we have modelled it correctly) fails to be L-atomic unless access to the control variables themselves are Hoare atomic.

It is not possible to conclude, from the results of this work, that one (correct) implementation is better than another in all circumstances. For example the implementation from [HS94] uses fewer control variables than those from [Tro89], but it is not a write-once implementation. The requirement to sometimes write a large data structure twice may have unacceptable performance implications. Simpson's 4-slot also uses 4 control variables, but the cost is that it is only L-atomic if the control variables can be implemented in a particular manner. In the large class of implementations where this is possible Simpson's implementation will be correct and may be the most efficient. All of the implementations have difficulties where they are executed so quickly that it is possible to re-read metastable values before they resolve into binary ones. However, all of the implementations, apart from Simpson's, exhibit a failure mode by which the use of a metastable value may affect the control flow of the algorithm. In some implementations, where Simpson's algorithm may fail, the Efficient 4-track from [Tro89] is the most efficient and most able to contain the effects metastability (of those considered). However, if the effects of metastability and the engineering solutions to mitigate them, are not modelled correctly, it is possible that more efficient ACM implementations may be abandoned in favour of less efficient ones, with more complex algorithms that are more difficult to verify to be correct against their specification.

7.7 Summary

The results demonstrate that it is important to model the possible effects of metastability carefully. In a single processor implementation of an ACM, where the individual actions of the reader and writer will be executed Hoare-atomically, the H-atomic variable (BIT0) may be an adequate model of all of its possible behaviours (as may the model in Chapter 6). In hardware implementations, however, where the reader and writer are truly asynchronous, and can clash on accessing control variables, metastability can occur when such a variable is read when a new value is being written to it. In such cases the effects of metastability, and any measures taken to contain them, should be taken into account in the models. If an abstract model is used that ignores metastability the results should be used with caution.

This chapter concludes the descriptions of the methods used to investigate and verify properties of ACMs. This thesis has shown how it is possible to verify properties of a complex asynchronous system, by starting with an abstract model of the required properties and verifying that increasingly realistic models of the implementation exhibit those required properties. In this way it is possible to gain an understanding of the behaviour of the system in an incremental manner, until sufficient confidence is gained in the correctness of a particular implementation. The results in this chapter show that it is very difficult to verify properties of fully asynchronous implementations. It is particularly important to ensure that the models of the system correctly take account of possible interference between the processes in order to be confident that the implementation will behave in the desired manner. The next chapter discusses the results, and conclusions, of the work in more detail.

Chapter 8

Conclusions

This thesis has described how it is possible to use formal models to explore properties of asynchronous systems. The main objectives for this work were: first, to reduce the amount of rework that is required in the later stages of the development process; and second, to develop a theory of communication mechanisms to be used with the RTN-SL to facilitate the analysis of end-to-end timing properties of systems.

A range of tools has been used to verify that an ACM implementation (Simpson's 4-slot) is L-atomic. Starting with an abstract model of the required specification, increasingly realistic models of the implementation have been built to its explore properties and better understand its behaviour. This increased understanding may help to eliminate errors and ambiguities in the specification and reduce the amount of rework that is required later in the development process. This chapter discusses the merits and disadvantages of the various tools. The work reported in this thesis may form the basis of an incremental development process, which may be used to develop a theory of (a wide range of) communication mechanisms.

The remainder of this chapter is organised as follows. First, Section 8.1 briefly reviews the results of attempts to use the taxonomy of ACMs from Chapter 2 as the basis of a theory of communication mechanisms. Section 8.2 discusses the merits of the tools that have been used to verify the 4-slot is L-atomic. Section 8.3 discusses the benefits of using a proof tool (PVS) to assist with the verification process. Section 8.4 introduces related work. Section 8.5 looks at possible future work, and Section 8.6 gives the final conclusions.

8.1 A Taxonomy of ACMs

Initial attempts to define a theory of ACMs were based on a taxonomy of ACMs. Lamport [Lam86b] introduces a taxonomy of ACMs and an ex-

tended taxonomy was given in Chapter 2 that includes formal definitions of additional useful types of ACM. The extended taxonomy includes formal definitions of the required properties of ACMs, including coherence and freshness.

The difficulty with using the taxonomy as the basis for a theory of communication mechanisms is that it defines the behaviour of the mechanisms in terms of complete reads and writes, and their behaviour when the reads and writes overlap with each other. In ACM implementations, such as the 4-slot, there is a crucial point within a write when the item written is released, which can vary from write to write depending on the recent history of interactions of the reader and writer of the ACM. Similarly there is a crucial point within a read when the reader acquires the item to be read, which is also dependent on the recent behaviour of the reader and writer. In addition the read and write actions are themselves implemented by a number of operations, and it is possible for an unbounded number of read operations to occur between any two write operations, and vice versa. It would be necessary to devise a set of proof rules to verify that the effect of a sequence of operations that comprise a read (write) in the implementation is equivalent to a read (write) in the definition despite interference from the writer (reader). This would be a very difficult task bearing in mind the individual reader operations in the implementation may interfere with writer, and vice versa. In order to define proof rules for the theory of communication mechanisms it was necessary to obtain a better understanding of the behaviour of implementations of those mechanisms, where the components can interfere with each other. As a first step a number of tools have been used to gain an understanding of the behaviour of an ACM implementation, Simpson's 4-slot, as described in the next section.

8.2 Verifying Properties of an ACM Implementation

This section describes the tools that have been used in this thesis to verify the correctness of an ACM implementation with respect to its specification. First an abstract model of L-atomicity was given. This model was easy to understand, and formally specified the required properties of the implementation. It was then possible to verify that the 4-slot implementation is a refinement of this specification, using Nipkow's retrieve rule, however this required an unrealistic assumption about the atomicity of the actions of the reader and writer of the ACM. In order to discharge the proof obligations some of the actions of the reader and writer in the implementation need to be combined into single actions, that are equivalent to the operations of

the abstract model, which are assumed to be executed in a Hoare-atomic manner. It is therefore recognised that this is not a full correctness proof for the ACM, because these groups of actions are not atomic in actual implementations of the 4-slot. A means of relaxing this atomicity assumption was therefore required, and a rely-guarantee proof method for shared variable concurrency was used for this purpose. This made it possible to verify that the implementation is L-atomic where the individual actions of the reader and writer are atomic, but can interleave in an unrestricted manner. Finally Chapter 7 described some related work that verifies properties of fully asynchronous implementations of the 4-slot, for example hardware implementations, where these individual actions may not be atomic.

8.2.1 Applying Refinement to Verify Properties of Systems

Verifying a refinement relation between the 4-slot implementation and the model of L-atomicity made it possible to explore some of the behaviours of the implementation, and helped in gaining an increased understanding of those behaviours. For example it identified that there are two points within the writer algorithm when the item that is being written can be released and made available to the reader. However, the notion of refinement requires that it is possible to reason about the equivalence of an action in an abstract model to an action (or sequence of actions) in a more concrete model of the implementation. In the case of the 4-slot it would be necessary to verify that a refinement relation exists when the action in the concrete model consists of a number of sequential sub-actions. This would be very difficult, because the individual sub-actions of the reader and writer either access control variables, or read or write to one of the slots. A number of writer actions interfere with the operation of the reader, and the *readerIndicatesPair* operation interferes with the operation of the writer. In addition it is possible for an unbounded number of writer actions to occur between any two reader actions, and vice versa. It is, therefore difficult to reason about the effect of a sequence of reader, or writer, sub-actions, and the equivalence of that effect to the result of a single action in the abstract model.

An incremental method, that uses refinement in the early stages of development, may make it possible to evaluate the risk of continuing with a particular approach to the implementation earlier in the process using an abstract model of the requirements, before incurring the cost of fully verifying the correctness of the proposed implementation to its specification. However, in order to verify the correctness of actual implementations of fully asynchronous systems, such as the 4-slot, where the individual operations of the component processes can occur concurrently or interleave with each

other in an unrestricted manner, it is necessary to reason about the potential interference of the components with each other. This is not possible using the current refinement rules, and requires the those rules to be extended to make it possible to reason about the effects of such interference.

8.2.2 Applying a Rely-Guarantee Proof Method

Chapter 6 described how a rely-guarantee method for interleaved shared variable concurrency, from [dR+01], can be used to verify that Simpson's 4-slot ACM implementation is L-atomic when the atomicity assumptions used in earlier models are relaxed. This method made it possible to verify that the 4-slot implementation is L-atomic when the individual actions of the reader and writer are themselves atomic, for example single processor implementations.

The use of this method makes it necessary to identify assertions that can be made in the different locations of the assertion networks of the component processes. The effort that was required to discover the assertions was outweighed by the advantages of the method. First, discharging the proof obligations helped to identify errors and ambiguities in the model. Second, it was possible to verify properties of infinite state space models using this method. Third, the guarantee conditions that have been verified to hold for the implementation can be used in compositional proofs of the correctness of systems where the ACM is used as a component and its rely-conditions hold.

In addition, discovering the assertions that hold in the locations of the assertion networks for the components and discharging the proof obligations helped in gaining a better understanding of the behaviour of the implementation. For example, it helped identify the different points in the interaction of the reader and writer when the reader can effectively acquire the slot (and therefore the item) it is going to return as the result of a read.

It may be possible to extend the models to verify properties of fully asynchronous implementations, but this way may make the assertions significantly more complex and the proofs may then be daunting, if not intractable.

8.2.3 Model Checking Using CSP

Chapter 7 described some joint work, by the author, Paynter and Armstrong [PHA04], where an increasingly sophisticated set of models (due to Paynter), in CSP, were used to model fully asynchronous implementations of the 4-slot. The advantage of using CSP is that it was possible to encode the 4-slot algorithm into the model and then adjust the behaviour of the control

variables to take account of the effects of metastability and the engineering solutions to contain its effects. However, while it is possible to make subtle changes to the models to explore their properties, and a model checker will provide a counter example if a property fails to exhibit a particular property, less insight is gained when model checking a particular property succeeds. While it may be possible to use a model checker more directly to verify properties of systems, the increased understanding of the possible behaviour of the system, gained from exploring properties of the earlier models, helped to give increased confidence in the results of model checking those models. Building an increased understanding of the behaviour of a system under development may help to avoid the use of an inappropriate abstraction, and give increased confidence in the correctness of any results obtained from verifying properties of the system using model checking techniques. In addition it is only possible to model check finite state models: there is always a danger that an abstraction that is used to reduce an infinite state space model to a finite state one will incorrectly hide the very behaviour that would make the system violate the property that is being checked.

8.3 Machine Assisted Formal Proofs

The formal proofs described in this thesis were completed using PVS. The use of a proof tool to verify properties of models was valuable for a number of reasons. First, it helped to facilitate the use of an evolutionary process: it was possible to verify properties of partial models, which included some of the desired properties. These partial models could then be extended more easily, and the existing partial proofs extended to verify properties of the system as the models evolved, until finally the complete models were verified to be correct. Second, the tool helped to identify any errors in the models, for example when it was not possible to discharge a particular branch of the proof. Section 6.4 briefly described how it was also possible to identify a defect in a 3-slot ACM implementation in this way.

However, care needs to be exercised when using the decision procedures of PVS. It is important to work out the expected tactics for discharging a proof in advance. If a theory is discharged unexpectedly this may be due to a typographic error which introduces a contradiction in the assumptions. While it may be possible to discharge proof obligations more quickly by using the more powerful decision procedures, it may then be more difficult to identify such errors.

8.4 Related Work

Communication often seems to be assumed to occur instantaneously when modelling systems, for example in timing diagrams in UML [Dou98]. Little evidence has been found of an attempt to define a theory of communication mechanisms that can be used to reason about the timing of different communication mechanisms when specifying systems. While there are tools available to model the timing of systems, for example Uppaal [LPY97] and Kronos [Yov97], they have been used to verify properties of individual implementations, for example [DY95], rather than to develop a theory of communication mechanisms.

[Sim03] gives axiomatic formal definitions of the timing behaviour of a family of asynchronous and synchronous communication mechanisms using a VDM-SL [ISO96] like notation combined with RTL [JM86, JMS88]. This work makes a valuable contribution towards defining a theory of communication mechanisms. However it is based around crucial release and acquire events rather than the compositional behaviour of the components of the ACMs. It may be difficult to use the definitions directly to reason, in a compositional manner, about the behaviour of larger systems that use the mechanisms defined for communication between their components.

It is not common to take account of metastability when constructing formal models of ACMs [KKV87] and in much of the related academic work which explores implementations of L-atomic ACMs, for example [HV01, HS94, Tro89] it is assumed that single bit variables are type-safe. However, Chapter 7 showed that a type-safeness may not be an adequate representation of the behaviour of control variables in ACMs.

[Sim92] describes a *Role Modelling Method* that can be used to explore the behaviour, and verify properties, of ACM implementations. This method applies *roles* to the slots in the ACM, for example to indicate which of the slots is being read by the reader. The method then automatically explores the state space of the implementation, but combines the states into equivalent ones, based on the roles allocated to the slots and critical actions of the reader and writer. This novel approach reduces the state space of the model, and has been used to explore and verify properties of implementations. The results need to be carefully analysed, but the method can be used to gain a better understanding of the behaviour of implementations. [Sim92] uses role modelling to analyse the possible faulty behaviour of the 3-slot implementation described in this thesis, and to identify the timing constraint that is necessary to ensure fault free operation. [XC99] and [XC00] demonstrate that Simpson's roles can be encoded into Petri-net models as a means of increasing confidence in the results obtained. The method does, however, rely on an assumption about the behaviour of the reader of a control variable

when a read clashes with a write: that the read will either return the value in the control variable before or after the write.

Clark and Xia, [Cla00, Xia00], have also modelled the behaviour of the 4-slot in the presence of metastability using Petri-nets, and have shown the ACM to be L-atomic. Their approach is to model the set up and hold times and propagation delays (as in the $BIT5\_LB2$ model in Chapter 7), and the results they report seem to agree with those shown in Table 7.2. In [Cla00] it is stated that the 4-slot fails to maintain coherence and freshness when the writer's local copy of the pair it is going to access goes metastable. This difference in the results may be because Clark was modelling a hardware variant of the 4-slot, where this variable is used twice in the same instruction to access a slot, immediately after a new value has been written to it. The use of Petri-nets may be more suited to verifying properties of hardware implementations, because there is tool support to derive a hardware design directly from the model of the system.

[Bro99] uses Timed CSP to model the behaviour of the 4-slot, and describes 4 attempts to define freshness in terms of the beginning and end of complete reads and writes to the mechanism. The 4-slot can fail to return fresh data according to all of these definitions, but this failure is due to the reader returning data that is *too fresh* i.e. the reader returns an item that has been written but not fully released by the writer. It seems reasonable to follow the conclusion of this work, that freshness was not adequately defined, rather than conclude that the ACM does not exhibit the desired property.

8.5 Future Work

8.5.1 An Incremental Development Method

It may be possible to use the approach described in this thesis as the basis for an incremental development method for a wider range of systems. The use of a tool set rather than a single tool to exploit the relative strengths of particular tools may be advantageous, using a range of modelling techniques to explore properties of the implementation. For example an iterative approach that initially uses refinement to verify properties of an implementation to an abstract model, and later uses model checking to verify correctness of a fully asynchronous implementation. This may require the development of proof rules to verify the equivalence of different models of the implementation. An incremental development method may assist in making a more informed choice between competing implementations, or analysing the risk in proceeding with a particular approach to the implementation, at an earlier stage of the development process. Combining this iterative approach with a hierarchical development process may make it possible to analyse the

behaviour of an implementation and its component processes in increasing detail until sufficient confidence is gained in the correctness of a particular implementation to its specification.

8.5.2 Developing a Theory of Communication Mechanisms

The RTN-SL is currently being developed to allow the rigorous specification of the functionality and timing constraints of computations in systems. A theory of communication mechanisms is required, which can be used with the RTN-SL to specify the complete behaviour of systems in a rigorous manner. This theory will need to encompass a wide range of synchronous and asynchronous mechanisms, for example the basic mechanisms described in Section 1.1.2, and mechanisms that are implemented using networks of components.

This thesis has described how a number of tools have been used to verify that a particular (small) ACM implementation, Simpson's 4-slot, is L-atomic, however the development of a theory of communication leads to a number of requirements. First, the notations will be required to reason about the timing behaviour of the mechanisms. Second, a proof theory will need to be developed to verify properties of those mechanisms. It is desirable that this proof theory should facilitate a compositional approach (for example rely-guarantee [Jon83] or "Design by Contract" [Mey88]) to assist with the verification, and upgrading of, systems where the mechanisms are used as components.

8.5.3 Tool support

Any incremental process that is used for system development must be cost effective: the extra cost of analysing properties of the system earlier in the development cycle must be recovered by a reduction in the cost in the later stages. Tool support for the process will help to achieve this goal. The tool(s) should be capable of automatically translating the specification and design of the system from a graphical design notation into the formal language(s) that is (are) used to analyse properties of the system. It may be possible to develop a set of tactics that can be used with a proof tool to automatically discharge a proportion of the proof obligations. The ability to reason about trade-offs in the design would also be beneficial. This may allow a choice to be made earlier in the development process and avoid abandoning an inappropriate design later in the development process. The utility and cost effectiveness of any tool set would need to be measured, using an appropriate case study.

8.5.4 Atomicity Refinement

It was necessary to use an unrealistic assumption about the atomicity of the actions of reader and writer in the 4-slot implementation, in order to verify that the implementation is a refinement of the model of L-atomicity. It may be possible to develop an "atomicity refinement" method verify a refinement relation exists between two models when a single action is replaced with a series of actions in implementations such as the 4-slot, where it is necessary to reason about possible interference of actions of another process in the system with the effect of the refined action.

8.5.5 Identifying and Verifying New Impossibility Results for ACM Implementations

Recent work [PHA05] casts doubt on the validity of some current impossibility results for atomic ACMs, since these results seem to be based on an assumption that the control variables of the mechanisms behave in a type-safe manner. It seems that new impossibility results need to be produced and verified based on more realistic definitions of the behaviour of the components of the mechanisms. In order to derive these new results it will be necessary to reason about the possible effects of metastability on components, and the solutions that can be adopted to contain those effects.

8.5.6 Verifying Properties of Fully Asynchronous Systems Using Rely-Guarantee

The rely-guarantee approach in this thesis can be used to verify properties of systems that communicate using shared variables, where the individual actions of the components are atomic and can interleave in an unconstrained manner. It may be possible to extend this approach to verify properties of fully asynchronous systems. For example, by extending the assertion networks to include start and end actions for the individual operations of the components. It may then be possible to define assertions that hold in the locations of the revised networks, and use the existing proof rules to verify that the required rely-guarantee conditions still hold. This may make it possible to verify properties of larger systems than can currently be model checked because of the large (potentially infinite) state space of the models.

8.6 Concluding Remarks

The contribution of the work in this thesis is that it demonstrates how it is possible to verify the correctness of an asynchronous real-time system to its

specification. Specifically the work:

1. demonstrates how a tool set can be used to gain an understanding of the behaviour of the system in an incremental manner. Starting with an abstract model of the required properties of the system and exploring properties of increasingly detailed models of the implementation; and

2. shows that it is possible to use a compositional rely-guarantee method to verify properties of systems where the individual components give few or no guarantees about their individual behaviour. It may then be possible to use the rely-guarantee conditions that have been verified to hold, to explore and verify properties of larger systems, where the system is itself used as a component. Rely-conditions can also be used to record assumptions about the system and its environment to ensure that they are not overlooked in the later stages of development.

Developing a realistic model of an asynchronous system may be difficult because its specification is complex, or because its components interact in unexpected ways. This thesis has shown how the strengths of a range of tools can be exploited to explore properties of an ACM implementation in an incremental manner. This incremental approach may allow the developer of a system to gain a better understanding of its behaviour by exploring properties of increasingly realistic models of its implementation, until sufficient confidence is gained in the correctness of that implementation against the specification. The extra effort that may be required in the earlier stages of development may be recovered by helping to reduce costs in later stages of the process due to errors and ambiguities in the specification.

Bibliography

[ABM98] S. Angerholm, J. Bicarregui, and S. Maharaj. On the Verification of VDM Specifications and Refinement with PVS. In J.C. Bicarregui, editor, *Proof in VDM: Case Studies*, FACIT. Springer, 1998.

[Acz83] P. Aczel. On an inference rule for parallel composition. Unpublished letter to Cliff Jones, March 1983.

[AG92] J.H. Anderson and M.G. Gouda. A criterion for atomicity. *Formal Aspects of Computing*, 4(3):273–298, 1992.

[Ash75] E. A. Ashcroft. Proving assertions about parallel programs. *JCSS*, 10:110–135, February 1975.

[Bac89] R.J. Back. Refining atomicity in parallel algorithms. In *PARLE Conference on Parallel Architectures and Languages Europe*. Springer-Verlag, June 1989.

[BFL+94] J.C. Bicarregui, J.S. Fitzgerald, P.A. Lindsay, R. Moore, and B. Ritchie. *Proof in VDM: A Practitioner's Guide*. FACIT. Springer-Verlag, 1994.

[BH95a] J.P. Bowen and M.G. Hinchey. Seven more myths of formal methods. *IEEE Software*, 12(4):34–41, July 1995.

[BH95b] J.P. Bowen and M.G. Hinchey. Ten commandments of formal methods. *IEEE Computer*, 28(4):56–63, April 1995.

[Bic98] J.C. Bicarregui, editor. *Proof in VDM: Case Studies*. FACIT. Springer, 1998.

[BP89a] James E. Burns and Gary L. Peterson. The Ambiguity of Choosing. In *Proceedings of 8^{th} Annual Symposium on Principles of Distributed Computing (PODC'89)*, pages 145–157. ACM Press, 1989.

[BP89b] J.E. Burns and G.L. Peterson. The ambiguity of choosing. In *Proceedings of the 8th Annual Symposium on Principles of Distributed Computing*, pages 145–157. Association for Computing Machinery, 1989.

[Bro99] P.J. Brooke. *A Timed Semantics for a Hierarchical Design Notation.* PhD thesis, Department of Computer Science, University of York, April 1999.

[BvW03] R.J.R. Back and J. von Wright. Compositional action system refinement. *Formal Aspects of Computing*, 15(2 and 3):103–117, November 2003.

[Cha87] Daniel M. Chapiro. Reliable High-Speed Arbitration and Synchronization. *IEEE Transactions on Computers*, 36(10):1251–1255, October 1987.

[Cla00] I.G. Clark. *A Unified Approach to the Study of Asynchronous Communication Mechansims in Real-Time Systems.* PhD thesis, London University, King's College, May 2000.

[CM73] Thomas J. Chaney and Charles E. Molnar. Anomalous Behavior of Synchronizer and Arbitor Circuits. *IEEE Transactions on Computers*, 22(4):421–422, April 1973.

[CXYD98] I.G. Clark, F. Xia, A.V. Yakovlev, and A.C. Davis. Petri Net Models of Latch Metastability. *Electronic Letters*, 34(7):635–636, 1998.

[Dij71] E.W Dijkstra. *Structured Programming*, chapter 1. Academic Press, 1971.

[Dij76] E.W. Dijkstra. *A Discipline of programming.* Prentice-Hall International, 1976.

[Dou98] Bruce Powel Douglas. *Real-Time UML: Developing Efficient Objects for Embedded Systems.* The Object Technology Series. Addison-Wesley, 1998.

[dR+01] Willem-Paul de Roever et al. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods.* Number 54 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2001.

[DY95] C. Daws and S. Yovine. Two examples of verification of multirate timed automata with kronos. In *Proc. 16th IEEE Real-Time Systems Symposium (RTSS'95)*, pages 66–75. IEEE Comp.Soc. Press, December 1995.

[Flo67] R W Floyd. Assigning meanings to programs. In *Proceedings AMS Symposium Applied Mathematics*, volume 19, pages 19–31. American Mathematical Society, 1967.

[FSE96] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement: The FDR 2.0 User Manual*, August 1996.

[FW96] S. Fowler and A.J. Wellings. Formal Analysis of a Real-Time Kernel Specification. In Bengt Jonsson and Joachim Parrow, editors, *Proceedings of the 4th International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems*, number 1135 in Lecture Notes in Computer Science. Springer, 1996.

[FW97] S. Fowler and A.J. Wellings. Formal Development of a Real-Time Kernel. In *Proceedings of the 18th IEEE Real-Time Systems Symposium - San Francisco*. IEEE, 1997.

[Gin03] Ran Ginosar. Fourteen Ways to Fool Your Synchronizer. In *Proceedings of ASYNC'03*, pages 89–95, 2003.

[Hal90] Anthony Hall. Seven Myths of Formal Methods. *IEEE Software*, ?(9):11–19, September 1990.

[HB99] M.G. Hinchey and J.P. Bowen, editors. *Industrial-Strength Formal Methods in Practice*. FACIT. Springer-Verlag, 1999.

[HEC89] Jens U. Horstmann, Hans W. Eichek, and Robert L. Coates. Metastability Behaviour of CMOS ASIC Flip-Flops in Theory and Test. *IEEE Journal of Solid-State Circuits*, 24(1):146–157, February 1989.

[Hen03] N Henderson. Proving the correctness of Simpson's 4-slot ACM using an assertional rely-guarantee method. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *Proceedings of the International Symposium of Formal Methods Europe, FME2003:Formal Methods*, number 2805 in LNCS, pages 244–263. Springer-Verlag, September 2003.

[Hoa69] C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580 + p. 583, 1969.

[Hoa72] C.A.R. Hoare. Proof of correctness of data representation. *Acta Informatica*, 1(4):271–281, 1972.

[Hoa74] C.A.R. Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 17(10):549–557, 1974.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[HP02a] N. Henderson and S.E. Paynter. The Formal Classification and Verification of Simpson's 4-Slot Asynchronous Communication Mechanism. Technical Report CS/TR/756, Centre for Software Reliability, Department of Computing, University of Newcastle, January 2002.

[HP02b] N. Henderson and S.E. Paynter. The Formal Classification and Verification of Simpson's 4-Slot Asynchronous Communication Mechanism. In L.-H. Eriksson and P.A. Lindsay, editors, *Proceedings of FME'02*, number 2391 in Lecture Notes in Computer Science, pages 350–369. Springer, 2002.

[HS94] S. Haldar and P.S. Subramanian. Space-optimum conflict-free construction of 1-writer 1-reader multivalued atomic variable. In *Proceedings of the Eighth International Workshop on Distributed Algorithms*, volume 857 of *Lecture Notes in Computing Science*, pages 116–129. Springer-Verlag, 1994.

[HV96] S. Haldar and K. Vidyasankar. Space-optimal buffer-based conflict-free constructions of 1-writer 1-reader multivalued atomic variables from safe bits. In *Proceedings of 15th ACM Symposium on the Principles of Distributed Computing.* ACM, 1996.

[HV01] S. Haldar and K. Vidyasankar. Space-Optimal Buffer-Based Conflict-Free Construction of 1-Writer 1-Reader Multivalued Atomic Variables from Safe Bits. Unpublished Paper, 2001.

[ISO96] ISO/IEC 13817-1. *VDM Specification Language, International Standard - Part 1: Base Language*, December 1996.

[JHJ89] Mark B. Josephs, C.A.R. Hoare, and He Jifeng. A Theory of Asynchronous Processes. Technical Report PRG-TR-6-89, Programming Research Group, Oxford University Computing Laboratory, February 1989.

[JIM87] Joint IECCA and MUF Committee on MASCOT (JIMCOM). *The Official Handbook of MASCOT: Version 3.1 - Issue 1*, June 1987. Crown Copyright.

[JLM88] F. Jahanian, R. Lee, and A.K. Mok. Semantics of Modechart in Real-Time Logic. In *IEEE Proceedings of the* 21^{st} *Annual Hawaiian International Conference on System Science*, 1988.

[JM86] F. Jahanian and A.K. Mok. Safety Analysis of Timing Properties in Real-Time Systems. *IEEE Transactions on Software Engineering*, 12(9):890–904, December 1986.

[JM94] F. Jahanian and A.K. Mok. Modechart: A Specification Language for Real-Time Systems. *IEEE Transactions on Software Engineering*, 20(12):933–947, 1994.

[JMS88] F. Jahanian, A.K. Mok, and D.A. Stuart. Formal Specification of Real-time Systems. Technical Report TR-88-25, Department of Computer Science - University of Texas at Austin, June 1988.

[Jon81] C.B. Jones. *Development Methods for Computing Programs Including a Notion of Interference.* PhD thesis, Oxford University Computing Laboratory, 1981.

[Jon83] C B Jones. Specification and design of (parallel) algorithms. *Information Processing Letters*, 9(83):321–331, 1983.

[Jon90] C.B. Jones. *Systematic Software Development Using VDM: Second Edition.* Prentice-Hall International Series in Computer Science, 1990.

[KBY02] David J. Kinniment, Alexandre Bystrov, and Alex V. Yakovlev. Synchronization Circuit Performance. *IEEE Journal of Solid-State Circuits*, 37(2):202–209, 2002.

[KKV87] L.M. Kirousis, E. Kranakis, and P.M.B. Vitányi. Atomic Multireader Register. In *Proceedings of the Workshop on Distributed Algorithms*, number 312 in Lecture Notes on Computer Science, pages 278–296. Springer, 1987.

[Kop90] H. Kopetz. *Software Engineer's Reference Book*, chapter 56. Number 0-750-61040-9. Butterworth-Heinemann Limited, 1990.

[Kop98] H Kopetz. *Real-time systems. Design principles for distributed embedded applications.* Kluwer Academic Publishers, 1998.

[Lam86a] L. Lamport. On Interprocess Communication - Part 1: Basic Formalism. *Distributed Computing*, 1:77–85, 1986.

[Lam86b] L. Lamport. On Interprocess Communication - Part 2: Algorithms. *Distributed Computing*, 1:86–101, 1986.

[LFB96] P.G. Larsen, J. Fitzgerald, and T. Brookes. Applying formal specification in industry. *IEEE Software*, 13(7):48–56, May 1996.

[LPY97] K.G. Larsen, P. Pettersson, and Wang Yi. Uppaal in a nutshell. *Journal of Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[LR93a] Patrick Lincoln and John Rushby. Formal verification of an algorithm for interactive consistency under a hybrid fault model. In Costas Courcoubetis, editor, *Computer-Aided Verification, CAV '93*, volume 697 of *Lecture Notes in Computer Science*, pages 292–304, Elounda, Greece, June/July 1993. Springer-Verlag.

[LR93b] Patrick Lincoln and John Rushby. A formally verifed algorithm for interactive consistency under a hybrid fault model. In *Fault Tolerant Computing Symposium 23*, pages 402–411, Toulouse, France, June 1993. IEEE Computer Society.

[LR94] Patrick Lincoln and John Rushby. Formal verification of an interactive consistency algorithm for the Draper FTP architecture under a hybrid fault model. In *COMPASS '94 (Proceedings of the Ninth Annual Conference on Computer Assurance)*, pages 107–120, Gaithersburg, MD, June 1994. IEEE Washington Section.

[Män88] Reinhard Männer. Metastable States in Asynchronous Digital Systems: Avoidable or Unavoidable? *Microelectronic Reliability*, 28(2):295–307, 1988.

[Mey88] Bertand Meyer. *Object-Oriented Software Construction.* Prentice-Hall International Series in Computer Science, 1988.

[MoD85] Ministry of Defence. *Modular Approach to Software Construction, Operation, and Test - MASCOT*, 1985. Defence Standard 00-17.

[MoD91] Ministry of Defence – Sea Systems Controllerate. *Requirements for Software for use with Digital Processors*, 1991. Naval Engineering Standard NES 620 – Issue 4.

[MP93] Z. Manna and A. Pnueli. Models of Reactivity. *Acta Informatica*, 30(7):609–678, 1993.

[MSJ96] A.K. Mok, D.A. Stuart, and F. Jahanian. Specification and Analysis of Real-Time Systems: Modechart Language and Toolset. In C. Heitmeyer and D. Mandrioli, editors, *Formal Methods for Real-Time Computing*, volume 5 of *Trends in Software*, chapter 2, pages 33–53. Wiley, 1996.

[Nip86] T. Nipkow. Non-Deterministic Data Types: Models and Implementations. *Acta Informatica*, 22:629–661, 1986.

[Nip87] T. Nipkow. *Behavioural Implementation Concepts for Nondeterministic Data Types.* PhD thesis, University of Manchester, May 1987.

[OG76a] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.

[OG76b] S. Owicki and D. Gries. Verifying properites of parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5):279–285, May 1976.

[OG76c] Susan Owicki and David Gries. An Axiomatic Proof Technique for Parallel Programs. *Acta Informatica*, 6:319–340, 1976.

[OSRSC99a] S. Owre, N. Shanker, J.M. Rushby, and D.W.J. Stringer-Calvert. PVS Language: Version 2.3. Technical report, Computer Science Laboratory - SRI International, September 1999.

[OSRSC99b] S. Owre, N. Shanker, J.M. Rushby, and D.W.J. Stringer-Calvert. PVS System Guide: Version 2.3. Technical report, Computer Science Laboratory - SRI International, September 1999.

[PAH00] S.E. Paynter, J.M. Armstrong, and J. Haveman. ADL: An Activity Description Language for Real-Time Networks. *Formal Aspects of Computing*, 12(2):120–144, 2000.

[Pay01] S.E. Paynter. Real-Time Logic Revisited. In José Nuno Oliveira and Pamela Zave, editors, *Proceedings of the International Symposium of Formal Methods Europe 2001: Formal Methods for Increasing Software Productivity*, number 2021 in Lecture Notes in Computer Science, pages 300–317. Springer, 2001.

[Pay02] S.E. Paynter. RTN-SL: The Real-Time Network Specification Language. Technical Report DR 20656, MBDA UK, March 2002. Issue 2.

[Per99] C Perrow. *Normal Accidents.* Princeton University Press, 1999.

[Pet83] G.L. Peterson. Concurrent reading while writing. *ACM Transactions on Programming Languages and Systems*, 5(1):46–55, January 1983.

[PHA04] S.E. Paynter, N. Henderson, and J.M. Armstrong. Ramifications of metastability in bit variables explored via simpson's 4-slot mechanism. *Formal Aspects of Computing*, 16(4):332–351, November 2004.

[PHA05] S.E. Paynter, N. Henderson, and J.M. Armstrong. Metastability in Asynchronous Wait-Free Protocols. *Accepted subject to revision by IEEE Transactions on Computers*, ?(?):?–?, 2005.

[RLKL95] B. Randell, J-C. Laprie, H. Kopetz, and B. Littlewood, editors. *Predictably Dependable Computer Systems.* Springer-Verlag, 1995.

[Ros98] A.W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall Series in Computer Science, 1998.

[Rus02] John Rushby. Model-Checking Simpson's Four-Slot Fully Asychronous Communication Mechanism. Technical Report Issued, Computer Science Laboratory – SRI International, July 2002.

[Sim86] H.R. Simpson. The MASCOT Method. *Software Engineering Journal*, 1(3):103–120, 1986.

[Sim90a] H.R. Simpson. Four-Slot Fully Asynchronous Communication Mechanism. *IEE Proceedings*, 137 Part E(1):17–30, January 1990.

[Sim90b] H.R. Simpson. Integrity Aspects of Real-Time Networks. In *Proceedings of the IEE Colloquium on MASCOT and Related Issues*, 1990.

[Sim90c] H.R. Simpson. MASCOT Real-Time Networks in Distributed System Design. In *Proceedings of the IEE Colloquium on MASCOT and Related Issues*, 1990.

[Sim92] H R Simpson. Correctness analysis for class of asynchronous communication mechanisms. *IEE Proceedings-E*, 139(1):35–49, January 1992.

[Sim94] H.R. Simpson. Architecture for Computer Based Systems. In *IEEE Workshop on the Engineering of Computer Based Systems*, Stockholm, May 1994.

[Sim96] H.R. Simpson. Layered Architecture(s): Principles and Practice in Concurrent and Distributed Systems. In *Proceedings of the 8^{th} IEEE Symposium on Parallel and Distributed Processing*, 1996.

[Sim97] H.R. Simpson. New Algorithms for Asynchronous Communication. *IEE Proceedings of Computers and Digital Techniques*, 144(4):227–231, July 1997.

[Sim03] H.R. Simpson. Protocols for Process Interaction. *IEE Proceedings on Computers and Digital Techniques*, 150(3):157–182, May 2003.

[Sim04] H.R. Simpson. Freshness Specification for a Class of Asynchronous Communication Mechanisms. *IEE Proceedings of Computers and Digital Techniques*, 151(2):110–118, March 2004.

[Tro89] J. Tromp. How to construct an atomic variable. In *Proceedings of the Third International Workshop on Distributed Algorithms*, number 392 in Lecture Notes in Computer Science, pages 292–302. Springer-Verlag, 1989.

[Whi01] R.G. White. ASRAAM: Software Requirements and Design, Specification and Test Strategy for the EPU Infrastructure Software (New EPU). Technical Report DL 21025, Matra BAe Dynamics, 2001.

[Wir71] Nicklaus Wirth. Program Development by Stepwise Refinement. *Communications of the ACM*, 14(4):221–227, 1971.

[XC99] F. Xia and I. Clark. Complementing the Role Model Method With Petri-Net Techniques in Studying Issues of Data Freshness of the Four-Slot Mechanism. Technical Report CS-TR-654, Department of Computer Science – University of Newcastle, January 1999.

[XC00] F. Xia and I. Clark. Complementing the Role Model Method With Petri Net Techniques in Studying Issues of Data Freshness of the Four-Slot Mechanism. In *Hardware Design and Petri-Nets*, pages 33–50. Kluwer Academic Publishers, 2000.

[Xia00] Fei Xia. *Supporting the MASCOT Method with Petri Net Techniques for Real-Time Systems Development.* PhD thesis, London University, King's College, January 2000.

[XYIS02] F. Xia, A.V. Yakovlev, I.G.Clark, and D. Shang. Data communication in systems with heterogeneous timing. *IEEE Micro*, 22(6), Nov-Dec 2002.

[Yov97] S. Yovine. KRONOS: A verification tool for real time systems. *International Journal of Software Tools for Technology Transfer*, 1(1 + 2):123–133, December 1997.

Appendix A

Translating from VDM-SL to the PVS Logic

The models in Chapters 2 to 6 of this thesis are given using a VDM-SL-like syntax. This appendix describes how the models have been translated into the PVS logic, using the encoding of VDM-SL operations from [ABM98]. Many of the translations are straightforward: for example a predicate using the universal quantifier:

$$\forall a : T_1; b : T_2 \cdot P(a, b) \tag{A.1}$$

becomes

$$\forall (a : T_1, b : T_2) : P(a, b) \tag{A.2}$$

Predicates defined using the existential quantifier are translated into PVS in a similar manner.

The remainder of this appendix illustrates how to translate from the VDM-SL like notation used in the body of this thesis to the PVS logic used to define the models in the rest of the appendices, using examples from the models.

The translation of enumeration types from VDM-SL to the PVS logic is straightforward. For example an enumeration type to represent the names of the two pairs in the model of Simpson's 4-slot in VDM-SL is:

$$PairIndex = p0 \mid p1;$$

which is given in the PVS logic as:

```
PairIndex: TYPE = {p0, p1}
```

The following basic VDM record:

$ACM :: baseType : Value\text{-set}$
$\quad validType : Value\text{-set}$
$\quad content : Time \xrightarrow{m} Value$

inv mk-$ACM\ (bT, vT, c) \triangleq (vT \in bT) \wedge (\mathsf{rng}\ c \in bT)$;

is translated to a record type in the PVS logic as follows:

```
ACM: NONEMPTY_TYPE =
[# base_type: A_Type,
   valid_type: {t: A_Type | ∀ (v: Value): (v ∈ t) ⇒ (v ∈ base_type)},
   content: {f: [Time → Value] | ∀ (t: Time): (f(t) ∈ base_type)} #]
```

where the beginning and end of the PVS record types are denoted by [# and #] respectively in the definition(a state model in VDM can be translated into a PVS *TYPE* in a similar manner). Instances of records are enclosed in (#*and*#) when they are introduced. The individual fields of the record in the PVS logic can be accessed in a similar manner to the field selector in VDM, for example given *acm*:*ACM* it is possible to access its base type using *acm*'*baseType* instead of *acm.baseType*. In the above example the invariant in the VDM model is encoded directly in the definitions of the types of the components of the record in the PVS equivalent. For example the first part of the invariant in the record is $vT \in bT$, which states that all of the items in the valid type must also be in the base type. This is translated into the PVS logic using a sub-type definition:

```
valid_type: {t: A_Type | ∀ (v: Value): (v ∈ t) ⇒ (v ∈ base_type)}.
```

Here the valid type is a set, composed of elements of type *A_Type*, which is defined separately, where all of the elements of the set are also in the base type set.

A further example of an invariant in VDM being encoded using a sub-type in the PVS logic, is from the abstract model of L-atomicity, where the sequence of values must always have a length of at least 1, which is given in the PVS logic as:

```
Val_Sequence: TYPE = {fin_seq: finite_sequence[Data] | fin_seq'length ≥ 1}
```

In some cases the invariant in the VDM model is encoded in functions which are called in the sub-type definition in PVS:

$Persistent\_ACM :: b\_acm : Basic\_ACM$

inv mk-$Persistent\_ACM\ (acm) \triangleq$
$\quad write\_val\_prop2(acm) \wedge persistent\_acm1(acm) \wedge$
$\quad persistent\_acm2(acm) \wedge persistent\_acm3(acm)$;

which is given in the PVS logic (again using a sub-type definition) as:

```
Persistent_ACM: TYPE = {acm: Basic_ACM | write_val_prop2(acm) ∧ persistent_acm1(acm) ∧
                        persistent_acm2(acm) ∧ persistent_acm3(acm)}
```

The functions are defined separately, for example the following VDM-SL-like definition:

$$
\begin{array}{l}
write\_val\_prop2 : Basic\_ACM \to \mathbb{B} \\
write\_val\_prop2\,(a) \triangleq \\
\quad \mathsf{let}\ w = a.writer, \\
\qquad acm = a.acm\ \mathsf{in} \\
\quad \forall\, i : Occ; v : Value; t_1, t_2 : Time \cdot \\
\qquad\quad communicates(w, i, t_1, t_2, v, acm)\ \Rightarrow \\
\qquad\quad acm.content(t_2) = v;
\end{array}
$$

translates directly into the PVS logic, except that the writer and the ACM need to be introduced into the VDM function using a let statement, whereas in the PVS logic this is not necessary. This is because the PVS *basic_ACM* is defined as a sub-type of type *ACM*, whereas in the VDM it is defined as a record type which has a field of type *ACM* and a field of type *Writer*:

```
write_val_prop2(acm: Basic_ACM): bool =
    ∀ (w: W_Action, i: Occ, v: Value, t1, t2:
      Time): communicates(w, i, t1, t2, v, acm) ⇒
        acm`content(t2) = v
```

Explicit VDM functions are translated directly into the PVS logic in a similar manner to the above.

Implicit functions cannot be translated directly into the PVS logic, for example given the following type:

$$
\begin{array}{l}
\mathsf{state}\ Conc\_State\ \mathsf{of} \\
\quad pairWritten : PairIndex \\
\quad slotWritten : PairIndex \xrightarrow{m} SlotIndex \\
\quad pairReading : PairIndex \\
\quad slots : PairIndex \times SlotIndex \xrightarrow{m} Data \\
\quad nri : NextReadInstruction \\
\quad nwi : NextWriteInstruction \\
\quad writer : WriterState \\
\quad reader : ReaderState
\end{array}
$$

$$\begin{aligned}\textsf{init}\ s \triangleq s = \textsf{mk-}&\mathit{Conc\_State}\,(p0, p0 \xrightarrow{m} s0, p1 \xrightarrow{m} s0, p1,\\ &\{(p0, s0) \xrightarrow{m} \textsf{mk-}\mathit{Data}\,(1, \textsf{mk-token}\,(\text{"}\mathit{initVal}\text{"})),\\ &(p0, s1) \xrightarrow{m} \mathit{nil}, (p1, s0) \xrightarrow{m} \mathit{nil},\\ &(p1, s1) \xrightarrow{m} \mathit{nil}\}, \mathit{nri} = \mathit{rcp}, \mathit{nwi} = \mathit{wcp},\\ &\textsf{mk-}\mathit{WriterState}\,(p0, s0),\\ &\textsf{mk-}\mathit{ReaderState}\,(p1, s1))\end{aligned}$$

end

the following function

$$\begin{aligned}&\mathit{readerIndicatesPair}\,()\\ &\textsf{ext wr}\ \mathit{nri} : \mathit{nextReadInstruction}\\ &\quad\textsf{wr}\ \mathit{pairReading} : \mathit{PairIndex}\\ &\quad\textsf{rd}\ \mathit{reader.readerPair} : \mathit{PairIndex}\\ &\textsf{pre}\ \ \mathit{nri} = \mathit{rip}\\ &\textsf{post}\ \mathit{nri} = \mathit{rcs} \wedge \mathit{pairReading} = \overleftarrow{\mathit{reader.readerPair}};\end{aligned}$$

is translated into the PVS logic as:

```
pre_readerIndicatesPair(p: Conc_State): bool = p'nri = rip

post_readerIndicatesPair(p: (pre_readerIndicatesPair))(prot: Conc_State): bool =
    prot = p WITH [nri := rcs, pairReading := p'reader'readerPair]

readerIndicatesPair: [p: (pre_readerIndicatesPair) → (post_readerIndicatesPair(p))]
```

The VDM-SL pre-condition is translated into a predicate on the arguments of the implicit function, and the postcondition is a binary relation on an item which satisfies the pre-condition and the result of the implicit function. The function itself is defined as an uninterpreted constant using a *FUNCTION* type, which given an argument p that satisfies the pre-condition, returns a result that is related to p by the post-condition. In the above post-condition the statement $\mathit{prot} = p$ WITH $[\mathit{nri}: = \mathit{rcs}, \mathit{pairReading}: = p\text{'}\mathit{reader}\text{'}\mathit{readerPair}]$ returns a new state, *prot*, which is the concrete state passed in as a parameter, p, with the *nri* field modified to *ris* and the *pairReading* field modified to be equal to the value of the *readerPair* local variable of the reader in p (the *hooked* value of *reader.readerPair* in the VDM-SL-like definition).

The abstract model of L-atomicity uses the following function to append a new item, when it is written, to the head of the sequence of values:

```
(seq: Val_Sequence ∪ {d: Data}): Val_Sequence =
    (# length := 1, seq := (λ ·(x: below[1]): d) #) ∘ seq
```

This function first creates a new sequence, with the new item, of length 1:

```
(# length := 1, seq := (λ ·(x: below[1]): d) #)
```

and then uses the function o from the finite sequences type in the PVS library to concatenate the new sequence to the head of the existing one.

The *Conc_State* type given above can be initialised in PVS using the following function (which assigns initial values to each of the components, where the reader and writer components have similar initialisation functions defined to create the values r and w respectively):

```
init_prot(p: Conc_State, init_val: Val, w: WriterState, r: ReaderState): bool =
    LET w = w WITH [writerPair := p0, writerSlot := s0],
        r = r WITH [readerPair := p1, readerSlot := s1]
     IN p = p WITH [pairWritten := p0,
                    slotWritten := (λ ·(p0: PairIndex): s0),
                    pairReading := p1,
                    slots := (λ ·(p0: PairIndex, s0: SlotIndex): init_val),
                    nri := rcp, nwi := wcp, writer := w, reader := r]
```

Here the *lambda* functions are used to assign values to some of the variables, for example:

```
slotWritten := (λ ·(p0: PairIndex): s0)
```

assigns the value s_0 to $slotWritten(p_0)$.

The assertions in the various models are also translated into the PVS logic using lambda functions. For example the assertion from the location in the abstract model of L-atomicity, where the reader and writer are not accessing the mechanism is given in VDM-SL as:

$$
\begin{aligned}
&noReader\_writer\_Assertion \triangleq \\
&\quad indexRead \leq nextIndex\text{-}\mathsf{len}\ vals \wedge \\
&\quad firstIndex \leq nextIndex\text{-}\mathsf{len}\ vals \wedge \\
&\quad vals(1).index = nextIndex\text{-}1);
\end{aligned}
$$

which translates in the PVS logic to (the last conjunct in the following assertion is actually part of the invariant in the VDM-like definitions in Chapter 4):

```
noReader_noWriter_Assertion: [Abs_State → bool] =
    (λ ·(abs: Abs_State):
      abs'indexRead ≤ abs'nextIndex-abs'vals'length ∧
       abs'firstIndexAvailable ≤ abs'nextIndex-abs'vals'length ∧
        abs'vals(0)'index = abs'nextIndex-1 ∧
         (∀ (n: nat): n < abs'vals'length ∧ n > 0 ⇒
             abs'vals(n)'index = abs'nextIndex-(n + 1))
```

Here the values are all fields of the *Abs_State* passed into the lambda function as a parameter, called *abs*, and so are accessed using the field selector (e.g.abs'$indexRead$). The length of the sequence is also a field of the finite sequence *vals* in the PVS record and so the field selector is used to access it (i.e.abs'$vals$'$length$), whereas the len operator is used to access the length of a sequence in VDM. The use of the lambda functions makes it possible to use the name of the assertion when defining the proof obligation, for example:

```
vc_noReader_noWriter_start_read: THEOREM
  ∀ (as1, as2: Abs_State):
    pre_start_read(as1) ∧
     ¬ as1'writerAccess ∧ noReader_noWriter_Assertion(as1) ∧ as2 = start_read(as1) ⇒
     as2'readerAccess ∧
      ¬ as2'writerAccess ∧ reader_noWriter_Assertion(as2)
```

The assertion can then be expanded in line when discharging the proof.

Mappings in PVS are simply defined as functions so the VDM type $Time \xrightarrow{m} Value$ translates to $[Time\text{-} > Value]$ and given a mapping, m, and a time, $t : Time$, it is possible to obtain the relevant item from the range using $m(t)$. This is advantageous when the domain of the mapping is a *composite value*, for example the *slots* in the model of the 4-slot are accessed using a *pairIndex* and a *slotIndex* the mapping is defined in PVS as:

```
slots: [PairIndex, SlotIndex → Val]
```

and it is possible to call the mapping using $slots(p_1, s_1)$.

A finite sets type is provided in the PVS library, and is defined as a sub type of the set type, where sets are represented as predicates. The usual operators are available, but are defined as *prefix operations*. For example to test whether x is a member of set a it is necessary to writer $member(x, a)$.

The PVS models also use a pre-defined finite sequences type, and use a number of the pre-defined functions, for example: to the following shortens a sequence to contain only the head item, or the first two items respectively

```
vals := p'vals ^ (0, 0)

vals := p'vals ^ (0, 1)
```

A function is defined that adds an item (*newItem*) to the head of the sequence (called *vals*) - see Appendix E for the full definition of this function.

```
vals := (vals ∪ {newItem})
```

Appendix B

An embedding of RTL in the PVS Logic

This appendix gives the shallow embedding of Real-time Logic (RTL) [JM86, JMS88], due to Paynter, in the PVS logic that has been used in the model of the extended taxonomy of ACMs in Chapter 2.

The definition is declared as a theory (and can then be used in other definitions using an IMPORT command (e.g. IMPORTING RTL).

```
RTL: THEORY
 BEGIN
```

Type definitions. A non-empty type of events, time (which is represented by real number, occurrences which are of type natural number, actions and states, which are non-empty types. The use of non-empty types is necessary to prevent a *Type Correctness Condition* (TCC) proof obligation being generated by PVS to verify that an element of the type exists (effectively the non-empty type definition is makes it an axiom of the model that elements of the type exist).

```
  Event: NONEMPTY_TYPE

  Time: TYPE = real

  Occ: TYPE = nat

  Action: NONEMPTY_TYPE

  State: NONEMPTY_TYPE
```

The RTL Theta (total) relation (from [JMS88]), which takes an event, an occurrence number and a time and returns a boolean, the value of which depends on whether the particular occurrence of the event occurred at the particular time.

```
  th: [Event, Occ, Time → bool]
```

Definition of psi: a relation which returns a boolean depending on whether a particular event occurred at a particular time.

ψ: [Event, Time → bool]

Functions for returning the events relating to entering and leaving states and start and stop actions of events.

enter, leave: [State → Event]

start, stop: [Action → Event]

A function for composing two sequential actions into a single (composite) action.

compose: [Action, Action → Action]

A function to check if an event occurs (at any time).

occurs(e: Event): bool = ∃ (t: Time): $\psi(e, t)$

A function to find the last occurrence time of an event.

last_occurrence_time(e: Event, t: Time): bool =
$\psi(e, t) \wedge \neg$ (∃ (t_1: Time): $t_1 > t \wedge \psi(e, t_1)$)

There are only finite occurrences of an event if there is a time of the last occurrence of the event.

only_finite_occurrences(e: Event): bool = ∃ (t: Time): last_occurrence_time(e, t)

There are infinite occurrences of an event, if it occurs and there is not only a finite number of occurrences of the event.

infinite_occurrences(e: Event): bool = occurs(e) ∧ ¬ only_finite_occurrences(e)

All of the occurrences of an event are bounded by a time if all of the occurrences occur before that time (potentially this is *zeno* behaviour. It is only necessary to reason about zeno behaviour where the version of RTL from [Pay01] that uses real numbers for time steps is used. The models in this thesis use finite time steps, but the definitions are included for information).

bounded_by(e: Event, t: Time): bool = ∀ (t_1: Time): $\psi(e, t_1) \Rightarrow t_1 < t$

is_bounded(e: Event): bool = ∃ (t: Time): bounded_by(e, t)

An occurrence of an event occurs at a unique time. .

RTLax1: AXIOM
∀ (e: Event, i: Occ, t_1, t_2: Time):
$\Theta(e, i, t_1) \wedge \Theta(e, i, t_2) \Rightarrow t_1 = t_2$

If an event has occurred for the $i + 1th$ time, the ith occurrence must have occurred at an earlier time.

RTLax2: AXIOM
  ∀ (e: Event, i: Occ, t_1: Time):
    $\Theta(e, i+1, t_1) \Rightarrow$ (∃ (t_2: Time): $\Theta(e, i, t_2) \wedge t_2 < t_1$)

Each event that occurs must have an occurrence number.

RTLax3_4: AXIOM
  ∀ (e: Event, t: Time): $\psi(e, t) \Leftrightarrow$ (∃ (i: Occ): $\Theta(e, i, t)$)

If there are infinite occurrences of an event, there must not be an upper bound for the times of those events (disallows zeno behaviour).

RTLax5: AXIOM ∀ (e: Event): infinite_occurrences(e) ⇒ ¬ is_bounded(e)

Start and stop actions relate to unique events.

Action1: AXIOM
  ∀ (a_1, a_2: Action):
    (stop(a_1) = stop(a_2) ⇒ $a_1 = a_2$) ∧
     (start(a_1) = start(a_2) ⇒ $a_1 = a_2$)

Start and stop events of actions are different events.

Action2: AXIOM
  ∀ (a_1, a_2: Action): stop(a_1) ≠ start(a_2) ∧ start(a_1) ≠ stop(a_2)

If the stop event of an action has occurred the action must have started at an earlier time.

Action3: AXIOM
  ∀ (a: Action, i: Occ, t_1: Time):
    Θ(stop(a), i, t_1) ⇒
     (∃ (t_2: Time): Θ(start(a), i, t_2) ∧ $t_2 \leq t_1$)

The start event of the (i + 1)th occurrence of an action is after the stop event of the ith occurrence.

Action4: AXIOM
  ∀ (a: Action, i: Occ, t_1: Time):
    Θ(start(a), i + 1, t_1) ⇒
     (∃ (t_2: Time): Θ(stop(a), i, t_2) ∧ $t_2 \leq t_1$)

Enter and leave events relate to unique states in the model.

State1: AXIOM
  ∀ (s_1, s_2: State):
    (leave(s_1) = leave(s_2) ⇒ $s_1 = s_2$) ∧
     (enter(s_1) = enter(s_2) ⇒ $s_1 = s_2$)

Leave and enter events for states are distinct.

State2: AXIOM
  ∀ (s_1, s_2: State): leave(s_1) ≠ enter(s_2) ∧ enter(s_1) ≠ leave(s_2)

If a leave event of a state occurs there must be an earlier enter event.

```
State3: AXIOM
  ∀ (s: State, i: Occ, t1: Time):
    Θ(leave(s), i, t1) ⇒
     (∃ (t2: Time): Θ(enter(s), i, t2) ∧ t2 ≤ t1)
```

In order to enter a state for the (i + 1)th time the state must have been exited for the ith time.

```
State4: AXIOM
  ∀ (s: State, i: Occ, t1: Time):
    Θ(enter(s), i + 1, t1) ⇒
     (∃ (t2: Time): Θ(leave(s), i, t2) ∧ t2 ≤ t1)
```

In order to compose two actions they must have both occurred.

```
compose: AXIOM
  ∀ (a, b, c: Action):
    a = compose(b, c) ⇒
     (∀ (i: Occ, t: Time): Θ(start(a), i, t) ⇔ Θ(start(b), i, t)) ∧
      (∀ (i: Occ, t: Time): Θ(stop(a), i, t) ⇔ Θ(stop(c), i, t)) ∧
       (∀ (i: Occ, t: Time):
          Θ(stop(b), i, t) ⇔ Θ(start(c), i, t))
```

The last occurrence of an event occurred at time of the last occurrence of that event.

```
last_occurrence_number(e: Event, i: Occ): bool =
    ∃ (t: Time): Θ(e, i, t) ∧ last_occurrence_time(e, t)
```

The latest occurrence of an event is at the latest time that the event occurred.

```
latest_occurrence_at_time(e: Event, i: Occ, t: Time): bool =
    ∃ (t1: Time):
      t1 ≤ t ∧
       Θ(e, i, t1) ∧ ¬ (∃ (t2: Time): t1 < t2 ∧ t2 ≤ t ∧ ψ(e, t2))
```

Either there have been no occurrences of an event or the number of occurrences is one greater than the last occurrence number (the first occurrence is numbered zero and occurs when the model is initialised, which is before the start time of the system being modelled).

```
no_of_occurrences_to_time(e: Event, t: Time, n: nat): bool =
    (occurs(e) ∧ n > 0 ∧ latest_occurrence_at_time(e, n-1, t)) ∨
     (¬ occurs(e) ∧ n = 0)
```

Function for checking which is the latest of two times.

```
latest(t1, t2, t3: Time): bool =
    (t2 ≥ t3 ⇒ t1 = t2) ∧ (t3 ≥ t2 ⇒ t1 = t3)
```

Functions for checking the latest of three times.

```
after_both(t1, t2, t3: Time): bool = t1 ≥ t2 ∧ t1 ≥ t3

strictly_after_both(t1, t2, t3: Time): bool = t1 > t2 ∧ t1 > t3
```

Definition of a periodic event.

```
periodic(e: Event, period: Time): bool =
    ∃ (t: Time):
      Θ(e,0,t) ∧
       (∀ (i: Occ, t1: Time):
          Θ(e,i,t1) ⇒ Θ(e,i+1,t1+period))
```

Definition of a sporadic event.

```
sporadic(e: Event, miat: Time): bool =
    ∀ (i: Occ, t1, t2: Time):
      Θ(e,i,t1) ∧ Θ(e,i+1,t2) ⇒ t2 ≥ t1+miat
```

Definition of a deadline.

```
deadline(e1, e2: Event, l, u: Time): bool =
    ∀ (i: Occ, t1: Time):
      Θ(e1,i,t1) ⇒
       (∃ (t2: Time):
          Θ(e2,i,t2) ∧ t1+u ≥ t2 ∧ t2 ≥ t1+l)
```

Definition of a window of time (all of the *clock ticks* between two times).

```
window(e, clk: Event, l, u: Time): bool =
    ∀ (i: Occ, t1: Time):
      Θ(e,i,t1) ⇒
       (∃ (j: Occ, t2: Time):
          Θ(clk,j,t2) ∧ t2+u ≥ t1 ∧ t1 ≥ t2+l)
```

Definition of jitter.

```
jitter(e, clk: Event, x1, x2, offset: Time): bool =
    ∀ (i: Occ, t2: Time):
      Θ(clk,i,t2) ⇒
       (∃ (t1: Time):
          Θ(e,i,t1) ∧
           t2+offset+x2 ≥ t1 ∧
            t1 ≥ t2+offset-x1)
```

Definition of consecutive occurrence bounds (lower and upper bounds on the time between two occurrences of an event).

```
COB(e: Event, max, min: Time): bool =
    ∃ (t: Time):
      Θ(e,0,t) ∧
       (∀ (i: Occ, t1: Time):
          Θ(e,i,t1) ⇒
           (∃ (t2: Time):
              Θ(e,i+1,t1) ∧
               t1+max ≥ t2 ∧ t2 ≥ t1+min))
```

Alternative definition of Axiom 2 - all occurrences of an event after the first must be preceded by the previous occurrence.

```
RTLax2_Alt: THEOREM
  ∀ (e: Event, i: Occ, t1: Time):
    Θ(e,i,t1) ∧ i > 0 ⇒
     (∃ (t2: Time): Θ(e,i-1,t2) ∧ t2 < t1)
```

Alternative definitions of Axiom 5 - non-zeno behaviour.

```
RTLax5_Alt: THEOREM
  ∀ (e: Event):
    ¬ occurs(e) ∨ only_finite_occurrences(e) ∨ ¬ is_bounded(e)

RTLax5_Alt2: THEOREM
  ∀ (e: Event):
    (¬ (∃ (t1: Time): ψ(e, t1))) ∨
     (∃ (t2: Time): ψ(e, t2) ∧ ¬ (∃ (t3: Time): ψ(e, t3) ∧ t3 > t2)) ∨
      (¬ (∃ (t4: Time): ∀ (t5: Time): ψ(e, t5) ⇒ t5 < t4))
```

The stop event of an action cannot be the same as the start event of an action, neither can the start event of an action be the same as the stop event of an action.

```
Action2_Alt: THEOREM
  ∀ (a1: Action):
    ¬ (∃ (a2: Action): stop(a1) = start(a2) ∨ start(a1) = stop(a2))
```

Earlier occurrences of events occur at earlier times.

```
mt: THEOREM
  ∀ (e: Event, i, j: Occ, t1: Time, t2: Time):
    (Θ(e,i,t1) ∧ Θ(e,j,t2) ∧ i < j) ⇒ t1 < t2
```

Two distinct occurrences of the same event cannot happen at the same time.

```
mo: THEOREM
  ∀ (e: Event, i, j: Occ, t: Time):
    Θ(e,i,t) ∧ Θ(e,j,t) ⇒ i = j
```

Previous occurrences of events occur at earlier times.

```
prev: THEOREM
  ∀ (e: Event, i: Occ, t1: Time):
    (Θ(e,i,t1) ∧ i > 0) ⇒
     (∀ (j: Occ): j < i ⇒ (∃ (t2: Time): Θ(e,j,t2) ∧ t2 < t1))

prev2: THEOREM
  ∀ (e: Event, i: Occ, j: Occ, t1: Time, t2: Time):
    (Θ(e,i,t1) ∧ Θ(e,j,t2) ∧ t1 < t2) ⇒ i < j

prev3: THEOREM
  ∀ (e: Event, i: Occ, j: Occ, t1: Time, t2: Time):
    (Θ(e,i,t1) ∧ Θ(e,j,t2) ∧ t1 ≤ t2) ⇒ i ≤ j

prev4: THEOREM
  ∀ (e: Event, i, j: Occ, t1: Time):
    Θ(e,j,t1) ∧ j ≥ i ⇒ (∃ (t2: Time): Θ(e,i,t2) ∧ t2 ≤ t1)
```

If a later action has started earlier occurrences of the action must have stopped.

```
Action_Prev: THEOREM
  ∀ (a: Action, i, j: Occ, t1: Time):
    Θ(start(a),i,t1) ∧ i > j ⇒
     (∃ (t2: Time): Θ(stop(a),j,t2) ∧ t2 ≤ t1)
END RTL
```

Appendix C

A Taxomony of ACMs

This appendix formally describes an extended taxonomy of ACMs, based on the taxonomy from [Lam86a], which gave (formal) definitions of (type-)safe, regular and atomic ACMs. The extended taxonomy described here includes other useful types of ACM, such as the persistent type that is used to implement the buffers in many atomic ACM implementations e.g those from [Tro89], [Sim90a] and [HS94].

First a number of basic definitions are given, including a definition of a basic ACM, which can best be described as faulty. The writer to a basic ACM writes valid values to it, but there is no guarantee that the ACM will either contain the valid written at the end of a write, or communicate a valid value to any reader. ACMs that give successively increasing guarantees about their behaviour are built out of this basic type in a hierarchical manner, with the final definition being that of an atomic ACM, which guarantees to communicate coherent and fresh data items (as defined in Section 2.5) from the writer to the reader.

```
General_ACMs: THEORY
 BEGIN

  IMPORTING RTL
```

The start time of the system and the number of occurrences of events are both positive natural numbers.

```
  Start_Time: posnat

  NOcc: TYPE = posnat
```

A type to define any time after system start up.

```
  NTime: TYPE = {t: Time | t ≥ Start_Time}
```

A non-empty type to represent values that can be transmitted by an ACM.

```
  Value: NONEMPTY_TYPE
```

The base type of values that can be transmitted by an ACM - essentially the set of values that can be represented by different bit representations of the ACM registers that store the values to be transmitted. For example an 8 bit register can potentially store 256 different values.

```
A_Type: NONEMPTY_TYPE = {t: setof[Value] | nonempty?(t)}
```

A general ACM has a base type (all of the possible values it can store (represent) for transmission to its reader); a valid type, which consists of all of the user defined values that are to be communicated by it that is a (potentially proper) subset of the base type; and a mapping from time to the particular value of the base type that the ACM contains at that time.

```
ACM: NONEMPTY_TYPE =
[# base_type: A_Type,
   valid_type: {t: A_Type | ∀ (v: Value): (v ∈ t) ⇒ (v ∈ base_type)},
   content:
     {f: [Time → Value] | ∀ (t: Time): (f(t) ∈ base_type)} #]
```

An ACM can be written to or read from.

```
Kind: TYPE = {read, write}

R_W_Action1: TYPE = [# kind: Kind #]

W_Action1: TYPE = {w: R_W_Action1 | w'kind = write}

R_Action1: TYPE = {r: R_W_Action1 | r'kind = read}
```

A mapping of reads and writes to RTL actions.

```
act: [R_W_Action1 → Action]
```

Uninterpreted functions that relate values read and written, and ACM accesses to read and write events.

```
val: [R_W_Action1, Occ, Value → bool]

access: [R_W_Action1, ACM → bool]
```

Functions that relate start and stop actions of reads and writes to RTL start and stop events.

```
stop(a: R_W_Action1): Event = stop(act(a))

start(a: R_W_Action1): Event = start(act(a))
```

A function that defines what it means for a reader or writer to communicate with an ACM - the reader or writer must start and end the read (write), access the ACM and read (write) a value from (to) the ACM.

```
communicates(a: R_W_Action1, i: Occ, t1, t2: Time, v: Value, acm: ACM): bool =
    Θ(start(a), i, t1) ∧
     Θ(stop(a), i, t2) ∧ val(a, i, v) ∧ access(a, acm)
```

A read or write must be related to a unique RTL action.

```
act_prop1(a1: R_W_Action1): bool =
    ∀ (a2: R_W_Action1): act(a1) = act(a2) ⇔ a1 = a2

R_W_Action2: TYPE = {a: R_W_Action1 | act_prop1(a)}

W_Action2: TYPE = {w: W_Action1 | act_prop1(w)}

R_Action2: TYPE = {r: R_Action1 | act_prop1(r)}
```

Read and write actions have unique start and stop events.

```
R_W_Action_th1: THEOREM
  ∀ (a1, a2: R_W_Action2):
    (stop(a1) = stop(a2) ⇒ a1 = a2) ∧
     (start(a1) = start(a2) ⇒ a1 = a2)
```

Start events of an action cannot be the same as stop events of another action.

```
R_W_Action_th2: THEOREM
  ∀ (a1, a2: R_W_Action2): stop(a1) ≠ start(a2) ∧ start(a1) ≠ stop(a2)
```

The start event for a read or write must occur before its stop event.

```
R_W_Action_th3: THEOREM
  ∀ (a: R_W_Action2, i: Occ, t1: Time):
    Θ(stop(a), i, t1) ⇒
     (∃ (t2: Time): Θ(start(a). i. t2) ∧ t2 ≤ t1)

R_W_Action_th3a: THEOREM
  ∀ (a: R_W_Action2, i: Occ, t1, t2: Time):
    Θ(start(a), i, t1) ∧ Θ(stop(a), i, t2) ⇒ t1 ≤ t2
```

Previous, and earlier, occurrences of a read or write must stop before later occurrences can start.

```
R_W_Action_th4: THEOREM
  ∀ (a: R_W_Action2, i: Occ, t1: Time):
    Θ(start(a), i + 1, t1) ⇒
     (∃ (t2: Time): Θ(stop(a), i, t2) ∧ t2 ≤ t1)

R_W_Action_th4a: THEOREM
  ∀ (a: R_W_Action2, i: Occ, t1, t2: Time):
    Θ(stop(a), i, t1) ∧ Θ(start(a), i + 1, t2) ⇒ t1 ≤ t2

R_W_Action_th5: THEOREM
  ∀ (a: R_W_Action2, i, j: Occ, t1: Time):
    Θ(start(a), i, t1) ∧ i > j ⇒
     (∃ (t2: Time): Θ(stop(a), j, t2) ∧ t2 ≤ t1)
```

A read must return a value (although that value may not be valid).

```
val_prop1(r: R_Action2): bool =
    ∀ (i: Occ):
      (∃ (t: Time): Θ(stop(r), i, t)) ⇔ (∃ (v: Value): val(r, i, v))
```

If a write to an ACM starts, there must be a value (that is to be written to the ACM) associated with the action.

```
val_prop2(w: W_Action2): bool =
    ∀ (i: Occ):
      (∃ (t: Time): Θ(start(w), i, t)) ⇔ (∃ (v: Value): val(w, i, v))
```

A (valid) read or write must relate to a unique value.

```
val_prop3(a: R_W_Action2): bool =
    ∀ (i: Occ, v1, v2: Value): val(a, i, v1) ∧ val(a, i, v2) ⇒ v1 = v2

valid_R_Action3(r: R_Action2): bool = val_prop1(r) ∧ val_prop3(r)

valid_W_Action3(w: W_Action2): bool = val_prop2(w) ∧ val_prop3(w)

valid_R_W_Action3(a: R_W_Action2): bool =
    (a'kind = read ⇒ valid_R_Action3(a)) ∧
     (a'kind = write ⇒ valid_W_Action3(a))

R_W_Action3: TYPE = {a: R_W_Action2 | valid_R_W_Action3(a)}

W_Action3: TYPE = {w: W_Action2 | valid_W_Action3(w)}

R_Action3: TYPE = {r: R_Action2 | valid_R_Action3(r)}
```

A read or a write must relate to an access to an ACM.

```
access_prop1(a: R_W_Action1): bool = ∃ (s: ACM): access(a, s)
```

A read or write must relate to an access to a unique ACM.

```
access_prop2(a: R_W_Action1): bool =
    ∀ (s1, s2: ACM): access(a, s1) ∧ access(a, s2) ⇒ s1 = s2
```

An ACM has a single writer.

```
access_prop3(s: ACM): bool =
    ∀ (w1, w2: W_Action1): access(w1, s) ∧ access(w2, s) ⇒ w1 = w2
```

Each ACM must have a reader and writer associated with it.

```
access_prop4(s: ACM): bool =
    ∃ (r: R_Action1, w: W_Action1): access(r, s) ∧ access(w, s)
```

Writers write valid values to ACMs.

```
write_val_prop1(acm: ACM): bool =
    ∀ (w: W_Action1, i: Occ, v: Value):
      val(w, i, v) ∧ access(w, acm) ⇒ (v ∈ acm'valid_type)
```

An initial value is written to an ACM at start up.

```
init_prop1(acm: ACM): bool =
    ∃ (w: W_Action1, v: Value, t: Time):
      t < Start_Time ∧ communicates(w, 0, t, Start_Time, v, acm)
```

(Valid) Read and Write actions relate to a unique ACM.

```
valid_R_W_Action(a: R_W_Action3): bool =
    access_prop1(a) ∧ access_prop2(a)

R_W_Action: TYPE = {a: R_W_Action3 | valid_R_W_Action(a)}

W_Action: TYPE = {w: W_Action3 | valid_R_W_Action(w)}

R_Action: TYPE = {r: R_Action3 | valid_R_W_Action(r)}
```

A read or write must relate to a unique access to an ACM.

```
comms_th: THEOREM
  ∀ (a: R_W_Action, i: Occ, t1, t2, t3, t4: Time, v1, v2: Value, acm1, acm2: ACM):
    communicates(a, i, t1, t2, v1, acm1) ∧ communicates(a, i, t3, t4, v2, acm2) ⇒
     t1 = t3 ∧ t2 = t4 ∧ v1 = v2 ∧ acm1 = acm2
```

A time is within an action if there is an earlier time when the action started and the action has not yet stopped.

```
time_in_action(a: R_W_Action, i: Occ, t: Time): bool =
    ∃ (t1: Time):
      t1 ≤ t ∧
       Θ(start(a), i, t1) ∧
        ¬ (∃ (t2: Time): t2 < t ∧ Θ(stop(a), i, t2))
```

Two actions overlap if they are both within instances of their actions at the same time.

```
overlapping_action(a1, a2: R_W_Action, i, j: Occ): bool =
    ∃ (t: Time): time_in_action(a1, i, t) ∧ time_in_action(a2, j, t)
```

Two actions conflict if they overlap and access the same ACM.

```
conflicting_actions(a1, a2: R_W_Action, i, j: Occ): bool =
    ∃ (acm: ACM):
      overlapping_action(a1, a2, i, j) ∧
       access(a1, acm) ∧ access(a2, acm)
```

A conflicting read is one that conflicts with a write.

```
conflicting_read(r: R_Action, i: Occ): bool =
    ∃ (w: W_Action, j: Occ): conflicting_actions(r, w, i, j)
```

An ACM is being written if there is a writer accessing it.

```
acm_being_written(acm: ACM, t: Time): bool =
    ∃ (w: W_Action, i: Occ): time_in_action(w, i, t) ∧ access(w, acm)
```

A read that accesses an ACM is either a conflicting read, or there was no write access to the ACM during the read.

```
conflicting_th: THEOREM
  ∀ (r: R_Action, i: Occ, t1, t2: Time, acm: ACM):
    Θ(start(r), i, t1) ∧ Θ(stop(r), i, t2) ∧ access(r, acm) ⇒
     conflicting_read(r, i) ∨
      ¬ (∃ (t: Time): acm_being_written(acm, t) ∧ t1 < t ∧ t ≤ t2)
```

If a read started after a write stopped, or a write started after the read stopped, they were not conflicting actions.

```
conflicting_th1: THEOREM
  ∀ (r: R_Action, w: W_Action, i, j: Occ, t1, t2: Time):
    (Θ(start(r), i, t1) ∧ Θ(stop(w), j, t2) ∧ t2 < t1) ∨
     (Θ(start(w), j, t2) ∧ Θ(stop(r), i, t1) ∧ t1 < t2)
     ⇒ ¬ conflicting_actions(r, w, i, j)
```

A basic ACM is one that has a single writer that writes valid values (values of the valid type as per the specification) to it, at least one reader, and is initialised with an initial value at system start up.

```
basic_acm(acm: ACM): bool =
    access_prop3(acm) ∧
     access_prop4(acm) ∧ write_val_prop1(acm) ∧ init_prop1(acm)

Basic_ACM: TYPE = {acm: ACM | basic_acm(acm)}
```

Only a single writer accesses a single basic ACM.

```
comms_th2: THEOREM
  ∀ (w1, w2: W_Action, i, j: Occ, t1, t2, t3, t4: Time, v1, v2: Value, acm: Basic_ACM):
    communicates(w1, i, t1, t2, v1, acm) ∧ communicates(w2, j, t3, t4, v2, acm) ⇒
     w1 = w2
```

If a writer communicates with an ACM for the (i + 1)th time, it must have previously communicated with it for the ith time.

```
comms_th3: THEOREM
  ∀ (w: W_Action, i: Occ, t1, t2: Time, v: Value, acm: Basic_ACM):
    communicates(w, i + 1, t1, t2, v, acm) ⇒
     (∃ (t3, t4: Time, v1: Value):
        communicates(w, i, t3, t4, v1, acm) ∧ t4 ≤ t1)
END General_ACMs
```

A persistent ACM retains the value that is written to it, until the value is overwritten. Any read that does not conflict with a write to the ACM will return the last value written. A read that conflicts with (occurs at the same time as, or overlaps in time with) a write can return any value from the base type of the ACM.

```
Persistent_ACM: THEORY
BEGIN

  IMPORTING General_ACMs
```

When a write to a persistent ACM finishes the content of the ACM is equal to the value written.

```
write_val_prop2(acm: Basic_ACM): bool =
    ∀ (w: W_Action, i: Occ, v: Value, t1, t2: Time):
      communicates(w, i, t1, t2, v, acm) ⇒ acm`content(t2) = v
```

A value remains in a persistent ACM until the start of the next write.

```
persistent_acm1(acm: Basic_ACM): bool =
    ∀ (w: W_Action, i: Occ, t1, t2: Time, v: Value):
      communicates(w, i, t1, t2, v, acm) ⇒
       (∃ (t3: Time):
          Θ(start(w), i + 1, t3) ∧ (∀ (t: Time): t2 ≤ t ∧ t < t3 ⇒ acm`content(t) = v)) ∨
        (¬ (∃ (t3: Time): Θ(start(w), i + 1, t3)) ∧
          (∀ (t: Time): t2 ≤ t ⇒ acm`content(t) = v))
```

A read that does not conflict with a write to a persistent ACM returns the value stored in (contents of) the ACM.

```
persistent_acm2(acm: Basic_ACM): bool =
    ∀ (r: R_Action, i: Occ, v: Value, t1, t2: Time):
      communicates(r, i, t1, t2, v, acm) ∧ ¬ conflicting_read(r, i) ⇒
       v = acm`content(t2)
```

A value contained in a persistent ACM must previously have been written to it, and a subsequent write must not have started.

```
persistent_acm3(acm: Basic_ACM): bool =
    ∀ (v: Value, t: Time):
      acm`content(t) = v ∧ ¬ acm_being_written(acm, t) ⇒
       (∃ (t1, t2: Time, w: W_Action, i: Occ):
          t2 < t ∧
           communicates(w, i, t1, t2, v, acm) ∧
            ¬ (∃ (t3: Time):
                 t2 ≤ t3 ∧ t3 < t ∧ Θ(start(w), i + 1, t3)))

Persistent_ACM: TYPE =
{acm: Basic_ACM |
          write_val_prop2(acm) ∧
           persistent_acm1(acm) ∧
            persistent_acm2(acm) ∧ persistent_acm3(acm)}
```

A writer writes valid values to a persistent ACM.

```
write_values: THEOREM
  ∀ (w: W_Action, i: Occ, v: Value, t1, t2: Time, acm: Persistent_ACM):
    communicates(w, i, t1, t2, v, acm) ⇒
     acm`content(t2) = v ∧ (v ∈ acm`valid_type)
```

Values written to a persistent ACM are valid between writes.

```
valid_between_writes: THEOREM
  ∀ (acm: Persistent_ACM, t: Time):
    (¬ (∃ (w: W_Action, i: Occ): time_in_action(w, i, t) ∧ access(w, acm))) ⇒
     (acm`content(t) ∈ acm`valid_type)
```

Values written to a persistent ACM do not change between writes.

```
unchanged_between_writes: THEOREM
  ∀ (acm: Persistent_ACM, t1, t2: Time):
    t1 ≤ t2 ∧
     (¬ (∃ (w: W_Action, i: Occ, t: Time):
           t1 ≤ t ∧ t ≤ t2 ∧ time_in_action(w, i, t) ∧ access(w, acm)))
    ⇒ acm`content(t1) = acm`content(t2)
```

A non-conflicting read that communicates with a persistent ACM will return a valid value.

```
persistent_reads_th1: THEOREM
  ∀ (r: R_Action, i: Occ, v: Value, t1, t2: Time, acm: Persistent_ACM):
    communicates(r, i, t1, t2, v, acm) ∧ ¬ conflicting_read(r, i) ⇒
     v = acm`content(t1)
```

The contents of a persistent ACM are written to it by the writer of the ACM.

```
  contents_are_written: THEOREM
    ∀ (v: Value, t: Time, acm: Persistent_ACM):
      v = acm`content(t) ∧ ¬ (∃ (w: W_Action, i: Occ): time_in_action(w, i, t)) ⇒
       (∃ (w: W_Action, i: Occ, t1, t2: Time):
          communicates(w, i, t1, t2, v, acm))
END Persistent_ACM
```

A (type) safe ACM is persistent, but gives the additional guarantee that a read that conflicts with a write will return a value of the type that the ACM is designed to communicate (a value of the valid type of the ACM).

```
Safe_ACM: THEORY
 BEGIN

  IMPORTING Persistent_ACM

  safe_acm(acm: Persistent_ACM): bool =
      ∀ (r: R_Action, i: Occ, t1, t2: Time, v: Value):
        communicates(r, i, t1, t2, v, acm) ⇒ (v ∈ acm`valid_type)

  Safe_ACM: TYPE = {acm: Persistent_ACM | safe_acm(acm)}
END Safe_ACM
```

A semi-regular ACM is (type) safe, and additionally guarantees that a read that conflicts with a write will return a value that has previously been written to it (this is the formal definition of a coherent ACM).

```
Semiregular_ACM: THEORY
 BEGIN

  IMPORTING Safe_ACM

  semiregular_acm(acm: Safe_ACM): bool =
      ∀ (r: R_Action, i: Occ, t1, t2: Time, v: Value):
        communicates(r, i, t1, t2, v, acm) ⇒
         (∃ (w: W_Action, j: Occ, t3, t4: Time):
            t3 ≤ t2 ∧ communicates(w, j, t3, t4, v, acm))

  Semiregular_ACM: TYPE = {acm: Safe_ACM | semiregular_acm(acm)}
END Semiregular_ACM
```

A regular ACM is semi-regular, but additionally guarantees that a non-conflicting read will return the value that was written by the last write. A conflicting read will either return the value written by the last write to end before the read started, or one of the values written by one of the conflicting writes (it will return a valid value), but, if a number of reads conflict with a write, it is possible that one of the later conflicting reads will return an item that was written before the value returned by one of the earlier conflicting reads i.e the values may not be returned in the order that they were written. This is the formal definition of local freshness.

```
Regular_ACM: THEORY
 BEGIN
```

```
IMPORTING Semiregular_ACM

regular_acm(acm: Semiregular_ACM): bool =
    ∀ (r: R_Action, i: Occ, t1, t2: Time, v: Value):
      communicates(r, i, t1, t2, v, acm) ∧ conflicting_read(r, i) ⇒
       (∃ (w: W_Action, j: Occ, t3, t4: Time):
          communicates(w, j, t3, t4, v, acm) ∧
           ((t4 < t1 ∧
              ¬ (∃ (t5, t6: Time, v1: Value):
                   communicates(w, j + 1, t5, t6, v1, acm) ∧ t6 ≤ t1))
             ∨ conflicting_actions(r, w, i, j)))

Regular_ACM: TYPE = {acm: Semiregular_ACM | regular_acm(acm)}
END Regular_ACM
```

An atomic ACM is regular, but additionally guarantees that items will be read from the ACM in the order that they are written to it. That is that the reads and writes to the ACM will appear to have happened in some Hoare atomic order, although it is possible for an item to be overwritten before it is read, or an single item to be read multiple times. This is the formal definition of global freshness. An atomic ACM communicates data items, which have an index number as well as a value. The index numbers start at zero and increment by one each time an item is written so that it is possible to reason about the order that the items are written and read.

```
Atomic_ACM: THEORY
BEGIN

IMPORTING Regular_ACM
```

A *DataItem* consists of an occurrence (sequence) number, a value and an ACM that contains it.

```
DataItem: TYPE = [# id: Occ, value: Value, acm: ACM #]
```

DataItems have unique Ids.

```
DataItem_Ax: AXIOM
  ∀ (x, y: DataItem): y`id = x`id ⇒ y`value = x`value ∧ y`acm = x`acm
```

A writer that communicates a *DataItem* to an ACM, must write the value of the *DataItem*, communicate with the ACM that contains the *DataItem* and the write must have the same occurrence number as the sequence number of the *DataItem* (because there is only a single writer to an ACM).

```
communicates(w: W_Action, i: Occ, t1, t2: Time, x: DataItem, acm: ACM): bool =
    communicates(w, i, t1, t2, x`value, acm) ∧ x`acm = acm ∧ x`id = i
```

A reader that communicates with an L-atomic ACM must read the item that was written by the last write to end before the read started, or by a write that overlaps in time with the read (the item must have the same sequence number as the occurrence number of a previous, or conflicting, write that communicated it to the ACM).

```
r_communicates(r: R_Action, i: Occ, t1, t2: Time, x: DataItem, acm: ACM): bool =
    communicates(r, i, t1, t2, x`value, acm) ∧ x`acm = acm ∧
      x`id ∈ {j: Occ | ∃ (w: W_Action, t3, t4: Time):
        communicates(w, j, t3, t4, x, acm) ∧ (t4 ≤ t1 ∧
          ¬ (∃ (t5, t6: Time, y: DataItem):
            communicates(w, j+1, t5, t6, y, acm) ∧
              t6 ≤ t1) ∨ conflicting_actions(r, w, i, j))}
```

A writer communicates with a single ACM.

```
comms_dataitem_th: THEOREM
  ∀ (w: W_Action, i: Occ, t1, t2, t3, t4: Time, x1, x2: DataItem, acm1, acm2: ACM):
    communicates(w, i, t1, t2, x1, acm1) ∧ communicates(w, i, t3, t4, x2, acm2) ⇒
      t1 = t3 ∧ t2 = t4 ∧ x1 = x2 ∧ acm1 = acm2
```

A unique write communicates a *DataItem* to an ACM.

```
comms_dataitem_th1: THEOREM
  ∀ (w: W_Action, i, j: Occ, t1, t2, t3, t4: Time, x: DataItem, acm1, acm2: ACM):
    communicates(w, i, t1, t2, x, acm1) ∧ communicates(w, j, t3, t4, x, acm2) ⇒
      i = j
```

A unique writer communicates with an ACM.

```
comms_dataitem_th2: THEOREM
  ∀ (w1, w2: W_Action, i, j: Occ, t1, t2, t3, t4: Time, x1, x2: DataItem, acm: Basic_ACM):
    communicates(w1, i, t1, t2, x1, acm) ∧ communicates(w2, j, t3, t4, x2, acm) ⇒
      w1 = w2
```

Items are written to ACMs by the writers of the ACMs.

```
item(w: W_Action, i: Occ, x: DataItem): bool =
   ∃ (t1, t2: Time): communicates(w, i, t1, t2, x, x`acm)
```

Any item written to an ACM has a sequence number that is the same as the write that communicated it.

```
item_prop1: THEOREM
  ∀ (w: W_Action, i: Occ, x: DataItem):
    item(w, i, x) ⇒
      (∃ (t1, t2: Time):
         communicates(w, i, t1, t2, x`value, x`acm) ∧ x`id = i)
```

If a writer communicates a value to an ACM, then a *DataItem* exists with the value, and relevant sequence number, in the ACM at the end of the write.

```
item_prop2: THEOREM
  ∀ (w: W_Action, i: Occ, t1, t2: Time, v: Value, acm: ACM):
    communicates(w, i, t1, t2, v, acm) ⇒
      (∃ (x: DataItem):
         item(w, i, x) ∧ x`value = v ∧ x`acm = acm ∧ x`id = i)
```

DataItems are unique.

```
item_th1: THEOREM
  ∀ (i: Occ, w: W_Action, x, y: DataItem):
    item(w, i, x) ∧ item(w, i, y) ⇒ x = y
```

A *DataItem* is written to an ACM once only.

```
item_th2: THEOREM
  ∀ (w: W_Action, i, j: Occ, x: DataItem):
    item(w, i, x) ∧ item(w, j, x) ⇒ i = j
```

There is a stop action of a write if and only if there is a *DataItem* that has been written by that write.

```
item_th3: THEOREM
  ∀ (w: W_Action, i: Occ):
    (∃ (t: Time): Θ(stop(w), i, t)) ⇔
     (∃ (x: DataItem): item(w, i, x))
```

If a writer communicates with an ACM there is an associated *DataItem* that is written.

```
item_th4: THEOREM
  ∀ (w: W_Action, i: Occ, t1, t2: Time, x: DataItem, acm: ACM):
    communicates(w, i, t1, t2, x, acm) ⇒ item(w, i, x)
```

All *DataItems* have been written by an associated writer that communicates with the ACM.

```
item_th5: THEOREM
  ∀ (w: W_Action, i: Occ, x: DataItem):
    item(w, i, x) ⇒
     (∃ (t1, t2: Time): communicates(w, i, t1, t2, x, x`acm))
```

An atomic ACM is a regular ACM that reads the items written in the order that they were written (once an item has been read it is not possible in any circumstances for a reader to read items that were written previously to the item read).

```
atomic_acm(acm: Regular_ACM): bool =
    ∀ (r: R_Action, i: Occ, t3, t4: Time, x2: DataItem):
      r_communicates(r, i+1, t3, t4, x2, acm) ⇒
       (∃ (t1, t2: Time, x1: DataItem):
          r_communicates(r, i, t1, t2, x1, acm) ∧ x2`id ≥ x1`id)

Atomic_ACM: TYPE = {acm: Regular_ACM | atomic_acm(acm)}
```

A reader can only read items that have previously been written to the ACM.

```
reads_read_items: THEOREM
  ∀ (r: R_Action, i: Occ, t1, t2: Time, v: Value, acm: Atomic_ACM):
    communicates(r, i, t1, t2, v, acm) ⇒
     (∃ (w: W_Action, j: Occ, t3, t4: Time, x: DataItem):
        communicates(w, j, t3, t4, x, acm) ∧ t3 ≤ t2 ∧ x`value = v)
```

An atomic ACM has all of the properties of a regular ACM.

```
atomic_acm_is_regular_th: THEOREM
  ∀ (r: R_Action, i: Occ, t1, t2: Time, v: Value, acm: Atomic_ACM):
    communicates(r, i, t1, t2, v, acm) ∧ conflicting_read(r, i) ⇒
     (∃ (w: W_Action, j: Occ, t3, t4: Time, x: DataItem):
        communicates(w, j, t3, t4, x, acm) ∧
         ((t4 < t1 ∧
```

```
        x'value = v ∧
         x'id = j ∧
          x'acm = acm ∧
           ¬ (∃ (t5, t6: Time, v1: Value, x1: DataItem):
                communicates(w, j+1, t5, t6, x1, acm) ∧ t6 ≤ t1))
      ∨ conflicting_actions(r, w, i, j)))
```

A reader can only communicate items that have previously been written.

```
r_communicates_th1: THEOREM
  ∀ (r: R_Action, i: Occ, t1, t2: Time, v: Value, acm: Atomic_ACM):
    communicates(r, i, t1, t2, v, acm) ⇒
     (∃ (x: DataItem): r_communicates(r, i, t1, t2, x, acm))

r_communicates_th2: THEOREM
  ∀ (r: R_Action, w: W_Action, i, j: Occ, t1, t2, t3, t4: Time, x: DataItem,
                                  acm: Atomic_ACM):
    r_communicates(r, i, t1, t2, x, acm) ∧ communicates(w, j, t3, t4, x, acm) ⇒
     t3 ≤ t2
```

Items that have been overwritten by subsequent writes are not available to the reader (they have been overwritten).

```
overwritten_items_lost_th: THEOREM
  ∀ (w: W_Action, i: Occ, t1, t2, t3, t4: Time, x1, x2:
                          DataItem, acm: Atomic_ACM):
    communicates(w, i, t1, t2, x1, acm) ∧ communicates(w, i+1, t3, t4, x2, acm) ⇒
     ¬ (∃ (r: R_Action, j: Occ, t5, t6: Time):
          r_communicates(r, j, t5, t6, x1, acm) ∧ t5 ≥ t4)

overwritten_items_lost_th1: THEOREM
  ∀ (w: W_Action, i: Occ, j: Occ, t1, t2, t3, t4: Time, x1, x2: DataItem, acm: Atomic_ACM):
    communicates(w, i, t1, t2, x1, acm) ∧ communicates(w, j, t3, t4, x2, acm) ∧ i < j ⇒
     ¬ (∃ (r: R_Action, k: Occ, t5, t6: Time):
          r_communicates(r, k, t5, t6, x1, acm) ∧ t5 ≥ t4)
```

If the previous occurrence of a read has read the value written by an overlapping write, the next read cannot get the value from a data item written by an earlier write (unless the value from the earlier data item is the same as the value from the later one). The theorem is illustrated as follows:

```
|-w,i-| |-w,i+1-|    |--w,i+2--|       |--w,i+3--|
t5   t6 t7     t8    t9      t10       t11     t12
               |---r,j---|  |--r,j+1--|
               t1       t2  t3       t4
```

```
atomic_test_th: THEOREM
  ∀ (r: R_Action, w: W_Action, i, j: Occ,
     t1, t2, t3, t4, t5, t6, t7, t8, t9, t10, t11, t12: Time, x1, x2, x3, x4: DataItem,
     v: Value, acm: Atomic_ACM):
    t2 ≥ t9 ∧
     t10 ≥ t3 ∧
      t11 > t4 ∧
       communicates(w, i, t5, t6, x1, acm) ∧
        communicates(w, i+1, t7, t8, x2, acm) ∧
         communicates(w, i+2, t9, t10, x3, acm) ∧
          communicates(w, i+3, t11, t12, x4, acm) ∧
           x3'value ≠ x2'value ∧
            x3'value ≠ x1'value ∧
             communicates(r, j, t1, t2, x3'value, acm) ∧
```

communicates(r, $j+1$, t_3, t_4, v, acm) $\wedge$ $t_1 \geq t_7$
$\Rightarrow$ x_3'value $= v$

END Atomic_ACM

Appendix D

Simpson's 4-slot

This appendix contains the full model of Simpson's 4-slot fully asynchronous ACM, introduced in Chapter 3, in the PVS logic. First some basic types are defined, which are used in the model.

```
Supporting_Types: THEORY
 BEGIN
```

A value type to represent the values that are communicated.

```
  Val: NONEMPTY_TYPE
```

The data items that are communicated consist of a serial number and a value.

```
  Data: TYPE = [# index: nat, val: Val #]
 END Supporting_Types

FOUR_SLOT: THEORY
 BEGIN

  IMPORTING Supporting_Types
```

Types to represent the pairs and slots in the ACM: there are two pairs of two slots in the 4-slot.

```
  PairIndex: TYPE = {p0, p1}

  SlotIndex: TYPE = {s0, s1}
```

The program counters for the reader and writer, which record the next instruction to be executed in their respective algorithms.

```
  NextReadInstruction: TYPE = {rcp, rip, rcs, rd}

  NextWriteInstruction: TYPE = {wcp, wcs, wr, wis, wip}
```

The local state of the writer: it keeps local copies of the names of the pair and slot it is accessing.

```
WriterState: TYPE = [# writerPair: PairIndex, writerSlot: SlotIndex #]
```

The local state of the reader: again it keeps local copies of the names of the pair and slot is is accessing.

```
ReaderState: TYPE = [# readerPair: PairIndex, readerSlot: SlotIndex #]
```

The ACM has three control variables: *pairWritten*, which records the name of the last pair of slots the writer has indicated it has accessed; *pairReading*, which records the name of the pair of slots the reader last indicated it has accessed, or is accessing; and *slotWritten*, an array of two values which contains the names of the slot that the writer last accessed in each pair of slots. The ACM also contains the four slots that are used to transmit the data, and the auxiliary variables: the reader and writer program counters: and *writerChangedPairNI*, which records if the writer has changed pairs and not yet executed
writerIndicatesPair to indicate it has changed. Finally it contains the reader and writer local state.

```
Conc_State: TYPE = [# pairWritten: PairIndex,
                      slotWritten: [PairIndex → SlotIndex],
                      pairReading: PairIndex,
                      slots: [PairIndex, SlotIndex → Val],
                      nri: NextReadInstruction,
                      nwi: NextWriteInstruction,
                      writer: WriterState,
                      reader: ReaderState,
                      writerChangedPairNI: bool #]
```

The first action in the reader algorithm is when it chooses the pair it is going to access during the read: it attempts to read the latest item by reading from the pair of slots the writer last indicated it has accessed.

```
pre_readerChoosesPair(p: Conc_State): bool = p'nri = rcp

post_readerChoosesPair(p: (pre_readerChoosesPair))(prot: Conc_State): bool =
    prot =
     p WITH [nri := rip,
             reader := p'reader WITH [readerPair := p'pairWritten]]

readerChoosesPair: [p: (pre_readerChoosesPair) → (post_readerChoosesPair(p))]

pre_readerIndicatesPair(p: Conc_State): bool = p'nri = rip
```

The second read action is to indicate the pair of slots that it is going to access, in the *pairReading* control variable.

```
post_readerIndicatesPair(p: (pre_readerIndicatesPair))(prot: Conc_State): bool =
    prot = p WITH [nri := rcs, pairReading := p'reader'readerPair]

readerIndicatesPair: [p: (pre_readerIndicatesPair) → (post_readerIndicatesPair(p))]
```

The reader chooses to read from the last slot the writer accessed, in the pair of slots it has chosen to read from.

```
pre_readerChoosesSlot(p: Conc_State): bool = p'nri = rcs

post_readerChoosesSlot(p: (pre_readerChoosesSlot))(prot: Conc_State): bool =
    prot =
     p WITH [nri := rd,
             reader := p'reader
                     WITH [readerSlot
                             := p'slotWritten(p'reader'readerPair)]]

readerChoosesSlot: [p: (pre_readerChoosesSlot) → (post_readerChoosesSlot(p))]
```

Finally the reader reads the data from its chosen slot.

```
pre_read(p: Conc_State): bool = p'nri = rd

post_read(p: (pre_read))(prot: Conc_State, v: Val): bool =
    v = p'slots(p'reader'readerPair, p'reader'readerSlot) ∧
     prot = p WITH [nri := rcp]

read: [p: (pre_read) → (post_read(p))]
```

The first action of the writer is to choose the pair of slots it is going to access. it attempts to avoid the reader by choosing to access the opposite pair of slots to the one that the reader last indicated it was going to access. It also sets the *writerIndicatesPairNI* boolean to true if the writer changes pairs for the write (the reader has indicated it is accessing the pair of slots the writer last indicated is was accessing), and to false if the writer does not change pairs (in fact this leaves the value unchanged, since it will have already been set to false by the previous *writerIndicatesPair* operation).

```
pre_writerChoosesPair(p: Conc_State): bool = p'nwi = wcp

post_writerChoosesPair(p: (pre_writerChoosesPair))(prot: Conc_State): bool =
    (p'pairReading = p'pairWritten ⇒
      (p'pairReading = p0 ⇒
        prot =
         p WITH [nwi := wcs,
                 writer := p'writer WITH [writerPair := p1],
                 writerChangedPairNI := TRUE]) ∧
       (p'pairReading = p1 ⇒
         prot =
          p WITH [nwi := wcs,
                  writer := p'writer WITH [writerPair := p0],
                  writerChangedPairNI := TRUE])) ∧
     (¬ p'pairReading = p'pairWritten ⇒
       (p'pairReading = p0 ⇒
         prot =
          p WITH [nwi := wcs,
                  writer := p'writer WITH [writerPair := p1],
                  writerChangedPairNI := FALSE]) ∧
        (p'pairReading = p1 ⇒
          prot =
           p WITH [nwi := wcs,
                   writer := p'writer WITH [writerPair := p0],
                   writerChangedPairNI := FALSE]))

writerChoosesPair: [p: (pre_writerChoosesPair) → (post_writerChoosesPair(p))]
```

The writer then chooses the slot it is going to access in the pair of slots it has chosen. It again attempts to avoid the reader by accessing the opposite slot to the one it accessed the last time in its chosen pair.

```
pre_writerChoosesSlot(p: Conc_State): bool = p'nwi = wcs

wcs_result: TYPE = [# prot: Conc_State, v: Val #]

post_writerChoosesSlot(p: (pre_writerChoosesSlot))(prot: Conc_State): bool =
    (p'slotWritten(p'writer'writerPair) = s0 ⇒
      prot = p WITH [nwi := wr, writer := p'writer WITH [writerSlot := s1]]) ∧
     (p'slotWritten(p'writer'writerPair) = s1 ⇒
       prot =
        p WITH [nwi := wr, writer := p'writer WITH [writerSlot := s0]])

writerChoosesSlot: [p: (pre_writerChoosesSlot) → (post_writerChoosesSlot(p))]
```

Once the writer has chosen the slot it is going to access it writes the new data item to its chosen slot.

```
pre_write(p: Conc_State): bool = p'nwi = wr

post_write(p: (pre_write))(prot: Conc_State): bool =
    prot =
     p WITH [nwi := wis,
             (slots)(p'writer'writerPair, p'writer'writerSlot) := v]

write: [p: (pre_write) → (post_write(p))]
```

After the writer has written the new item of data to its chosen slot it indicates the slot it has written in the relevant element of the *slotWritten* array.

```
pre_writerIndicatesSlot(p: Conc_State): bool = p'nwi = wis

post_writerIndicatesSlot(p: (pre_writerIndicatesSlot))(prot: Conc_State): bool =
    prot =
     p WITH [nwi := wip,
             (slotWritten)(p'writer'writerPair) := (p'writer'writerSlot)]

writerIndicatesSlot: [p: (pre_writerIndicatesSlot) → (post_writerIndicatesSlot(p))]
```

The final writer action in each write is to indicate the pair of slots it has accessed in the *pairWritten* control variable.

```
pre_writerIndicatesPair(p: Conc_State): bool = p'nwi = wip

post_writerIndicatesPair(p: (pre_writerIndicatesPair))(prot: Conc_State): bool =
    prot =
     p WITH [nwi := wcp,
             pairWritten := p'writer'writerPair,
             writerChangedPairNl := FALSE]

writerIndicatesPair: [p: (pre_writerIndicatesPair) → (post_writerIndicatesPair(p))]
```

The following functions combine the reader and writer actions in the implementation into combined actions that are equivalent to the actions in the abstract model in Appendix E, so that it is possible to show that the implementation is a refinement of the model (provided the combined actions are executed atomically).

startRd is a combination of *readerChoosesPair*, *readerIndicatesPair* and *readerChoosesSlot*.

```
pre_startRd(p: Conc_State): bool = p'nri = rcp

post_startRd(p: (pre_readerChoosesPair))(prot: Conc_State): bool =
    prot = readerChoosesSlot(readerIndicatesPair(readerChoosesPair(p)))

startRd: [p: (pre_readerChoosesPair) → (post_startRd(p))]
```

endRd is only required to return the value read from the chosen slot and therefore only uses the read operation.

```
pre_endRd(p: Conc_State): bool = p'nri = rd

post_endRd(p: (pre_read))(p1: Conc_State, v: Val): bool =
    p1 = read(p)'1 ∧ v = read(p)'2

endRd: [p: (pre_read) → (post_endRd(p))]
```

startWr combines the *writerChoosesPair*, *writerChoosesSlot*, *write* and *writerIndicatesSlot* operations and is equivalent to start write in the abstract model: it adds the new item to the ACM, and makes it available to be read in some circumstances, by indicating the slot it has written to (the reader may then read this item if it is accessing the same pair as the writer, and executes *startRd* after the *startWr* operation, even if the writer has not executed *endWr* to inicated the pair it has accessed).

```
pre_startWr(p: Conc_State): bool = p'nwi = wcp

post_startWr(p: (pre_writerChoosesPair))(prot: Conc_State): bool =
    prot =
     writerIndicatesSlot(write(writerChoosesSlot(writerChoosesPair(p))))

startWr: [p: (pre_writerChoosesPair) → (post_startWr(p))]
```

endWr completes the write by executing *writerIndicatesPair*.

```
pre_endWr(p: Conc_State): bool = p'nwi = wip

post_endWr(p: (pre_writerIndicatesPair))(p1: Conc_State): bool =
    p1 = writerIndicatesPair(p)

endWr: [p: (pre_writerIndicatesPair) → (post_endWr(p))]
```

The reader and writer local states are initialised to point to different pairs and different slots. The initialisation is not important, because the reader will always attempt to follow the writer to the latest slot written (or to access the slot with the initial data item if the first read occurs before the first write), and the writer will always attempt to avoid the reader.

```
init_writer(w: WriterState): bool =
    w = w WITH [writerPair := p0, writerSlot := s0]

init_reader(r: ReaderState): bool =
    r = r WITH [readerPair := p1, readerSlot := s1]
```

One of the slots is initialised and the control variables are set to point to this slot. *pairReading* is set equal to the reader local variable *readerPair* and the program counters are set to their initial values: so that the initial read and write actions will be executed first.

```
init_prot(p: Conc_State, init_val: Val, w: WriterState, r: ReaderState): bool =
    LET w = w WITH [writerPair := p0, writerSlot := s0],
        r = r WITH [readerPair := p1, readerSlot := s1]
      IN
      p = p
         WITH [pairWritten := p0,
               slotWritten := ((λ ·(p0: PairIndex): s0),
               pairReading := p1,
               slots := ((λ ·(p0: PairIndex, s0: SlotIndex): init_val),
               nri := rcp,
               nwi := wcp,
               writer := w,
               reader := r]
END FOUR_SLOT
```

Appendix E

An Abstract Model of L-Atomicity

This appendix contains the full model of L-atomicity, which was introduced in Chapter 4, in the PVS logic. This model uses the basic types in the Supporting Types theory given in Appendix D.

```
Abstract_Protocol: THEORY
 BEGIN

  IMPORTING Supporting_Types, finite_sequences[Data]
```

The model uses a (PVS) finite sequence to contain the items that are available to the reader. The sequence has a minimum length of 1.

```
  Val_Sequence: TYPE =
  {fin_seq: finite_sequence[Data] | fin_seq'length ≥ 1}

  (seq: Val_Sequence ∪ {d: Data}): Val_Sequence =
      (# length := 1, seq := (λ ·(x: below[1]): d) #) ∘ seq
```

The abstract model of the ACM, which has a sequence of data items, two booleans to record whether the reader and writer are accessing the ACM or not, and three auxiliary variables that are used to check for L-atomicity: *nextIndex*, *indexRead* and *firstIndexAvailable*.

```
  Abs_State: TYPE =
  [# vals: Val_Sequence,
     writerAccess: bool,
     readerAccess: bool,
     nextIndex: nat,
     indexRead: nat,
     firstIndexAvailable: nat #]

  pre_start_read(prot: Abs_State): bool = prot'readerAccess = FALSE
```

At start read the reader shortens the sequence, if necessary, to contain only those items that are available to be read. If the sequence is of length greater

than 1 it is shortened to length one if there is no write in progress and to length 2 otherwise (the item written by the last complete write and the item written by the write that is in progress). It also sets *firstIndexAvailable* equal to the index of the first item that is available to be read: which will be the index of the item written by the last complete write.

```
post_start_read(p: (pre_start_read))(prot: Abs_State): bool =
    IF p'vals'length = 1
      THEN prot =
            p WITH [readerAccess := TRUE, firstIndexAvailable := p'vals(0)'index]
    ELSE IF ¬ p'writerAccess
           THEN prot =
                 p
                   WITH [vals := p'vals ^ (0, 0),
                         readerAccess := TRUE,
                         firstIndexAvailable := p'vals(0)'index]
         ELSE prot =
               p
                 WITH [vals := p'vals ^ (0, 1),
                       readerAccess := TRUE,
                       firstIndexAvailable := p'vals(1)'index]
         ENDIF
    ENDIF

start_read: [p: (pre_start_read) → (post_start_read(p))]
```

End read chooses an item to be read from the sequence of items, returns that item and removes all of the older items from the sequence. It also sets *indexRead* equal to the index of the item returned.

```
pre_end_read(prot: Abs_State): bool = prot'readerAccess = TRUE

post_end_read(p: (pre_end_read))(prot: Abs_State, read_item: Val): bool =
    ∃ (i: nat):
      i < p'vals'length ∧
       read_item = p'vals'seq(i)'val ∧
        IF p'vals'length > 1
          THEN prot =
                p
                  WITH [vals := p'vals ^ (0, i),
                        readerAccess := FALSE,
                        indexRead := p'vals'seq(i)'index]
        ELSE prot = p WITH [readerAccess := FALSE, indexRead := p'vals'seq(i)'index]
        ENDIF

end_read: [p: (pre_end_read) → (post_end_read(p))]
```

Start write shortens the sequence of items to length 1 if there is no read in progress, because the only item available to the reader at this stage is the one last written. It also adds the item being written to the head of the sequence. Each item has a sequence number, equal to *nextIndex*, which is incremented in time for the next write.

```
pre_start_write(prot: Abs_State): bool = prot'writerAccess = FALSE

write_parameter: TYPE = [# p1: (pre_start_write), val: Val #]

post_start_write(p: write_parameter)(prot: Abs_State): bool =
```

```
    LET newItem: Data = (# index := p'p1'nextIndex, val := p'val #) IN
      prot =
       p'p1
         WITH [vals := (p'p1'vals ∪ {newItem}),
               writerAccess := TRUE,
               nextIndex := p'p1'nextIndex + 1]

start_write: [p: write_parameter → (post_start_write(p))]
```

End write shortens the sequence to contain only the item just written, if there is no read in progress, since this is the only item that is now available to be read. If there is a read in progress the writer cannot tell which item the reader will choose to read (at end read) and it therefore leaves the sequence unchanged.

```
pre_end_write(prot: Abs_State): bool = prot'writerAccess = TRUE

post_end_write(p: (pre_end_write))(prot: Abs_State): bool =
    (p'readerAccess = TRUE ⇒ prot = p WITH [writerAccess := FALSE]) ∧
     (p'readerAccess = FALSE ⇒
       prot = p WITH [vals := p'vals ^ (0, 0), writerAccess := FALSE])

end_write: [p: (pre_end_write) → (post_end_write(p))]
```

The sequence is initialised with a data item, in case a read starts before the first write.

```
init(prot: Abs_State, init_item: Val): bool =
    prot = prot
       WITH [vals := (# length := 1, seq := (# index := 0, val := init_item #) #),
             writerAccess := FALSE,
             readerAccess := FALSE,
             nextIndex := 1,
             indexRead := 0,
             firstIndexAvailable := 0]
```

The assertions that are made in the locations of the state machine of the model. There are four locations, when there is no read or write in progress, when there is only a read in progress, when there is only a write in progress and when there is both a read and a write in progress. The assertions are defined using lambda functions, so that they can be used by name in the proof obligations and expanded in line in the proofs. The assertions relate the values of the auxiliary variables and are sufficiently strong that to verify that the ACM in the model is L-atomic as described in Chapter 4. The final conjunct in the assertions is part of the invariant in the VDM-SL-like model in Chapter 4.

```
noReader_noWriter_Assertion: [Abs_State → bool] =
    (λ  · (abs: Abs_State):
      abs'indexRead ≤ abs'nextIndex-abs'vals'length ∧
       abs'firstIndexAvailable ≤ abs'nextIndex-abs'vals'length ∧
        abs'vals(0)'index = abs'nextIndex-1 ∧
         (∀ (n: nat):
            n < abs'vals'length ∧ n > 0 ⇒
             abs'vals(n)'index =
```

```
            abs'nextIndex-(n + 1))

reader_noWriter_Assertion: [Abs_State → bool] =
    (λ ·(abs: Abs_State):
      abs'indexRead ≤ abs'nextIndex-abs'vals'length ∧
       abs'firstIndexAvailable = abs'nextIndex-abs'vals'length ∧
        abs'vals(0)'index = abs'nextIndex-1 ∧
         (∀ (n: nat):
            n < abs'vals'length ∧ n > 0 ⇒
             abs'vals(n)'index =
              abs'nextIndex-(n + 1))

noReader_writer_Assertion: [Abs_State → bool] =
    (λ ·(abs: Abs_State):
      abs'indexRead ≤ abs'nextIndex-abs'vals'length ∧
       abs'firstIndexAvailable ≤ abs'nextIndex-abs'vals'length ∧
        abs'vals(0)'index = abs'nextIndex-1 ∧
         (∀ (n: nat):
            n < abs'vals'length ∧ n > 0 ⇒
             abs'vals(n)'index =
              abs'nextIndex-(n + 1))

reader_writer_Assertion: [Abs_State → bool] =
    (λ ·(abs: Abs_State):
      abs'indexRead ≤ abs'nextIndex-abs'vals'length ∧
       abs'firstIndexAvailable = abs'nextIndex-abs'vals'length ∧
        abs'vals(0)'index = abs'nextIndex-1 ∧
         (∀ (n: nat):
            n < abs'vals'length ∧ n > 0 ⇒
             abs'vals(n)'index =
              abs'nextIndex-(n + 1))
```

The model must satisfy the following conjecture in order to be model of L-atomicity: any item read must have an index number greater than or equal to the index of the first item available to the reader, less than the index of the next item to be written, and greater than or equal to the index of the item last read.

```
lamport: [Abs_State, Abs_State → bool] =
    (λ ·(as1, as2: Abs_State):
      as1'indexRead ≤ as2'indexRead ∧
       as2'firstIndexAvailable ≤ as2'indexRead ∧
        as2'nextIndex-1 ≥ as2'indexRead
```

The proof obligations, that need to be discharged: when a transition is enabled and the associated operation is executed, that if the assertion in the start location of the transition holds before the operation is executed, the assertion in the target location will hold after the operation is executed. In addition when a read is executed the index of the item read must satisfy the "lamport" conjecture above, in order to satisfy L-atomicity.

```
vc_noReader_noWriter_start_read: THEOREM
  ∀ (as1, as2: Abs_State):
    pre_start_read(as1) ∧
     ¬ as1'writerAccess ∧ noReader_noWriter_Assertion(as1) ∧ as2 = start_read(as1) ⇒
     as2'readerAccess ∧
      ¬ as2'writerAccess ∧ reader_noWriter_Assertion(as2)
```

```
vc_reader_noWriter_end_read: THEOREM
  ∀ (as1, as2: Abs_State):
    pre_end_read(as1) ∧
     ¬ as1'writerAccess ∧ reader_noWriter_Assertion(as1) ∧ as2 = end_read(as1)'1 ⇒
     ¬ as2'readerAccess ∧
      ¬ as2'writerAccess ∧
       noReader_noWriter_Assertion(as2) ∧ lamport(as1, as2)

vc_reader_noWriter_start_write: THEOREM
  ∀ (w: write_parameter, as2: Abs_State):
    pre_start_write(w'p1) ∧
     w'p1'readerAccess ∧ reader_noWriter_Assertion(w'p1) ∧ as2 = start_write(w)
     ⇒ as2'readerAccess ∧ as2'writerAccess ∧ reader_writer_Assertion(as2)

vc_reader_writer_end_write: THEOREM
  ∀ (as1, as2: Abs_State):
    pre_end_write(as1) ∧ as1'readerAccess ∧ reader_writer_Assertion(as1) ∧ as2 = end_write(as1) ⇒
     as2'readerAccess ∧
      ¬ as2'writerAccess ∧ reader_noWriter_Assertion(as2)

vc_noReader_noWriter_start_write: THEOREM
  ∀ (w: write_parameter, as2: Abs_State):
    pre_start_write(w'p1) ∧
     ¬ w'p1'readerAccess ∧ noReader_noWriter_Assertion(w'p1) ∧ as2 = start_write(w) ⇒
     as2'writerAccess ∧
      ¬ as2'readerAccess ∧ noReader_writer_Assertion(as2)

vc_noReader_writer_end_write: THEOREM
  ∀ (as1, as2: Abs_State):
    pre_end_write(as1) ∧
     ¬ as1'readerAccess ∧ noReader_writer_Assertion(as1) ∧ as2 = end_write(as1) ⇒
     ¬ as2'writerAccess ∧
      ¬ as2'readerAccess ∧ noReader_noWriter_Assertion(as2)

vc_noReader_writer_start_read: THEOREM
  ∀ (as1, as2: Abs_State):
    pre_start_read(as1) ∧
     as1'writerAccess ∧ noReader_writer_Assertion(as1) ∧ as2 = start_read(as1)
     ⇒ as2'writerAccess ∧ as2'readerAccess ∧ reader_writer_Assertion(as2)

vc_reader_writer_end_read: THEOREM
  ∀ (as1, as2: Abs_State):
    pre_end_read(as1) ∧ as1'writerAccess ∧ reader_writer_Assertion(as1) ∧ as2 = end_read(as1)'1
     ⇒
     as2'writerAccess ∧
      ¬ as2'readerAccess ∧
       noReader_writer_Assertion(as2) ∧ lamport(as1, as2)

END Abstract_Protocol
```

Appendix F

The Retrieve Relation

This appendix contains the retrieve relation, in the PVS logic, that has been used to demonstrate that Simpson's 4-slot ACM is a refinement of the abstract model of atomicity, using Nipkow's retrieve rule, as described in Chapter 5.

```
Retrieve: THEORY
 BEGIN

  IMPORTING Abstract_Protocol, FOUR_SLOT
```

The retrieve relation is in four parts: the first part describes the relation between the two models when the neither the reader and writer are accessing the ACM. In this case the reader and writer program counters point to the first operations in the read and write algorithms (they are equal to *rcp* and *wcp* respectively, which mean that the next action will be either *readerChoosesPair* in the model of the 4-slot and *startRd* in the abstract model, or *writerChoosesPair* in the model of the 4-slot and *startWr* in the abstract model). Since the writer and reader are not accessing the ACM the local copies of the control variables will be equal in value to the relevant control variables in the ACM, the item at the head of the sequence of values will be the last one written (pointed to by the writer local variables *writerPair* and *writerSlot*)and the length of the sequence will be at least 1.

```
  R(as: Abs_State, cs: Conc_State): bool =
   (¬ as`readerAccess ∧ ¬ as`writerAccess ⇒
     cs`nri = rcp ∧
      cs`nwi = wcp ∧
       cs`writer`writerPair = cs`pairWritten ∧
        cs`writer`writerSlot = cs`slotWritten(cs`writer`writerPair) ∧
         cs`reader`readerPair = cs`pairReading ∧
          cs`slots(cs`writer`writerPair, cs`writer`writerSlot) = as`vals`seq(0) ∧
           as`vals`length ≥ 1) ∧
```

The second part of the retrieve relation describes the relation between the two models when only the writer is accessing the ACM. The next reader

and writer actions in the implementation will be *rcp* and *wip* respectively, the value of the writer control variable *writerSlot* will be equal to the value of the element of the *slotWritten* appropriate to the pair of slots the writer is accessing and the reader local variable *readerPair* will be equal to the *pairReading* control variable. The item at the head of the sequence will be the item the writer has added during the current write and will be pointed to by the writer local variables. If the writer has changed pairs for this write the reader cannot in any circumstances read the item at the head of the sequence until after the writer has executed *endWr* and indicated the pair of slots it has accessed: the sequence must therefore be of length greater than 1 (the last item written and the item added by the current write must both be present in the sequence). Since, in this case, the writer has changed pairs: the reader must have indicated that it had changed pairs to read from the same pair as the writer before the start of the current write, *pairReading* is therefore equal to *pairWritten*; and the writer local variable *writerPair* will not be equal to *pairWritten*. In addition the item written by the last write will be the second item in the sequence. If the writer has not changed pairs the writer local variable *writerPair* will be equal to the control variable *pairWritten* and the sequence must be at least of length 1 (it is possible for a complete read to occur during the write; for the reader to access the item that has been written by the write during the current write; and therefore the sequence to be shortened to contain only that single item - the item at the head of the sequence).

```
(¬ as'readerAccess ∧ as'writerAccess ⇒
  cs'nri = rcp ∧
   cs'nwi = wip ∧
    cs'writer'writerSlot = cs'slotWritten(cs'writer'writerPair) ∧
     cs'reader'readerPair = cs'pairReading ∧
      cs'slots(cs'writer'writerPair, cs'writer'writerSlot) = as'vals'seq(0) ∧
       (cs'writerChangedPairNI ⇒
         as'vals'length > 1 ∧
          cs'pairReading = cs'pairWritten ∧
           ¬ cs'pairWritten = cs'writer'writerPair ∧
            cs'slots(cs'pairWritten, cs'slotWritten(cs'pairWritten)) = as'vals'seq(1)) ∧
            (¬ cs'writerChangedPairNI ⇒ cs'pairWritten = cs'writer'writerPair) ∧
             as'vals'length ≥ 1) ∧
```

The third part of the retrieve relation relates the states of the two models when only the reader is accessing the ACM. The program counters will be equal to *rip* and *wcp*, and all of the local variables will be equal to the relevant control variables (in the reader's case, because it has indicated the pair it is reading from during *startRd*). The sequence must be of length at least 1, the last item written will be at the head of the sequence and there will be an item on the sequence equivalent to the one that the reader has chosen to read in the model of the implementation.

```
(as'readerAccess ∧ ¬ as'writerAccess ⇒
```

```
cs'nri = rd ∧
 cs'nwi = wcp ∧
  cs'pairWritten = cs'writer'writerPair ∧
   cs'writer'writerSlot = cs'slotWritten(cs'writer'writerPair) ∧
    cs'reader'readerPair = cs'pairReading ∧
     cs'slots(cs'writer'writerPair, cs'writer'writerSlot) = as'vals'seq(0) ∧
      (∃ (i: nat):
         i < as'vals'length ∧
          cs'slots(cs'reader'readerPair, cs'reader'readerSlot) = as'vals'seq(i) ∧
           as'vals'length ≥ 1)) ∧
```

The final part of the retrieve relation relates the states in the two models when the reader and writer are both accessing the mechanism. This is effectively a combination of the two parts where only one of the reader and writer processes is accessing the mechanism. The program counters will be equal to *rd* and *wip*, the local variable *writerSlot* will be equal to the relevant element of *slotWritten*, the local variable *readerPair* will be equal to *pairReading*, and item written by the current write will be on the head of the sequence. If the writer has changed pairs the local variable *writerPair* will not be equal to the control variable *pairWritten*, the control variables *pairReading* and *pairWritten* will be equal, the item written by the last write will be the second on the sequence, the sequence will be of length greater than 1, and there will be an item on the sequence which is equal to the item chosen by the reader in the implementation. If the writer has not changed pairs the writer local variable writerPair will be equal to the control variable *pairWritten*, there will be an item on the sequence equal to the one chosen by the reader in the model of the implementation and the sequence length will be at least 1.

```
(as'readerAccess ∧ as'writerAccess ⇒
  cs'nri = rd ∧
   cs'nwi = wip ∧
    cs'writer'writerSlot = cs'slotWritten(cs'writer'writerPair) ∧
     cs'reader'readerPair = cs'pairReading ∧
      cs'slots(cs'writer'writerPair, cs'writer'writerSlot) = as'vals'seq(0) ∧
       (cs'writerChangedPairNI ⇒
         as'vals'length > 1 ∧
          cs'pairReading = cs'pairWritten ∧
           ¬ cs'pairWritten = cs'writer'writerPair ∧
            cs'slots(cs'pairWritten, cs'slotWritten(cs'pairWritten)) = as'vals'seq(1) ∧
             (∃ (i: nat): i > 0 ∧
                i < as'vals'length ∧
                 cs'slots(cs'reader'readerPair, cs'reader'readerSlot) = as'vals'seq(i))) ∧
       (¬ cs'writerChangedPairNI ⇒
         cs'pairWritten = cs'writer'writerPair ∧
          (∃ (i: nat):
             i < as'vals'length ∧
              cs'slots(cs'reader'readerPair, cs'reader'readerSlot) = as'vals'seq(i)))
               ∧ as'vals'length ≥ 1)
```

The PVS encoding of the proof obligations: first the domain proofs; that if the pre-condition for an operation holds in the abstract model the equivalent pre-condition will also hold in the model of the implementation.

```
dom_start_write: THEOREM
  ∀ (cs: Conc_State, as: Abs_State):
    R(as, cs) ∧ pre_start_write(as) ⇒ pre_startWr(cs)

dom_end_write: THEOREM
  ∀ (cs: Conc_State, as: Abs_State):
    R(as, cs) ∧ pre_end_write(as) ⇒ pre_endWr(cs)

dom_start_read: THEOREM
  ∀ (cs: Conc_State, as: Abs_State):
    R(as, cs) ∧ pre_start_read(as) ⇒ pre_startRd(cs)

dom_end_read: THEOREM
  ∀ (cs: Conc_State, as: Abs_State):
    R(as, cs) ∧ pre_end_read(as) ⇒ pre_endRd(cs)
```

The result proof obligations. If it is possible to relate states in the two models using the retrieve relation, and the pre-condition holds for the operation in the abstract model, then if the operation in the implementation (that is equivalent to abstract operation for which the pre-condition is enables) is executed; it is possible to find a state in the abstract model, such that it is possible to execute the equivalent operation in the abstract model and the retrieve relation holds between this state and the target state of the transition associated with operation executed in the model of the implementation.

```
res_start_read: THEOREM
  ∀ (cs, cs1: Conc_State, as: Abs_State):
    R(as, cs) ∧ pre_start_read(as) ∧ post_startRd(cs)(cs1) ⇒
     (∃ (as1: Abs_State): R(as1, cs1) ∧ post_start_read(as)(as1))

res_end_read: THEOREM
  ∀ (cs, cs1: Conc_State, as: Abs_State, v: Val):
    R(as, cs) ∧ pre_end_read(as) ∧ post_endRd(cs)(cs1, v) ⇒
     (∃ (as1: Abs_State): R(as1, cs1) ∧ post_end_read(as)(as1, v))

res_start_write: THEOREM
  ∀ (cs, cs1: Conc_State, as: Abs_State):
    R(as, cs) ∧ pre_start_write(as) ∧ post_startWr(cs)(cs1) ⇒
     (∃ (as1: Abs_State): R(as1, cs1) ∧ post_start_write(as)(as1))

res_end_write: THEOREM
  ∀ (cs, cs1: Conc_State, as: Abs_State):
    R(as, cs) ∧ pre_end_write(as) ∧ post_endWr(cs)(cs1) ⇒
     (∃ (as1: Abs_State): R(as1, cs1) ∧ post_end_write(as)(as1))
END Retrieve
```

Appendix G

Proof of Coherence

The model of the 4-slot implementation given in this appendix is the same as the one given in Appendix D, except that there are a number of additional auxiliary variables. These additional variables are required to verify that the ACM transmits coherent data between its reader and writer, when the reader and writer operations are executed atomically, but can interleave in an unrestricted manner, using a compositional proof method for shared variable concurrency, based on the rely-guarantee method given in [dR+01].

```
FOUR_SLOT: THEORY
 BEGIN
```

The ACM transmits data items, consisting of a value and an index number, between its reader and writer.

```
  Val: NONEMPTY_TYPE

  Data: TYPE = [# index: nat, val: Val #]
```

Types to represent the names of the pairs and slots in the ACM.

```
  PairIndex: TYPE = {p0, p1}

  SlotIndex: TYPE = {s0, s1}
```

The program counters, which record the next operation (instruction) to be executed by the reader and writer.

```
  NextReadInstruction: TYPE = {firstRcp, rcp, rip, rcs, rd}

  NextWriteInstruction: TYPE = {firstWcp, wcp, wcs, wr, wis, wip}
```

Types to record the current locations of the reader and writer in their respective assertion networks.

```
  ReaderNetworkState: TYPE = {sr, lr1, lr2, lr3, lr4, tr}

  WriterNetworkState: TYPE = {sw, lw1, lw2, lw3, lw4, lw5, tw}
```

The local state of the writer, which has an auxiliary variable, *currentState*, to record its current location in its assertion network.

```
WriterState: TYPE =
[# writerPair: PairIndex,
   writerSlot: SlotIndex,
   currentState: WriterNetworkState #]
```

The local state of the reader, which also has an auxiliary variable to record its location in its assertion network.

```
ReaderState: TYPE =
[# readerPair: PairIndex,
   readerSlot: SlotIndex,
   currentState: ReaderNetworkState #]
```

The state of the ACM, which has auxiliary variables called *wisOccurred* and *rcsSinceWis*, which are used to reason about the ordering of the writer operation *writerIndicatesSlot* and the reader operation *readerChoosesSlot*. This ordering can affect the slot that the reader accesses during a particular read. It also introduces the auxiliary variable *maxFresh*, which is used in the proof of atomicity (which will be described in Appendix H).

```
Conc_State: TYPE =
[# pairWritten: PairIndex,
   slotWritten: [PairIndex → SlotIndex],
   lastSlotWritten: [PairIndex → SlotIndex],
   pairReading: PairIndex,
   slots: [PairIndex, SlotIndex → Data],
   nri: NextReadInstruction,
   nwi: NextWriteInstruction,
   writer: WriterState,
   reader: ReaderState,
   wisOccurred: bool,
   rcsSinceWis: bool,
   maxFresh: nat #]
```

Each of the operations implements one of the actions of the 4-slot algorithm, from Table 3.4, for either the reader or the writer, and sets the program counter equal to the next operation to be executed (for example *readerChoosesPair* sets *nri* equal to *rip* - *readerIndicatesPair* which is the next operation the reader will execute). The operations also set the current state of the reader, or writer, to the next state in their respective assertion networks.

The initial *readerChoosespair* operation is executed once at start up. The *readerChoosesPair* operation is identical, but is executed during each read after the first one. These operations choose the pair of slots in the mechanism that the reader is going to access during the current read.

```
pre_firstReaderChoosesPair(p: Conc_State): bool = p'nri = firstRcp

post_firstReaderChoosesPair(p: (pre_firstReaderChoosesPair))(prot: Conc_State): bool =
    prot = p WITH [nri := rip,
```

```
                    reader := p'reader WITH [readerPair := p'pairWritten,
                                             currentState := lr1]]

firstReaderChoosesPair:
[p: (pre_firstReaderChoosesPair) → (post_firstReaderChoosesPair(p))]

pre_readerChoosesPair(p: Conc_State): bool = p'nri = rcp

post_readerChoosesPair(p: (pre_readerChoosesPair))(prot: Conc_State): bool =
    prot = p WITH [nri := rip,
                   reader := p'reader WITH [readerPair := p'pairWritten,
                                            currentState := lr1]]

readerChoosesPair:
[p: (pre_readerChoosesPair) → (post_readerChoosesPair(p))]

pre_readerIndicatesPair(p: Conc_State): bool = p'nri = rip

post_readerIndicatesPair(p: (pre_readerIndicatesPair))(prot: Conc_State): bool =
    prot = p WITH [nri := rcs,
                   pairReading := p'reader'readerPair,
                   reader := p'reader WITH [currentState := lr2]]
```

readerIndicatesPair sets the control variable *pairReading* equal to the pair the reader is accessing.

```
readerIndicatesPair:
[p: (pre_readerIndicatesPair) → (post_readerIndicatesPair(p))]

pre_readerChoosesSlot(p: Conc_State): bool = p'nri = rcs

post_readerChoosesSlot(p: (pre_readerChoosesSlot))(prot: Conc_State): bool =
    prot = p WITH [nri := rd,
                   reader := p'reader WITH [readerSlot :=
                               p'slotWritten(p'reader'readerPair), currentState := lr3],
                                            rcsSinceWis := TRUE]
```

The *readerChoosesSlot* operation chooses the slot the reader is going to access - by setting *readerSlot* equal to the value of the element of the *slotWritten* array for the pair the reader is accessing. It also sets the auxiliary variable *rcsSinceWis* to true. This variable is used, with the auxiliary variable *wisOccurred*, to help decide whether the reader accesses the slot he writer has just accessed during the current write (when a read and write occur concurrently and the reader and writer access the same pair of slots) as described in Section 6.3.3.

```
readerChoosesSlot:
[p: (pre_readerChoosesSlot) → (post_readerChoosesSlot(p))]

pre_read(p: Conc_State): bool = p'nri = rd

post_read(p: (pre_read))(prot: Conc_State, v: Val): bool =
    v = p'slots(p'reader'readerPair, p'reader'readerSlot)'val ∧
     prot = p WITH [nri := rcp, reader := p'reader WITH [currentState := lr4]]
```

During the *read* operation the reader accesses the chosen slot and returns the value read.

```
read: [p: (pre_read) → (post_read(p))]
```

The *firstWriterChoosesPair* and *writerChoosesPair* operations are identical, but *firstWriterChoosesPair* is executed once at start up and *writerChoosesPair* is executed thereafter. The operations set the pair the writer is going to access (*writerPair*) equal to the opposite to the one the reader last indicated it was accessing (*pairReading*).

```
pre_firstWriterChoosesPair(p: Conc_State): bool = p'nwi = firstWcp

post_firstWriterChoosesPair(p: (pre_firstWriterChoosesPair))(prot: Conc_State): bool =
    (p'pairReading = p0 ⇒
      prot = p WITH [nwi := wcs,
                     writer := p'writer WITH [writerPair := p1, currentState := lw1],
                     maxFresh := p'maxFresh + 1]) ∧
     (p'pairReading = p1 ⇒
       prot = p WITH [nwi := wcs,
                      writer := p'writer WITH [writerPair := p0, currentState := lw1],
                      maxFresh := p'maxFresh + 1])

firstWriterChoosesPair:
[p: (pre_firstWriterChoosesPair) → (post_firstWriterChoosesPair(p))]

pre_writerChoosesPair(p: Conc_State): bool = p'nwi = wcp

post_writerChoosesPair(p: (pre_writerChoosesPair))(prot: Conc_State): bool =
    (p'pairReading = p0 ⇒
      prot = p WITH [nwi := wcs,
                     writer := p'writer WITH [writerPair := p1, currentState := lw1],
                     maxFresh := p'maxFresh + 1,
                     wisOccurred := FALSE]) ∧
     (p'pairReading = p1 ⇒
       prot = p WITH [nwi := wcs,
                      writer := p'writer WITH [writerPair := p0, currentState := lw1],
                      maxFresh := p'maxFresh + 1,
                      wisOccurred := FALSE])

writerChoosesPair:
[p: (pre_writerChoosesPair) → (post_writerChoosesPair(p))]
```

The *writerChoosesSlot* operation chooses the slot the writer is going to access during the *write* operation. The writer chooses the opposite slot, in the pair it is accessing, to the one it accessed during the last write.

```
pre_writerChoosesSlot(p: Conc_State): bool = p'nwi = wcs

post_writerChoosesSlot(p: (pre_writerChoosesSlot))(prot: Conc_State): bool =
    (p'slotWritten(p'writer'writerPair) = s0 ⇒
      prot = p WITH [nwi := wr, writer := p'writer
                                    WITH [writerSlot := s1, currentState := lw2]]) ∧
     (p'slotWritten(p'writer'writerPair) = s1 ⇒
       prot = p WITH [nwi := wr, writer := p'writer
                                     WITH [writerSlot := s0, currentState := lw2]])

writerChoosesSlot:
[p: (pre_writerChoosesSlot) → (post_writerChoosesSlot(p))]
```

The *write* operation adds the new item to the slot that the writer has chosen to access.

```
pre_write(p: Conc_State): bool = p'nwi = wr

write_parameter: TYPE = [# p1: (pre_write), v: Val #]

post_write(p: write_parameter)(prot: Conc_State): bool =
    prot = p'p1 WITH [nwi := wis,
                      (slots)(p'p1'writer'writerPair, p'p1'writer'writerSlot)
                          := (# index := p'p1'maxFresh, val := p'v #),
                      writer := p'p1'writer WITH [currentState := lw3]]

write: [p: write_parameter → (post_write(p))]
```

writerIndicatesSlot sets the appropriate element of the *slotWritten* array equal to the slot that the writer has just accessed during the *write* operation for the pair it is accessing.

```
pre_writerIndicatesSlot(p: Conc_State): bool = p'nwi = wis

post_writerIndicatesSlot(p: (pre_writerIndicatesSlot))(prot: Conc_State): bool =
    prot = p WITH [nwi := wip,
                   (slotWritten)(p'writer'writerPair) := (p'writer'writerSlot),
                   writer := p'writer WITH [currentState := lw4],
                   wisOccurred := TRUE,
                   rcsSinceWis := FALSE]

writerIndicatesSlot:
[p: (pre_writerIndicatesSlot) → (post_writerIndicatesSlot(p))]
```

The *writerIndicatesPair* operation sets the *pairWritten* control variable equal to the pair that the writer has accessed during the current write (equal to the writer local variable *writerPair*).

```
pre_writerIndicatesPair(p: Conc_State): bool = p'nwi = wip

post_writerIndicatesPair(p: (pre_writerIndicatesPair))(prot: Conc_State): bool =
    prot = p WITH [nwi := wcp,
                   pairWritten := p'writer'writerPair,
                   writer := p'writer WITH [currentState := lw5]]

writerIndicatesPair:
[p: (pre_writerIndicatesPair) → (post_writerIndicatesPair(p))]
```

The initialisation operations for the model. The reader and writer both start in the initial locations of their respective assertion networks but their remaining initialisation values are unimportant, because the components both choose a pair and slot to access before they access the ACM on each occasion. In the case of the ACM itself one slot is initialised with an initial value, and the *pairWritten* and *slotWritten* control variable are set to point to this slot.

```
init_writer(w: WriterState): bool =
    w = w WITH [writerPair := p0, writerSlot := s0, currentState := sw]

init_reader(r: ReaderState): bool =
    r = r WITH [readerPair := p1, readerSlot := s1, currentState := sr]

init_data(init_data: Data, init_val: Val): bool =
```

```
    init_data = init_data WITH [index := 0, val := init_val]

init_prot(p: Conc_State, init_data: Data, w: WriterState, r: ReaderState): bool =
    p = p WITH [pairWritten := p0,
                slotWritten := ((λ ·(p0: PairIndex): s0),
                pairReading := p0,
                slots := ((λ ·(p0: PairIndex, s0: SlotIndex): init_data),
                nri := firstRcp,
                nwi := firstWcp,
                writer := w,
                reader := r,
                wisOccurred := FALSE,
                rcsSinceWis := FALSE,
                maxFresh := 0]
```

The assertions for the reader and writer assertion networks are given and described below. It is not necessary to make any assertions in the locations in the reader network where the reader is about to execute *firstReaderChoosesPair*, *readerChoosesPair* or *readerIndicatesPair*, since the relationship between the control variables in the mechanism and the reader local state that is required to verify coherence is established by the *readerIndicatesPair* and *readerChoosesSlot* operations.

First when the reader is about to execute the *readerChoosesSlot* operation the control variable *pairReading* will be equal to the reader local variable *pairReading*.

```
readerChoosesSlot_Assertion: [Conc_State → bool] =
    (λ ·(cs: Conc_State):
      cs'nri = rcs ⇒ cs'pairReading = cs'reader'readerPair
```

When the reader is about to execute the *read* operation it has already indicated the pair it is going to access, so the reader local variable *readerPair* is equal to the control variable *pairReading*. The remainder of the assertion is required to establish that the reader accesses a different slot to the writer when they are both accessing the same pair of slots at the same time: if they are accessing different pairs they are, by definition, accessing different slots. If the writer starts a new write when the reader is about to read the data from the ACM, the writer will change pairs, since the reader has already indicated the pair of slots it is going to read. It is, therefore, only necessary to reason about the relationship between the control variables and the reader local variables when the reader and writer are accessing the same pair of slots in the mechanism. There are three different cases to consider:

1. If the writer has not yet indicated the slot is is going to access (*wisOccurred* = false), the the reader local variable *readerSlot* will be equal to the element of the *slotWritten* array for the pair the reader is accessing (since the reader's last action was to choose the slot it was going to access in its current pair).

2. If the writer has indicated the slot it is using during the current write and the reader chose its slot before the writer executed *writerIndicatedSlot* (*wisOccurred* = true ∧ *rcsSinceWis* = false) the reader will access the opposite slot to the writer (since the writer chooses the opposite slot to the one it accessed the last time in the current pair, and the reader chooses to read from the slot the writer indicated it accessed the during the last write).

3. If the reader chooses the slot it is going to access after the writer executes *writerIndicatesSlot* (*wisOccurred* = true∧*rcsSinceWis* = true) it will access the slot the writer has written data to during the current write. This is fine, because the writer has finished accessing the slot to write the data before it executes *writerIndicatesSlot*. In a sense the reader manages to read the item of data before it has been fully released by the writer.

```
read_Assertion: [Conc_State → bool] =
    (λ ·(cs: Conc_State):
      cs'nri = rd ⇒
       cs'pairReading = cs'reader'readerPair ∧
        (cs'reader'readerPair = cs'writer'writerPair ⇒
          (¬ cs'wisOccurred ⇒ cs'reader'readerSlot = cs'slotWritten(cs'reader'readerPair)) ∧
           (cs'wisOccurred ⇒
             (cs'rcsSinceWis ⇒ cs'reader'readerSlot = cs'slotWritten(cs'reader'readerPair)) ∧
              (¬ cs'rcsSinceWis ⇒
                ¬ cs'reader'readerSlot =
                   cs'slotWritten(cs'reader'readerPair))))
```

When the writer is about to execute *firstWriterChoosesPair* and *writerChoosesSlot* it is only necessary to assert that it has not yet indicated the pair it is accessing during the write (¬ *wisOccurred*).

```
firstWriterChoosesPair_Assertion: [Conc_State → bool] =
    (λ ·(cs: Conc_State): cs'nwi = firstWcp ⇒ ¬ cs'wisOccurred

writerChoosesSlot_Assertion: [Conc_State → bool] =
    (λ ·(cs: Conc_State): cs'nwi = wcs ⇒ ¬ cs'wisOccurred
```

When the writer is accessing the data slot in the ACM , and before it executes *writerIndicatesSlot* it has chosen to access the opposite slot to the one it accessed during the last write in the pair it is currently accessing. It has not yet executed *writerIndicatesSlot*, so *wisOccurred* is still false.

```
write_Assertion: [Conc_State → bool] =
    (λ ·(cs: Conc_State):
      cs'nwi = wr ⇒
       ¬ cs'wisOccurred ∧
        ¬ cs'writer'writerSlot = cs'slotWritten(cs'writer'writerPair)

writerIndicatesSlot_Assertion: [Conc_State → bool] =
    (λ ·(cs: Conc_State):
      cs'nwi = wis ⇒
       ¬ cs'wisOccurred ∧
        ¬ cs'writer'writerSlot = cs'slotWritten(cs'writer'writerPair)
```

When the writer is about to execute *writerIndicatesPair* it has executed *writerIndicatesSlot* and the local variable *slotWritten* is equal to the element of the *slotWritten* array relating to the pair of slots the writer is accessing.

```
writerIndicatesPair_Assertion: [Conc_State → bool] =
    (λ  · (cs: Conc_State):
      cs'nwi = wip ⇒
       cs'wisOccurred ∧
        cs'writer'writerSlot = cs'slotWritten(cs'writer'writerPair)
```

The proof obligations follow. The first is to show that the initialisation of the ACM establishes the *firstWriterChoosesPair* assertion There is no assertion for *firstReaderChoosesPair* so there is no equivalent proof for the reader.

```
vc_initWriter: THEOREM
  ∀ (cs: Conc_State, init: Data, w: WriterState, r: ReaderState):
    init_prot(cs, init, w, r) ⇒ firstWriterChoosesPair_Assertion(cs)
```

The first proof obligation for each of the remaining locations in the reader and writer networks (*vc1_op_name*) is to establish for each transition in the respective networks that:

1. If the assertion in the start location of the transition associated with each operation holds, and the transition is enabled, that the assertion in the target location of the transition will hold after executing the operation that is associated with the transition. In the case of the four slot the guards for each of the transitions is effectively true i.e. the transition is enabled whenever the component is in the start location of the transition (since the pre-condition for the operation is simply that the program counter for the component is such that the operation is to be executed next).

2. That each of the components does not interfere with the assertions in the network of the other component e.g. if the assertions in the locations of the network of the other component hold before the operation is executed, they will still hold after the operation is executed.

This requires the following proof obligation to be completed for every location in the network of the writer: *vc1_op_name*

$$
\begin{aligned}
&\forall cs1, cs2 : Conc\_State \cdot \\
&\quad pre\_start\_writer\_op\_name(cs1) \wedge \\
&\quad\quad start\_writer\_op\_name\_Assertion(cs1) \wedge \\
&\quad\quad\quad readerChoosesSlot\_Assertion(cs1) \wedge \\
&\quad\quad\quad\quad read\_Assertion(cs1) \wedge post\_writer\_op\_name(cs1, cs2) \Rightarrow \\
&\quad\quad\quad\quad\quad cs2.nwi = targetLocationInstruction \wedge \\
&\quad\quad\quad\quad\quad\quad target\_writer\_op\_name\_Assertion(cs2) \wedge \\
&\quad\quad\quad\quad\quad\quad\quad readerChoosesSlot\_Assertion(cs2) \wedge
\end{aligned}
$$

$$read\_Assertion(cs2)$$

Similarly the following proof obligation must be completed for every location in the network of the reader:

$vc1\_op\_name$

$$\forall cs1, cs2 : Conc\_State \cdot$$
$$pre\_start\_reader\_op\_name(cs1) \wedge$$
$$start\_reader\_op\_name\_Assertion(cs1) \wedge$$
$$firstWriterChoosesPair\_Assertion(cs1) \wedge$$
$$writerChoosesSlot\_Assertion(cs1) \wedge$$
$$write\_Assertion(cs1) \wedge$$
$$writerIndicatesSlot\_Assertion(cs1) \wedge$$
$$writerIndicatesPair\_Assertion(cs1) \wedge$$
$$post\_reader\_op\_name(cs1, cs2) \Rightarrow$$
$$cs2`nri = targetLocationInstruction \wedge$$
$$target\_reader\_op\_name\_Assertion(cs2) \wedge$$
$$firstWriterChoosesPair\_Assertion(cs2) \wedge$$
$$writerChoosesSlot\_Assertion(cs2) \wedge$$
$$write\_Assertion(cs2) \wedge$$
$$writerIndicatesSlot\_Assertion(cs2) \wedge$$
$$writerIndicatesPair\_Assertion(cs2)$$

```
vc1_firstWriterChoosesPair: THEOREM
  ∀ (cs1, cs2: Conc_State):
    pre_firstWriterChoosesPair(cs1) ∧
     firstWriterChoosesPair_Assertion(cs1) ∧
      (readerChoosesSlot_Assertion(cs1)) ∧
       (read_Assertion(cs1)) ∧ cs2 = firstWriterChoosesPair(cs1) ⇒
     cs2`nwi = wcs ∧
      (writerChoosesSlot_Assertion(cs2)) ∧
       (readerChoosesSlot_Assertion(cs2)) ∧ (read_Assertion(cs2))

vc1_writerChoosesPair: THEOREM
  ∀ (cs1, cs2: Conc_State):
    pre_writerChoosesPair(cs1) ∧
     (readerChoosesSlot_Assertion(cs1)) ∧ (read_Assertion(cs1)) ∧ cs2 = writerChoosesPair(cs1) ⇒
     cs2`nwi = wcs ∧
      (writerChoosesSlot_Assertion(cs2)) ∧
       (readerChoosesSlot_Assertion(cs2)) ∧ (read_Assertion(cs2))

vc1_writerChoosesSlot_lw1: THEOREM
  ∀ (cs1, cs2: Conc_State):
    pre_writerChoosesSlot(cs1) ∧
     (writerChoosesSlot_Assertion(cs1)) ∧
      (readerChoosesSlot_Assertion(cs1)) ∧ (read_Assertion(cs1)) ∧ cs2 = writerChoosesSlot(cs1) ⇒
     cs2`nwi = wr ∧
      (write_Assertion(cs2)) ∧
       (readerChoosesSlot_Assertion(cs2)) ∧ (read_Assertion(cs2))

vc1_write_lw2: THEOREM
  ∀ (w: write_parameter, cs2: Conc_State):
    pre_write(w`p1) ∧
     (write_Assertion(w`p1)) ∧
      (readerChoosesSlot_Assertion(w`p1)) ∧ (read_Assertion(w`p1)) ∧ cs2 = write(w) ⇒
     cs2`nwi = wis ∧
```

```
      cs2'slots(cs2'writer'writerPair, cs2'writer'writerSlot)'val = w'v ∧
       (writerIndicatesSlot_Assertion(cs2)) ∧
        (readerChoosesSlot_Assertion(cs2)) ∧ (read_Assertion(cs2))

vc1_writerIndicatesSlot_lw3: THEOREM
  ∀ (cs1, cs2: Conc_State):
    pre_writerIndicatesSlot(cs1) ∧
     (writerIndicatesSlot_Assertion(cs1)) ∧
      (readerChoosesSlot_Assertion(cs1)) ∧
       (read_Assertion(cs1)) ∧ cs2 = writerIndicatesSlot(cs1) ⇒
     cs2'nwi = wip ∧
      (writerIndicatesPair_Assertion(cs2)) ∧
       (readerChoosesSlot_Assertion(cs2)) ∧ (read_Assertion(cs2))

vc1_writerIndicatesPair_lw4: THEOREM
  ∀ (cs1, cs2: Conc_State):
    pre_writerIndicatesPair(cs1) ∧
     (writerIndicatesPair_Assertion(cs1)) ∧
      (readerChoosesSlot_Assertion(cs1)) ∧
       (read_Assertion(cs1)) ∧ cs2 = writerIndicatesPair(cs1) ⇒
     cs2'nwi = wcp ∧
      (readerChoosesSlot_Assertion(cs2)) ∧ (read_Assertion(cs2))

vc1_firstReaderChoosesPair: THEOREM
  ∀ (cs1, cs2: Conc_State):
    pre_firstReaderChoosesPair(cs1) ∧
     firstWriterChoosesPair_Assertion(cs1) ∧
      (writerChoosesSlot_Assertion(cs1)) ∧
       (write_Assertion(cs1)) ∧
        (writerIndicatesSlot_Assertion(cs1)) ∧
         (writerIndicatesPair_Assertion(cs1)) ∧ cs2 = firstReaderChoosesPair(cs1) ⇒
     cs2'nri = rip ∧
      firstWriterChoosesPair_Assertion(cs2) ∧
       (writerChoosesSlot_Assertion(cs2)) ∧
        (write_Assertion(cs2)) ∧
         (writerIndicatesSlot_Assertion(cs2)) ∧
          (writerIndicatesPair_Assertion(cs2))

vc1_readerChoosesPair: THEOREM
  ∀ (cs1, cs2: Conc_State):
    pre_readerChoosesPair(cs1) ∧
     firstWriterChoosesPair_Assertion(cs1) ∧
      (writerChoosesSlot_Assertion(cs1)) ∧
       (write_Assertion(cs1)) ∧
        (writerIndicatesSlot_Assertion(cs1)) ∧
         (writerIndicatesPair_Assertion(cs1)) ∧ cs2 = readerChoosesPair(cs1) ⇒
     cs2'nri = rip ∧
      firstWriterChoosesPair_Assertion(cs1) ∧
       (writerChoosesSlot_Assertion(cs2)) ∧
        (write_Assertion(cs2)) ∧
         (writerIndicatesSlot_Assertion(cs2)) ∧
          (writerIndicatesPair_Assertion(cs2))

vc1_readerIndicatesPair: THEOREM
  ∀ (cs1, cs2: Conc_State):
    pre_readerIndicatesPair(cs1) ∧
     firstWriterChoosesPair_Assertion(cs1) ∧
      (writerChoosesSlot_Assertion(cs1)) ∧
       (write_Assertion(cs1)) ∧
        (writerIndicatesSlot_Assertion(cs1)) ∧
         (writerIndicatesPair_Assertion(cs1)) ∧ cs2 = readerIndicatesPair(cs1) ⇒
     cs2'nri = rcs ∧
      readerChoosesSlot_Assertion(cs2) ∧
       firstWriterChoosesPair_Assertion(cs2) ∧
```

```
        (writerChoosesSlot_Assertion(cs2)) ∧
         (write_Assertion(cs2)) ∧
          (writerIndicatesSlot_Assertion(cs2)) ∧
           (writerIndicatesPair_Assertion(cs2))

vc1_readerChoosesSlot: THEOREM
  ∀ (cs1, cs2: Conc_State):
    pre_readerChoosesSlot(cs1) ∧
     (readerChoosesSlot_Assertion(cs1)) ∧
      firstWriterChoosesPair_Assertion(cs1) ∧
       (writerChoosesSlot_Assertion(cs1)) ∧
        (write_Assertion(cs1)) ∧
         (writerIndicatesSlot_Assertion(cs1)) ∧
          (writerIndicatesPair_Assertion(cs1)) ∧ cs2 = readerChoosesSlot(cs1) ⇒
     cs2'nri = rd ∧
      cs2'rcsSinceWis ∧
       read_Assertion(cs2) ∧
        firstWriterChoosesPair_Assertion(cs1) ∧
         (writerChoosesSlot_Assertion(cs2)) ∧
          (write_Assertion(cs2)) ∧
           (writerIndicatesSlot_Assertion(cs2)) ∧
            (writerIndicatesPair_Assertion(cs2))

vc1_read: THEOREM
  ∀ (cs1, cs2: Conc_State, v: Val):
    pre_read(cs1) ∧
     (read_Assertion(cs1)) ∧
      firstWriterChoosesPair_Assertion(cs1) ∧
       (writerChoosesSlot_Assertion(cs1)) ∧
        (write_Assertion(cs1)) ∧
         (writerIndicatesSlot_Assertion(cs1)) ∧
          (writerIndicatesPair_Assertion(cs1)) ∧ v = read(cs1)'2 ∧ cs2 = read(cs1)'1 ⇒
     cs2'nri = rcp ∧
      firstWriterChoosesPair_Assertion(cs2) ∧
       (writerChoosesSlot_Assertion(cs2)) ∧
        (write_Assertion(cs2)) ∧
         (writerIndicatesSlot_Assertion(cs2)) ∧
          (writerIndicatesPair_Assertion(cs2))
```

The remaining proof obligations are first to show that the required guarantee condition holds in the start location for each transition. In this case it follows immediately that the guarantee condition for the ACM holds since it is identical to the guarantee condition for each of the transitions. In the case of the write the following proof obligations must be discharged:

$vc2\_op\_name$

$$
\begin{aligned}
&\forall cs1 : Conc\_State \cdot \\
&\quad pre\_start\_writer\_op\_name(cs1) \wedge \\
&\quad\ start\_writer\_op\_name\_Assertion(cs1) \wedge \\
&\quad\quad readerChoosesSlot\_Assertion(cs1) \wedge \\
&\quad\quad\ read\_Assertion(cs1) \Rightarrow \\
&\quad\quad\quad (cs1.nri = rd \wedge cs1.nwi = wr \Rightarrow \\
&\quad\quad\quad (\neg\, cs1.reader.readerPair = cs1.writer.writerPair \vee \\
&\quad\quad\quad\ \neg\, cs1.writer.writerSlot = cs1.reader.readerSlot))
\end{aligned}
$$

It is also necessary to show that the guarantee condition holds in the target location of the transition, as follows:

$vc3\_op\_name$
$\forall cs1, cs2 : Conc\_State \cdot$
$pre\_start\_writer\_op\_name(cs1) \wedge$
$start\_writer\_op\_name\_Assertion(cs1) \wedge$
$readerChoosesSlot\_Assertion(cs1) \wedge$
$read\_Assertion(cs1) \wedge$
$post\_writer\_op\_name(cs1, cs2) \Rightarrow$
$(cs2.nri = rd \wedge cs2.nwi = wr \Rightarrow$
$(\neg\, cs2.reader.readerPair = cs2.writer.writerPair \vee$
$\neg\, cs2.writer.writerSlot = cs2.reader.readerSlot))$

Similarly, for the reader, the following two proof obligations must be discharged:

$vc2\_op\_name$
$\forall cs1 : Conc\_State \cdot$
$pre\_start\_reader\_op\_name(cs1) \wedge$
$start\_reader\_op\_name\_Assertion(cs1) \wedge$
$firstWriterChoosesSlot\_Assertion(cs1) \wedge$
$writerChoosesSlot\_Assertion(cs1) \wedge$
$writerChoosesSlot\_Assertion(cs1) \wedge$
$write\_Assertion(cs1) \wedge$
$writerIndicatesSlot\_Assertion(cs1) \wedge$
$writerIndicatesPair\_Assertion(cs1) \Rightarrow$
$(cs1.nri = rd \wedge cs1.nwi = targetLocationInstruction \Rightarrow$
$(\neg\, cs1.reader.readerPair = cs1.writer.writerPair \vee$
$\neg\, cs1.writer.writerSlot = cs1.reader.readerSlot))$

$vc3\_op\_name$
$\forall cs1, cs2 : Conc\_State \cdot$
$pre\_start\_reader\_op\_name(cs1) \wedge$
$start\_reader\_op\_name\_Assertion(cs1) \wedge$
$firstWriterChoosesPair\_Assertion(cs1) \wedge$
$writerChoosesSlot\_Assertion(cs1) \wedge$
$write\_Assertion(cs1) \wedge$
$writerIndicatesSlot\_Assertion(cs1) \wedge$
$writerIndicatesPair\_Assertion(cs1) \wedge$
$post\_reader\_op\_name(cs1, cs2) \Rightarrow$
$(cs2.nri = rd \wedge cs2.nwi = wr \Rightarrow$
$(\neg\, cs2.reader.readerPair = cs2.writer.writerPair \vee$
$\neg\, cs2.writer.writerSlot = cs2.reader.readerSlot))$

```
vc2_firstWriterChoosesPair: THEOREM
  ∀ (cs1: Conc_State):
    (pre_firstWriterChoosesPair(cs1)) ∧
     firstWriterChoosesPair_Assertion(cs1) ∧
```

```
      (readerChoosesSlot_Assertion(cs1)) ∧
        (read_Assertion(cs1)) ⇒
      (cs1'nri = rd ∧ cs1'nwi = wr ⇒
        (¬ cs1'reader'readerPair = cs1'writer'writerPair) ∨
         (¬ cs1'reader'readerSlot = cs1'writer'writerSlot))

vc3_firstWriterChoosesPair: THEOREM
  ∀ (cs1, cs2: Conc_State):
    pre_firstWriterChoosesPair(cs1) ∧
     firstWriterChoosesPair_Assertion(cs1) ∧
      cs2 = firstWriterChoosesPair(cs1) ∧
       (readerChoosesSlot_Assertion(cs2)) ∧
        (read_Assertion(cs2)) ⇒
     (cs2'nri = rd ∧ cs2'nwi = wr ⇒
       ((¬ cs2'reader'readerPair = cs2'writer'writerPair) ∨
         (¬ cs2'reader'readerSlot = cs2'writer'writerSlot)))

vc2_writerChoosesPair: THEOREM
  ∀ (cs1: Conc_State):
    (pre_writerChoosesPair(cs1)) ∧ (readerChoosesSlot_Assertion(cs1)) ∧
      (read_Assertion(cs1)) ⇒
   (cs1'nri = rd ∧ cs1'nwi = wr ⇒
       (¬ cs1'reader'readerPair = cs1'writer'writerPair) ∨
        (¬ cs1'reader'readerSlot = cs1'writer'writerSlot))

vc3_writerChoosesPair: THEOREM
  ∀ (cs1, cs2: Conc_State):
    pre_writerChoosesPair(cs1) ∧
     cs2 = writerChoosesPair(cs1) ∧ (readerChoosesSlot_Assertion(cs2)) ∧
      (read_Assertion(cs2)) ⇒
     (cs2'nri = rd ∧ cs2'nwi = wr ⇒
       ((¬ cs2'reader'readerPair = cs2'writer'writerPair) ∨
         (¬ cs2'reader'readerSlot = cs2'writer'writerSlot)))

vc2_writerChoosesSlot: THEOREM
  ∀ (cs1: Conc_State):
    pre_writerChoosesSlot(cs1) ∧
     (writerChoosesSlot_Assertion(cs1)) ∧
      (readerChoosesSlot_Assertion(cs1)) ∧ (read_Assertion(cs1)) ⇒
     (cs1'nri = rd ∧ cs1'nwi = wr ⇒
       (¬ cs1'reader'readerPair = cs1'writer'writerPair) ∨
        (¬ cs1'reader'readerSlot = cs1'writer'writerSlot))

vc3_writerChoosesSlot: THEOREM
  ∀ (cs1, cs2: Conc_State):
    pre_writerChoosesSlot(cs1) ∧
     (writerChoosesSlot_Assertion(cs1)) ∧
      cs2 = writerChoosesSlot(cs1) ∧ (readerChoosesSlot_Assertion(cs2)) ∧
        (read_Assertion(cs2)) ⇒
     cs2'nri = rd ∧ cs2'nwi = wr ⇒
      ((¬ cs2'reader'readerPair = cs2'writer'writerPair) ∨
        (¬ cs2'reader'readerSlot = cs2'writer'writerSlot))

vc2_write_lw2: THEOREM
  ∀ (w: write_parameter):
    pre_write(w'p1) ∧
     (write_Assertion(w'p1)) ∧ (readerChoosesSlot_Assertion(w'p1)) ∧
      (read_Assertion(w'p1)) ⇒
   (w'p1'nri = rd ∧ w'p1'nwi = wr ⇒
       ((¬ w'p1'reader'readerPair = w'p1'writer'writerPair) ∨
         (¬ w'p1'reader'readerSlot = w'p1'writer'writerSlot)))

vc3_write_lw2: THEOREM
  ∀ (w: write_parameter, cs2: Conc_State):
```

```
    pre_write(w'p1) ∧
     (write_Assertion(w'p1)) ∧
      cs2 = write(w) ∧ (readerChoosesSlot_Assertion(cs2)) ∧ (read_Assertion(cs2)) ⇒
     cs2'nri = rd ∧ cs2'nwi = wr ⇒
      ((¬ cs2'reader'readerPair = cs2'writer'writerPair) ∨
        (¬ cs2'reader'readerSlot = cs2'writer'writerSlot))

vc2_writerIndicatesSlot_lw3: THEOREM
  ∀ (cs1: Conc_State):
    pre_writerIndicatesSlot(cs1) ∧
     (writerIndicatesSlot_Assertion(cs1)) ∧
      (readerChoosesSlot_Assertion(cs1)) ∧ (read_Assertion(cs1)) ⇒
     (cs1'nri = rd ∧ cs1'nwi = wr ⇒
       ((¬ cs1'reader'readerPair = cs1'writer'writerPair) ∨
         (¬ cs1'reader'readerSlot = cs1'writer'writerSlot)))

vc3_writerIndicatesSlot_lw3: THEOREM
  ∀ (cs1, cs2: Conc_State):
    pre_writerIndicatesSlot(cs1) ∧
     (writerIndicatesSlot_Assertion(cs1)) ∧
      cs2 = writerIndicatesSlot(cs1) ∧
       (readerChoosesSlot_Assertion(cs2)) ∧ (read_Assertion(cs2)) ⇒
     (cs2'nri = rd ∧ cs2'nwi = wr ⇒
       ((¬ cs2'reader'readerPair = cs2'writer'writerPair) ∨
         (¬ cs2'reader'readerSlot = cs2'writer'writerSlot)))

vc2_writerIndicatesPair_cp: THEOREM
  ∀ (cs1: Conc_State):
    pre_writerIndicatesPair(cs1) ∧
     (writerIndicatesPair_Assertion(cs1)) ∧
      (readerChoosesSlot_Assertion(cs1)) ∧ (read_Assertion(cs1)) ⇒
     (cs1'nri = rd ∧ cs1'nwi = wr ⇒
       ((¬ cs1'reader'readerPair = cs1'writer'writerPair) ∨
         (¬ cs1'reader'readerSlot = cs1'writer'writerSlot)))

vc3_writerIndicatesPair_cp: THEOREM
  ∀ (cs1, cs2: Conc_State):
    pre_writerIndicatesPair(cs1) ∧
     (writerIndicatesPair_Assertion(cs1)) ∧
      cs2 = writerIndicatesPair(cs1) ∧
       (readerChoosesSlot_Assertion(cs2)) ∧ (read_Assertion(cs2)) ⇒
     (cs2'nri = rd ∧ cs2'nwi = wr ⇒
       ((¬ cs2'reader'readerPair = cs2'writer'writerPair) ∨
         (¬ cs2'reader'readerSlot = cs2'writer'writerSlot)))

vc2_firstReaderChoosesPair: THEOREM
  ∀ (cs1: Conc_State):
    pre_firstReaderChoosesPair(cs1) ∧
     firstWriterChoosesPair_Assertion(cs1) ∧
      (writerChoosesSlot_Assertion(cs1)) ∧
       (write_Assertion(cs1)) ∧
        (writerIndicatesSlot_Assertion(cs1)) ∧ (writerIndicatesPair_Assertion(cs1)) ⇒
     (cs1'nri = rd ∧ cs1'nwi = wr ⇒
       (¬ cs1'reader'readerPair = cs1'writer'writerPair) ∨
        (¬ cs1'reader'readerSlot = cs1'writer'writerSlot))

vc3_firstReaderChoosesPair: THEOREM
  ∀ (cs1, cs2: Conc_State):
    pre_firstReaderChoosesPair(cs1) ∧
     cs2 = firstReaderChoosesPair(cs1) ∧
      firstWriterChoosesPair_Assertion(cs1) ∧
       (writerChoosesSlot_Assertion(cs2)) ∧
        (write_Assertion(cs2)) ∧
         (writerIndicatesSlot_Assertion(cs2)) ∧ (writerIndicatesPair_Assertion(cs2)) ⇒
```

```
    (cs2'nri = rd ∧ cs2'nwi = wr ⇒
      ((¬ cs2'reader'readerPair = cs2'writer'writerPair) ∨
        (¬ cs2'reader'readerSlot = cs2'writer'writerSlot)))

vc2_readerChoosesPair: THEOREM
  ∀ (cs1: Conc_State):
    pre_readerChoosesPair(cs1) ∧
     firstWriterChoosesPair_Assertion(cs1) ∧
      (writerChoosesSlot_Assertion(cs1)) ∧
       (write_Assertion(cs1)) ∧
        (writerIndicatesSlot_Assertion(cs1)) ∧ (writerIndicatesPair_Assertion(cs1)) ⇒
     (cs1'nri = rd ∧ cs1'nwi = wr ⇒
       (¬ cs1'reader'readerPair = cs1'writer'writerPair) ∨
        (¬ cs1'reader'readerSlot = cs1'writer'writerSlot))

vc3_readerChoosesPair: THEOREM
  ∀ (cs1, cs2: Conc_State):
    pre_readerChoosesPair(cs1) ∧
     cs2 = readerChoosesPair(cs1) ∧
      firstWriterChoosesPair_Assertion(cs1) ∧
       (writerChoosesSlot_Assertion(cs2)) ∧
        (write_Assertion(cs2)) ∧
         (writerIndicatesSlot_Assertion(cs2)) ∧ (writerIndicatesPair_Assertion(cs2)) ⇒
     (cs2'nri = rd ∧ cs2'nwi = wr ⇒
       ((¬ cs2'reader'readerPair = cs2'writer'writerPair) ∨
         (¬ cs2'reader'readerSlot = cs2'writer'writerSlot)))

vc2_readerIndicatesPair: THEOREM
  ∀ (cs1: Conc_State):
    pre_readerIndicatesPair(cs1) ∧
     firstWriterChoosesPair_Assertion(cs1) ∧
      (writerChoosesSlot_Assertion(cs1)) ∧
       (write_Assertion(cs1)) ∧
        (writerIndicatesSlot_Assertion(cs1)) ∧ (writerIndicatesPair_Assertion(cs1)) ⇒
     (cs1'nri = rd ∧ cs1'nwi = wr ⇒
       ((¬ cs1'reader'readerPair = cs1'writer'writerPair) ∨
         (¬ cs1'reader'readerSlot = cs1'writer'writerSlot)))

vc3_readerIndicatesPair: THEOREM
  ∀ (cs1, cs2: Conc_State):
    pre_readerIndicatesPair(cs1) ∧
     cs2 = readerIndicatesPair(cs1) ∧
      firstWriterChoosesPair_Assertion(cs1) ∧
       (writerChoosesSlot_Assertion(cs2)) ∧
        (write_Assertion(cs2)) ∧
         (writerIndicatesSlot_Assertion(cs2)) ∧ (writerIndicatesPair_Assertion(cs2)) ⇒
     (cs2'nri = rd ∧ cs2'nwi = wr ⇒
       ((¬ cs2'reader'readerPair = cs2'writer'writerPair) ∨
         (¬ cs2'reader'readerSlot = cs2'writer'writerSlot)))

vc2_readerChoosesSlot: THEOREM
  ∀ (cs1: Conc_State):
    pre_readerChoosesSlot(cs1) ∧
     (readerChoosesSlot_Assertion(cs1)) ∧
      firstWriterChoosesPair_Assertion(cs1) ∧
       (writerChoosesSlot_Assertion(cs1)) ∧
        (write_Assertion(cs1)) ∧
         (writerIndicatesSlot_Assertion(cs1)) ∧ (writerIndicatesPair_Assertion(cs1)) ⇒
     (cs1'nri = rd ∧ cs1'nwi = wr ⇒
       (¬ cs1'reader'readerPair = cs1'writer'writerPair) ∨
        (¬ cs1'reader'readerSlot = cs1'writer'writerSlot))

vc3_readerChoosesSlot: THEOREM
  ∀ (cs1, cs2: Conc_State):
```

```
    pre_readerChoosesSlot(cs1) ∧
     (readerChoosesSlot_Assertion(cs1)) ∧
      cs2 = readerChoosesSlot(cs1) ∧
       firstWriterChoosesPair_Assertion(cs1) ∧
        (writerChoosesSlot_Assertion(cs2)) ∧
         (write_Assertion(cs2)) ∧
          (writerIndicatesSlot_Assertion(cs2)) ∧ (writerIndicatesPair_Assertion(cs2)) ⇒
     (cs2'nri = rd ∧ cs2'nwi = wr ⇒
       (¬ cs2'reader'readerPair = cs2'writer'writerPair) ∨
        (¬ cs2'reader'readerSlot = cs2'writer'writerSlot))

vc2_read: THEOREM
  ∀ (cs1: Conc_State):
    pre_read(cs1) ∧
     (read_Assertion(cs1)) ∧
      firstWriterChoosesPair_Assertion(cs1) ∧
       (writerChoosesSlot_Assertion(cs1)) ∧
        (write_Assertion(cs1)) ∧
         (writerIndicatesSlot_Assertion(cs1)) ∧ (writerIndicatesPair_Assertion(cs1)) ⇒
     (cs1'nri = rd ∧ cs1'nwi = wr ⇒
       (¬ cs1'reader'readerPair = cs1'writer'writerPair) ∨
        (¬ cs1'reader'readerSlot = cs1'writer'writerSlot))

vc3_read: THEOREM
  ∀ (cs1, cs2: Conc_State):
    pre_read(cs1) ∧
     (read_Assertion(cs1)) ∧
      cs2 = read(cs1)'1 ∧
       firstWriterChoosesPair_Assertion(cs1) ∧
        (writerChoosesSlot_Assertion(cs2)) ∧
         (write_Assertion(cs2)) ∧
          (writerIndicatesSlot_Assertion(cs2)) ∧ (writerIndicatesPair_Assertion(cs2)) ⇒
     (cs2'nri = rd ∧ cs2'nwi = wr ⇒
       (¬ cs2'reader'readerPair = cs2'writer'writerPair) ∨
        (¬ cs2'reader'readerSlot = cs2'writer'writerSlot))

END FOUR_SLOT
```

Appendix H

The Freshness Proof

The model of the 4-slot implementation given in this appendix is the same as the one given in Appendix D, except that there are a number of additional variables which are required for the verify the ACM transmits fresh data between its reader and writer. The model has been used to prove that Simpson's 4-slot ACM transmits globally fresh data between its reader and writer, when the reader and writer actions are atomic, but can interleave in an unrestricted manner. This proof, together with the proof of coherence from Appendix G is sufficient to prove that the ACM is L-atomic.

```
FOUR_SLOT: THEORY
 BEGIN
```

The ACM transmits data items, consisting of a value and an index number, between its reader and writer.

```
  Val: NONEMPTY_TYPE

  Data: TYPE = [# index: nat, val: Val #]
```

Types to represent the names of the pairs and slots in the ACM.

```
  PairIndex: TYPE = {p0, p1}

  SlotIndex: TYPE = {s0, s1}
```

The program counters, which record the next operation (instruction) to be executed by the reader and writer.

```
  NextReadInstruction: TYPE = {firstRcp, rcp, rip, rcs, rd}

  NextWriteInstruction: TYPE = {firstWcp, wcp, wcs, wr, wis, wip}
```

Types to record the current locations of the reader and writer in their respective assertion networks.

```
  ReaderNetworkState: TYPE = {sr, lr1, lr2, lr3, lr4, tr}

  WriterNetworkState: TYPE = {sw, lw1, lw2, lw3, lw4, lw5, tw}
```

The local state of the writer, which has an auxiliary variable, *currentState*, to record its current location in its assertion network.

```
WriterState: TYPE =
[# writerPair: PairIndex,
   writerSlot: SlotIndex,
   currentState: WriterNetworkState #]
```

The local state of the reader, which also has an auxiliary variable to record its location in its assertion network.

```
ReaderState: TYPE =
[# readerPair: PairIndex,
   readerSlot: SlotIndex,
   currentState: ReaderNetworkState #]
```

The state of the ACM, which has auxiliary variables called *wisOccurred* and *rcsSinceWis*, which are used to reason about the ordering of the writer operation *writerIndicatesSlot* and the reader operation *readerChoosesSlot*. This ordering can affect the slot that the reader accesses during a particular read. It also has auxiliary variables which are used to verify the ACM is L-atomic (their use is explained before the relevant operations and proofs) called *minFresh*, *maxFresh*, *newMaxFresh*, *indexRead* and *newIndexRead*.

```
Conc_State: TYPE =
[# pairWritten: PairIndex,
   slotWritten: [PairIndex → SlotIndex],
   lastSlotWritten: [PairIndex → SlotIndex],
   pairReading: PairIndex,
   slots: [PairIndex, SlotIndex → Data],
   nri: NextReadInstruction,
   nwi: NextWriteInstruction,
   writer: WriterState,
   reader: ReaderState,
   maxFresh: nat,
   newMaxFresh: nat,
   minFresh: nat,
   indexRead: nat,
   lastIndexRead: nat,
   wisOccurred: bool,
   rcsSinceWis: bool #]
```

The reader and writer operations follow: in each case the pre-condition is simply that the program counter has the correct value to execute the operation.

The *firstReaderChoosesPair* operation sets the local variable *readerPair* equal to *pairWritten*, since the reader attempts to follow the writer in order to read the latest data written. The auxiliary variable *maxFresh* records the index of the last item written prior to the start of the read, and the reader records this index in *minFresh* (the index of the oldest item that is available to be read).

```
pre_firstReaderChoosesPair(p: Conc_State): bool = p'nri = firstRcp

post_firstReaderChoosesPair(p: (pre_firstReaderChoosesPair))(prot: Conc_State): bool =
    prot = p WITH [nri := rip,
                   reader := p'reader WITH
                         [readerPair := p'pairWritten, currentState := lr1],
                   minFresh := p'maxFresh]

firstReaderChoosesPair:
[p: (pre_firstReaderChoosesPair) → (post_firstReaderChoosesPair(p))]
```

The *readerChoosesPair* operation is similar to the *firstReaderChoosesPair* except that the first item available to the reader may be different depending on the recent history of the ACM. If the reader and writer are accessing the same pair of slots and the writer has already executed *writerIndicatesSlot* the reader cannot access the item written during the last write: in this case *minFresh* is set equal to the index of the item written during the current write, *newMaxFresh*. Otherwise *minFresh* is set equal to *maxFresh*.

```
pre_readerChoosesPair(p: Conc_State): bool = p'nri = rcp

post_readerChoosesPair(p: (pre_readerChoosesPair))(prot: Conc_State): bool =
    (p'reader'readerPair = p'writer'writerPair ⇒
      (p'wisOccurred ⇒
        prot = p WITH [nri := rip,
                       reader := p'reader WITH
                             [readerPair := p'pairWritten, currentState := lr1],
                       minFresh := p'newMaxFresh]) ∧
       (¬ p'wisOccurred ⇒
         prot = p WITH [nri := rip,
                        reader := p'reader WITH
                             [readerPair := p'pairWritten, currentState := lr1],
                        minFresh := p'maxFresh])) ∧
     (¬ p'reader'readerPair = p'writer'writerPair ⇒
       prot = p WITH [nri := rip,
                      reader := p'reader WITH
                             [readerPair := p'pairWritten, currentState := lr1],
                      minFresh := p'maxFresh])

readerChoosesPair:
[p: (pre_readerChoosesPair) → (post_readerChoosesPair(p))]
```

The *readerIndicatesPair* operation sets the control variable *pairReading* equal to the reader local variable *readerPair*.

```
pre_readerIndicatesPair(p: Conc_State): bool = p'nri = rip

post_readerIndicatesPair(p: (pre_readerIndicatesPair))(prot: Conc_State): bool =
    prot = p WITH [nri := rcs,
                   pairReading := p'reader'readerPair,
                   reader := p'reader WITH [currentState := lr2]]

readerIndicatesPair:
[p: (pre_readerIndicatesPair) → (post_readerIndicatesPair(p))]
```

At *readerChoosesSlot* the reader chooses the slot it is going to read from in its current pair, by setting the local variable *readerSlot* equal to the value of the element of the *slotWritten* array for its current pair. The reader has now

chosen the item it is going to read so the auxiliary variable *indexRead* is set equal to the index of the item chosen, and *lastIndexRead* is set equal to the value of *indexRead* before the operation is executed (the index of the item read during the last read). It also sets the auxiliary variable *rcsSinceWis* to true to record that *readerChoosesSlot* has occurred since *writerIndicatesSlot*.

```
pre_readerChoosesSlot(p: Conc_State): bool = p'nri = rcs

post_readerChoosesSlot(p: (pre_readerChoosesSlot))(prot: Conc_State): bool =
    (p'pairWritten = p'pairReading ∧ p'reader'readerPair = p'writer'writerPair ⇒
      prot = p WITH [nri := rd,
                     reader := p'reader WITH [readerSlot := p'slotWritten(p'reader'readerPair),
                                              currentState := lr3],
                     indexRead := p'slots(p'reader'readerPair,
                                          p'slotWritten(p'reader'readerPair))'index,
                     lastIndexRead := p'indexRead,
                     rcsSinceWis := TRUE]) ∧
     (¬ (p'pairWritten = p'pairReading ∧ p'reader'readerPair = p'writer'writerPair) ⇒
      prot = p WITH [nri := rd,
                     reader := p'reader WITH [readerSlot := p'slotWritten(p'reader'readerPair),
                                              currentState := lr3],
                     indexRead := p'slots(p'reader'readerPair,
                                          p'slotWritten(p'reader'readerPair))'index,
                     lastIndexRead := p'indexRead,
                     rcsSinceWis := TRUE])

readerChoosesSlot:
[p: (pre_readerChoosesSlot) → (post_readerChoosesSlot(p))]
```

The *read* operation returns the item read.

```
pre_read(p: Conc_State): bool = p'nri = rd

post_read(p: (pre_read))(prot: Conc_State, v: Val): bool =
    v = p'slots(p'reader'readerPair, p'reader'readerSlot)'val ∧
     prot = p WITH [nri := rcp, reader := p'reader WITH [currentState := lr4]]

read: [p: (pre_read) → (post_read(p))]
```

The *firstWriterChoosesPair* operation chooses the pair that the writer is going to access during the write: it chooses the opposite slot to the one the reader last indicated it was reading. The operation also increments *newMaxFresh* by 1 (the index of the item that is going to be written).

```
pre_firstWriterChoosesPair(p: Conc_State): bool = p'nwi = firstWcp

post_firstWriterChoosesPair(p: (pre_firstWriterChoosesPair))(prot: Conc_State): bool =
    (p'pairReading = p0 ⇒
      prot = p WITH [nwi := wcs,
                     writer := p'writer WITH [writerPair := p1, currentState := lw1],
                     newMaxFresh := p'newMaxFresh + 1]) ∧
     (p'pairReading = p1 ⇒
      prot = p WITH [nwi := wcs,
                     writer := p'writer WITH [writerPair := p0, currentState := lw1],
                     newMaxFresh := p'newMaxFresh + 1])

firstWriterChoosesPair:
[p: (pre_firstWriterChoosesPair) → (post_firstWriterChoosesPair(p))]
```

writerChoosesPair is the same as *firstWriterChoosesPair*, except that it sets the auxiliary variables *wisOccurred* and *rcsSinceWis* to false to record the *writerIndicatesSlot* has not occurred during the current write and it is no longer necessary to record that *readerChoosesSlot* has occurred after *writerIndicatesSlot* (this is only important if the reader and writer are accessing the same pair: if this was the case the writer would change pairs at the start of this write to access the opposite pair to the reader).

```
pre_writerChoosesPair(p: Conc_State): bool = p'nwi = wcp

post_writerChoosesPair(p: (pre_writerChoosesPair))(prot: Conc_State): bool =
    (p'pairReading = p0 ⇒
      prot = p WITH [nwi := wcs,
                     writer := p'writer WITH [writerPair := p1, currentState := lw1],
                     newMaxFresh := p'newMaxFresh + 1,
                     wisOccurred := FALSE,
                     rcsSinceWis := FALSE]) ∧
     (p'pairReading = p1 ⇒
       prot = p WITH [nwi := wcs,
                      writer := p'writer WITH [writerPair := p0, currentState := lw1],
                      newMaxFresh := p'newMaxFresh + 1,
                      wisOccurred := FALSE,
                      rcsSinceWis := FALSE])

writerChoosesPair:
[p: (pre_writerChoosesPair) → (post_writerChoosesPair(p))]
```

The *writerChoosesSlot* operation chooses the slot the writer is going to access: the opposite slot to the one that it used the last item it accessed its current pair of slots.

```
pre_writerChoosesSlot(p: Conc_State): bool = p'nwi = wcs

post_writerChoosesSlot(p: (pre_writerChoosesSlot))(prot: Conc_State): bool =
    (p'slotWritten(p'writer'writerPair) = s0 ⇒
      prot = p WITH [nwi := wr, writer := p'writer WITH [writerSlot := s1, currentState := lw2]]) ∧
     (p'slotWritten(p'writer'writerPair) = s1 ⇒
       prot = p WITH [nwi := wr, writer := p'writer WITH [writerSlot := s0, currentState := lw2]])

writerChoosesSlot:
[p: (pre_writerChoosesSlot) → (post_writerChoosesSlot(p))]
```

The write operation writes the new item to the ACM (with index *newMaxFresh*).

```
pre_write(p: Conc_State): bool = p'nwi = wr

write_parameter: TYPE = [# p1: (pre_write), v: Val #]

post_write(p: write_parameter)(prot: Conc_State): bool =
    prot = p'p1 WITH [nwi := wis,
                      (slots)(p'p1'writer'writerPair, p'p1'writer'writerSlot)
                            := (# index := p'p1'newMaxFresh, val := p'v #),
                      writer := p'p1'writer WITH [currentState := lw3]]

write: [p: write_parameter → (post_write(p))]
```

writerIndicatesSlot indicates the slot the writer has accessed, by setting the appropriate element of the *slotWritten* for the pair the writer is accessing

equal to the writer local variable *writerSlot*. It also sets *wisOccurred* to true to indicate that the operation has been executed and the auxiliary variable *rcsSincWis* to false to indicate that *readerChoosesSlot* has not occurred since *writerIndicatesSlot*.

```
pre_writerIndicatesSlot(p: Conc_State): bool = p'nwi = wis

post_writerIndicatesSlot(p: (pre_writerIndicatesSlot))(prot: Conc_State): bool =
    prot = p WITH [nwi := wip,
                   (slotWritten)(p'writer'writerPair) := (p'writer'writerSlot),
                   writer := p'writer WITH [currentState := lw4],
                   wisOccurred := TRUE,
                   rcsSinceWis := FALSE]

writerIndicatesSlot:
[p: (pre_writerIndicatesSlot) → (post_writerIndicatesSlot(p))]
```

The *writerIndicatesPair* operation indicates the pair the writer has accessed by setting the control variable *pairWritten* equal to the writer local variable *writerPair*. It also sets *maxFresh* equal to *newMaxFresh*.

```
pre_writerIndicatesPair(p: Conc_State): bool = p'nwi = wip

post_writerIndicatesPair(p: (pre_writerIndicatesPair))(prot: Conc_State): bool =
    prot = p WITH [nwi := wcp,
                   pairWritten := p'writer'writerPair,
                   writer := p'writer WITH [currentState := lw5],
                   maxFresh := p'newMaxFresh]

writerIndicatesPair:
[p: (pre_writerIndicatesPair) → (post_writerIndicatesPair(p))]
```

Initialisation functions for the reader and writer. Except for correctly setting the respective locations in the assertion networks to their appropriate values the initial values of the variables are irrelevant, since the reader and writer both choose the slot and pair they are going to access by reference to the control variables in the ACM before accessing their chosen slots.

```
init_writer(w: WriterState): bool =
    w = w WITH [writerPair := p0, writerSlot := s0, currentState := sw]

init_reader(r: ReaderState): bool =
    r = r WITH [readerPair := p1, readerSlot := s1, currentState := sr]
```

The initialisation function for the ACM initialises slot 0 in pair 0 (the other slots are initialised with an invalid value), sets the control variables to point to this slot and sets the auxiliary variables to their initial values.

```
init_data(init_data: Data, init_val: Val): bool =
    init_data = init_data WITH [index := 0, val := init_val]

init_prot(p: Conc_State, init_Val, inv_Val: Val, w: WriterState, r: ReaderState): bool =
        p = p WITH [pairWritten := p0,
                    (slotWritten)(p0) := s0,
                    (slotWritten)(p1) := s0,
                    pairReading := r'readerPair,
                    (slots)(p0, s0) := (# index := 1, val := init_Val #),
```

```
(slots)(p0, s1) := (# index := 0, val := inv_Val #),
(slots)(p1, s0) := (# index := 0, val := inv_Val #),
(slots)(p1, s1) := (# index := 0, val := inv_Val #),
nri := rcp,
nwi := wcp,
writer := w,
reader := r,
maxFresh := 1,
newMaxFresh := 1,
minFresh := 0,
indexRead := 0,
lastIndexRead := 0,
wisOccurred := TRUE]
```

The *firstReaderChoosesPair_Assertion* simply asserts that the auxiliary variables *indexRead* and *lastIndexRead* are both equal to their initial values (0) and *rcsSinceWis* is false.

```
firstReaderChoosesPair_Assertion: [Conc_State → bool] =
    (λ  · (cs: Conc_State):
      cs'nri = firstRcp ⇒
       ¬ cs'rcsSinceWis ∧ cs'indexRead = 0 ∧ cs'lastIndexRead = 0
```

The remaining reader assertions assert the relative values of the auxiliary variables *indexRead*, *lastIndexRead*, *minFresh* and *maxFresh* which are used to ensure that the reader always reads fresh data. In each case the relationship is given for each of the possible cases in the assertion: this is not strictly necessary, but it makes it easier to discharge the proof obligations using PVS.

The *readerChoosesPair_Assertion* states that *readerPair* is equal to *pairReading* since the reader has not chosen the pair it is going to read from. There are then two possible cases for the values of the auxiliary variables depending on the recent history of the mechanism. If the reader accessed the same pair as the writer during the last write, it chose its slot after the write indicated the slot it had accesses (*rcsSinceWis* = true), and the writer has not completed the write by executing *writerIndicatesPair* the reader may have read the latest item that has not been released so *indexRead* $\leq$ *newMaxFresh*, in all other cases *indexRead* $\leq$ *maxFresh*. The reader must read the items in order, therefore *indexRead* $\geq$ *lastIndexRead*. *minFresh* records the index of the first item available to the reader, therefore *indexRead* $\geq$ *minFresh*, and also *minFresh* $\leq$ *maxFresh*, since the reader can only read items that have been written.

```
readerChoosesPair_Assertion: [Conc_State → bool] =
    (λ  · (cs: Conc_State):
      cs'nri = rcp ⇒
       cs'reader'readerPair = cs'pairReading ∧
        (cs'pairReading = cs'pairWritten ∧
          cs'reader'readerPair = cs'writer'writerPair ∧ cs'reader'readerPair = cs'pairReading ⇒
          (¬ cs'wisOccurred ⇒
            cs'minFresh ≤ cs'maxFresh ∧
             cs'indexRead ≤ cs'maxFresh ∧
```

```
      cs'indexRead ≥ cs'minFresh ∧ cs'lastIndexRead ≤ cs'indexRead) ∧
   (cs'wisOccurred ⇒
     (¬ cs'rcsSinceWis ⇒
       cs'minFresh ≤ cs'maxFresh ∧
        cs'indexRead ≤ cs'maxFresh ∧
         cs'indexRead ≥ cs'minFresh ∧ cs'lastIndexRead ≤ cs'indexRead) ∧
      (cs'rcsSinceWis ⇒
        cs'minFresh ≤ cs'newMaxFresh ∧
         cs'indexRead ≤ cs'newMaxFresh ∧
          cs'indexRead ≥ cs'minFresh ∧ cs'lastIndexRead ≤ cs'indexRead))) ∧
(cs'pairReading = cs'pairWritten ∧
  ¬ cs'reader'readerPair = cs'writer'writerPair ∧
   cs'reader'readerPair = cs'pairReading ⇒
  cs'minFresh ≤ cs'maxFresh ∧
   cs'indexRead ≤ cs'maxFresh ∧
    cs'indexRead ≥ cs'minFresh ∧ cs'lastIndexRead ≤ cs'indexRead) ∧
 (¬ cs'pairReading = cs'pairWritten ∧
   ¬ cs'reader'readerPair = cs'writer'writerPair ∧
    cs'reader'readerPair = cs'pairReading ⇒
   cs'minFresh ≤ cs'maxFresh ∧
    cs'indexRead ≤ cs'maxFresh ∧
     cs'indexRead ≥ cs'minFresh ∧
      cs'lastIndexRead ≤ cs'indexRead)
```

When the reader is about to execute *readerIndicatesPair* it may have changed pairs, it is therefore not possible to assert anything about the values of the control variables since it has not yet indicated that it has changed. The relationship between the auxiliary variables is almost identical to that for *readerChoosesPair* assertion except: the reader incremented *minFresh* to be equal to *maxFresh* during the last operation so now $indexRead \leq minFresh$; and also if the writer has executed *writerIndicatesPair* ($wisOccurred = \mathsf{true}$) and the reader and writer are accessing the same pair the reader may be able to read the item written during the current write and *minFresh* and *indexRead* are related to the value of *newMaxFresh* ($minFresh \leq newMaxFresh \wedge indexRead \leq newMaxFresh$). There is also an extra possible case to consider, where the reader has changed pairs to follow the writer and has not yet indicated it has changed ($\neg\, readerPair = pairReading \wedge \neg\, pairWritten = pairReading$) - this is the only time this relationship can possibly hold. In addition it is necessary to record the relationship between the value of *minFresh* and the index of an item in one of the slots. This relationship depends on which slot contained the first item available to the reader when *readerChoosesSlot* was executed. At *readerChoosesPair* *minFresh* is set equal to the value of the index of the first item available to the reader ($slots(pairWritten, slotWritten(pairWritten).index)$ so *minFresh* is normally less than or equal to this value. The only exception is if the reader writer has changed pairs since the reader chose the slot to access, when *minFresh* is related to the index of the item in the last slot written in the opposite pair to the writer, and *indexRead* must be less than or equal to this value (the writer may have written subsequent items to the ACM) e.g. $pairWritten = p_1 \Rightarrow minFresh \leq slots(p_0, slotWritten(p_0)).index$.

```
readerIndicatesPair_Assertion: [Conc_State → bool] =
    (λ ·(cs: Conc_State):
      cs'nri = rip ⇒
       (cs'pairReading = cs'pairWritten ⇒ cs'pairReading = cs'reader'readerPair) ∧
        (cs'pairReading = cs'pairWritten ∧
          cs'reader'readerPair = cs'writer'writerPair ∧ cs'reader'readerPair = cs'pairReading ⇒
          (¬ cs'wisOccurred ⇒
            cs'minFresh ≤ cs'maxFresh ∧
             cs'indexRead ≤ cs'maxFresh ∧
              cs'indexRead ≤ cs'minFresh ∧ cs'lastIndexRead ≤ cs'indexRead) ∧
           (cs'wisOccurred ⇒
             cs'minFresh ≤ cs'newMaxFresh ∧
              cs'indexRead ≤ cs'newMaxFresh ∧
               cs'indexRead ≤ cs'minFresh ∧ cs'lastIndexRead ≤ cs'indexRead)
            ∧ cs'minFresh ≤ cs'slots(cs'pairWritten, cs'slotWritten(cs'pairWritten))'index) ∧
         (cs'pairReading = cs'pairWritten ∧
           ¬ cs'reader'readerPair = cs'writer'writerPair ∧
            cs'reader'readerPair = cs'pairReading ⇒
           cs'minFresh ≤ cs'maxFresh ∧
            cs'indexRead ≤ cs'maxFresh ∧
             cs'indexRead ≤ cs'minFresh ∧
              cs'lastIndexRead ≤ cs'indexRead ∧
               cs'minFresh ≤ cs'slots(cs'pairWritten, cs'slotWritten(cs'pairWritten))'index) ∧
          (¬ cs'pairReading = cs'pairWritten ∧
            cs'reader'readerPair = cs'writer'writerPair ∧
             ¬ cs'reader'readerPair = cs'pairReading ⇒
            cs'minFresh ≤ cs'maxFresh ∧
             cs'indexRead ≤ cs'maxFresh ∧
              cs'indexRead ≤ cs'minFresh ∧
               cs'lastIndexRead ≤ cs'indexRead ∧
                cs'minFresh ≤ cs'slots(cs'pairWritten, cs'slotWritten(cs'pairWritten))'index) ∧
           (¬ cs'pairReading = cs'pairWritten ∧
             ¬ cs'reader'readerPair = cs'writer'writerPair ∧
              cs'reader'readerPair = cs'pairReading ⇒
             cs'minFresh ≤ cs'maxFresh ∧
              cs'indexRead ≤ cs'maxFresh ∧
               cs'indexRead ≤ cs'minFresh ∧
                cs'lastIndexRead ≤ cs'indexRead ∧
                 (cs'pairWritten = p0 ⇒ cs'minFresh ≤ cs'slots(p1, cs'slotWritten(p1))'index) ∧
                  (cs'pairWritten = p1 ⇒
                    cs'minFresh ≤
                     cs'slots(p0, cs'slotWritten(p0))'index))
```

When the reader is about to execute *readerChoosesSlot* it has indicated the pair it is accessing, therefore the control variable *pairReading* is equal to the reader local variable *readerPair*. The reader can no longer be accessing the same pair as the writer unless *pairReading* is equal to *pairWritten* so the extra relationship between the control variables that was necessary in the *readerIndicatesPair_Assertion* is no longer required, otherwise the assertion is identical to the previous one.

```
readerChoosesSlot_Assertion: [Conc_State → bool] =
    (λ ·(cs: Conc_State):
      cs'nri = rcs ⇒
       cs'reader'readerPair = cs'pairReading ∧
        (cs'pairReading = cs'pairWritten ∧
          cs'reader'readerPair = cs'writer'writerPair ∧ cs'reader'readerPair = cs'pairReading ⇒
          (¬ cs'wisOccurred ⇒
            cs'minFresh ≤ cs'maxFresh ∧
             cs'indexRead ≤ cs'maxFresh ∧
```

```
        cs'indexRead ≤ cs'minFresh ∧ cs'lastIndexRead ≤ cs'indexRead) ∧
      (cs'wisOccurred ⇒
        cs'minFresh ≤ cs'newMaxFresh ∧
         cs'indexRead ≤ cs'newMaxFresh ∧
          cs'indexRead ≤ cs'minFresh ∧ cs'lastIndexRead ≤ cs'indexRead)
       ∧ cs'minFresh ≤ cs'slots(cs'pairWritten, cs'slotWritten(cs'pairWritten))'index) ∧
     (cs'pairReading = cs'pairWritten ∧
      ¬ cs'reader'readerPair = cs'writer'writerPair ∧
       cs'reader'readerPair = cs'pairReading ⇒
       cs'minFresh ≤ cs'maxFresh ∧
        cs'indexRead ≤ cs'maxFresh ∧
         cs'indexRead ≤ cs'minFresh ∧
          cs'lastIndexRead ≤ cs'indexRead ∧
           cs'minFresh ≤ cs'slots(cs'pairWritten, cs'slotWritten(cs'pairWritten))'index) ∧
      (¬ cs'pairReading = cs'pairWritten ∧
       ¬ cs'reader'readerPair = cs'writer'writerPair ∧
        cs'reader'readerPair = cs'pairReading ⇒
        cs'minFresh ≤ cs'maxFresh ∧
         cs'indexRead ≤ cs'maxFresh ∧
          cs'indexRead ≤ cs'minFresh ∧
           cs'lastIndexRead ≤ cs'indexRead ∧
            (cs'pairWritten = p0 ⇒ cs'minFresh ≤ cs'slots(p1, cs'slotWritten(p1))'index) ∧
             (cs'pairWritten = p1 ⇒
               cs'minFresh ≤
                cs'slots(p0, cs'slotWritten(p0))'index))
```

When the reader has chosen the slot it is going to access it can start to read the item at any time. *readerChoosesSlot* is therefore taken to mark the start of the read access, and sets *indexRead* equal to the index of the item in the slot the reader has chosen. This assertion is identical to the previous one except that it therefore asserts $indexRead \geq minFresh$.

```
read_Assertion: [Conc_State → bool] =
    (λ ·(cs: Conc_State):
      cs'nri = rd ⇒
       cs'reader'readerPair = cs'pairReading ∧
        (cs'pairReading = cs'pairWritten ∧
          cs'reader'readerPair = cs'writer'writerPair ∧ cs'reader'readerPair = cs'pairReading ⇒
          (¬ cs'wisOccurred ⇒
            cs'minFresh ≤ cs'maxFresh ∧
             cs'indexRead ≤ cs'maxFresh ∧
              cs'indexRead ≥ cs'minFresh ∧ cs'lastIndexRead ≤ cs'indexRead) ∧
           (cs'wisOccurred ⇒
             (¬ cs'rcsSinceWis ⇒
               cs'minFresh ≤ cs'maxFresh ∧
                cs'indexRead ≤ cs'maxFresh ∧
                 cs'indexRead ≥ cs'minFresh ∧ cs'lastIndexRead ≤ cs'indexRead) ∧
              (cs'rcsSinceWis ⇒
                cs'minFresh ≤ cs'newMaxFresh ∧
                 cs'indexRead ≤ cs'newMaxFresh ∧
                  cs'indexRead ≥ cs'minFresh ∧ cs'lastIndexRead ≤ cs'indexRead))
            ∧ cs'minFresh ≤ cs'slots(cs'pairWritten, cs'slotWritten(cs'pairWritten))'index) ∧
          (cs'pairReading = cs'pairWritten ∧
           ¬ cs'reader'readerPair = cs'writer'writerPair ∧
            cs'reader'readerPair = cs'pairReading ⇒
            cs'minFresh ≤ cs'maxFresh ∧
             cs'indexRead ≤ cs'maxFresh ∧
              cs'indexRead ≥ cs'minFresh ∧
               cs'lastIndexRead ≤ cs'indexRead ∧
                cs'minFresh ≤ cs'slots(cs'pairWritten, cs'slotWritten(cs'pairWritten))'index) ∧
           (¬ cs'pairReading = cs'pairWritten ∧
```

```
¬ cs'reader'readerPair = cs'writer'writerPair ∧
 cs'reader'readerPair = cs'pairReading ⇒
cs'minFresh ≤ cs'maxFresh ∧
 cs'indexRead ≤ cs'maxFresh ∧
  cs'indexRead ≥ cs'minFresh ∧
   cs'lastIndexRead ≤ cs'indexRead ∧
    (cs'pairWritten = p0 ⇒ cs'minFresh ≤ cs'slots(p1, cs'slotWritten(p1))'index) ∧
     (cs'pairWritten = p1 ⇒
       cs'minFresh ≤
        cs'slots(p0, cs'slotWritten(p0))'index))
```

The *firstWriterChoosesPair* and *writerChoosesPair* assertions are identical, except for the value of the program counter, and *wisOccurred* is false when *firstWriterChoosesPair* is about to be executed (the variable is set to false by *writerChoosesPair* during future writes). The writer local variables are equal to the relevant control variables, *maxFresh* is equal to *newMaxFresh*, the index of the latest item (in the slot pointed to by the control variables) is equal to *maxFresh* and the indices of the items in the other slots must be at least one less that *maxFresh*.

```
firstWriterChoosesPair_Assertion: [Conc_State → bool] =
    (λ ·(cs: Conc_State):
      cs'nwi = firstWcp ⇒
       ¬ cs'wisOccurred ∧
        cs'writer'writerPair = cs'pairWritten ∧
         cs'writer'writerSlot = cs'slotWritten(cs'pairWritten) ∧
          cs'maxFresh = cs'newMaxFresh ∧
           cs'maxFresh = cs'slots(cs'pairWritten, cs'slotWritten(cs'pairWritten))'index ∧
            (cs'slotWritten(cs'pairWritten) = s0 ⇒
              cs'slots(cs'pairWritten, s1)'index ≤ cs'maxFresh-1) ∧
             (cs'slotWritten(cs'pairWritten) = s1 ⇒
               cs'slots(cs'pairWritten, s0)'index ≤ cs'maxFresh-1) ∧
              (cs'pairWritten = p0 ⇒
                cs'slots(p1, s0)'index ≤ cs'maxFresh-1 ∧
                 cs'slots(p1, s1)'index ≤ cs'maxFresh-1) ∧
               (cs'pairWritten = p1 ⇒
                 cs'slots(p0, s0)'index ≤ cs'maxFresh-1 ∧
                  cs'slots(p0, s1)'index ≤ cs'maxFresh-1)

writerChoosesPair_Assertion: [Conc_State → bool] =
    (λ ·(cs: Conc_State):
      cs'nwi = wcp ⇒
       cs'wisOccurred ∧
        cs'writer'writerPair = cs'pairWritten ∧
         cs'writer'writerSlot = cs'slotWritten(cs'pairWritten) ∧
          cs'maxFresh = cs'newMaxFresh ∧
           cs'maxFresh = cs'slots(cs'pairWritten, cs'slotWritten(cs'pairWritten))'index ∧
            (cs'slotWritten(cs'pairWritten) = s0 ⇒
              cs'slots(cs'pairWritten, s1)'index ≤ cs'maxFresh-1) ∧
             (cs'slotWritten(cs'pairWritten) = s1 ⇒
               cs'slots(cs'pairWritten, s0)'index ≤ cs'maxFresh-1) ∧
              (cs'pairWritten = p0 ⇒
                cs'slots(p1, s0)'index ≤ cs'maxFresh-1 ∧
                 cs'slots(p1, s1)'index ≤ cs'maxFresh-1) ∧
               (cs'pairWritten = p1 ⇒
                 cs'slots(p0, s0)'index ≤ cs'maxFresh-1 ∧
                  cs'slots(p0, s1)'index ≤ cs'maxFresh-1)
```

When the writer is about to execute *writerChoosesSlot* it may have changed pairs during the previous operation, so it only possible to state that the

writer local variable *writerSlot* will still be equal to the element of the *slotWritten* array for the pair the writer accessed during the last write; and if the writer has changed pairs and indicated and the reader has not indicated that it has subsequently followed the writer to the new pair ($\neg$ *pairWritten* = *pairReading*) then the writer local variable *writerPair* will be equal to the control variable *pairWritten* (the writer will not have changed pairs at *writerChoosesPair*). The last operation set *wisOccurred* to false, and incremented *newMaxFresh* so it is now 1 greater than *maxFresh*. Otherwise the assertion is identical to the previous one.

```
writerChoosesSlot_Assertion: [Conc_State → bool] =
    (λ ·(cs: Conc_State):
      cs'nwi = wcs ⇒
       ¬ cs'wisOccurred ∧
        cs'maxFresh = cs'newMaxFresh-1 ∧
         (¬ cs'pairWritten = cs'pairReading ⇒ cs'pairWritten = cs'writer'writerPair) ∧
          cs'writer'writerSlot = cs'slotWritten(cs'pairWritten) ∧
           cs'maxFresh = cs'slots(cs'pairWritten, cs'slotWritten(cs'pairWritten))'index ∧
            (cs'slotWritten(cs'pairWritten) = s0 ⇒
              cs'slots(cs'pairWritten, s1)'index ≤ cs'maxFresh-1) ∧
             (cs'slotWritten(cs'pairWritten) = s1 ⇒
               cs'slots(cs'pairWritten, s0)'index ≤ cs'maxFresh-1) ∧
              (cs'pairWritten = p0 ⇒
                cs'slots(p1, s0)'index ≤ cs'maxFresh-1 ∧
                 cs'slots(p1, s1)'index ≤ cs'maxFresh-1) ∧
               (cs'pairWritten = p1 ⇒
                 cs'slots(p0, s0)'index ≤ cs'maxFresh-1 ∧
                  cs'slots(p0, s1)'index ≤ cs'maxFresh-1)
```

The assertion when the writer is about to execute the *write* operation is identical to the previous one, except that the writer has now chosen the slot it is going to access, so the local variable *writerSlot* is equal to the opposite value to the one recorded in the element of the *slotWritten* array for the pair the writer is accessing.

```
write_Assertion: [Conc_State → bool] =
    (λ ·(cs: Conc_State):
      cs'nwi = wr ⇒
       ¬ cs'wisOccurred ∧
        cs'maxFresh = cs'newMaxFresh-1 ∧
         (¬ cs'pairWritten = cs'pairReading ⇒ cs'pairWritten = cs'writer'writerPair) ∧
          ¬ cs'writer'writerSlot = cs'slotWritten(cs'writer'writerPair) ∧
           cs'maxFresh = cs'slots(cs'pairWritten, cs'slotWritten(cs'pairWritten))'index ∧
            (cs'slotWritten(cs'pairWritten) = s0 ⇒
              cs'slots(cs'pairWritten, s1)'index ≤ cs'maxFresh-1) ∧
             (cs'slotWritten(cs'pairWritten) = s1 ⇒
               cs'slots(cs'pairWritten, s0)'index ≤ cs'maxFresh-1) ∧
              (cs'pairWritten = p0 ⇒
                cs'slots(p1, s0)'index ≤ cs'maxFresh-1 ∧
                 cs'slots(p1, s1)'index ≤ cs'maxFresh-1) ∧
               (cs'pairWritten = p1 ⇒
                 cs'slots(p0, s0)'index ≤ cs'maxFresh-1 ∧
                  cs'slots(p0, s1)'index ≤ cs'maxFresh-1)
```

The assertion when the writer is about to execute the *writerIndicatesSlot* operation is again identical to the previous one, except that the slot pointed

to by the writer control variables has had the new item written to it. so the item it contains has an index equal to *newMaxFresh*.

```
writerIndicatesSlot_Assertion: [Conc_State → bool] =
    (λ ·(cs: Conc_State):
      cs'nwi = wis ⇒
       ¬ cs'wisOccurred ∧
        cs'maxFresh = cs'newMaxFresh-1 ∧
         (¬ cs'pairWritten = cs'pairReading ⇒ cs'pairWritten = cs'writer'writerPair) ∧
          ¬ cs'writer'writerSlot = cs'slotWritten(cs'writer'writerPair) ∧
           cs'maxFresh = cs'slots(cs'pairWritten, cs'slotWritten(cs'pairWritten))'index ∧
            cs'newMaxFresh = cs'slots(cs'writer'writerPair, cs'writer'writerSlot)'index ∧
             (cs'writer'writerPair = cs'pairWritten ⇒
               (cs'pairWritten = p0 ⇒
                 cs'slots(p1, s0)'index ≤ cs'maxFresh-1 ∧
                  cs'slots(p1, s1)'index ≤ cs'maxFresh-1) ∧
                (cs'pairWritten = p1 ⇒
                  cs'slots(p0, s0)'index ≤ cs'maxFresh-1 ∧
                   cs'slots(p0, s1)'index ≤ cs'maxFresh-1)) ∧
              (¬ cs'writer'writerPair = cs'pairWritten ⇒
                (cs'slotWritten(cs'pairWritten) = s0 ⇒
                  cs'slots(cs'pairWritten, s1)'index ≤ cs'maxFresh-1) ∧
                 (cs'slotWritten(cs'pairWritten) = s1 ⇒
                   cs'slots(cs'pairWritten, s0)'index ≤ cs'maxFresh-1) ∧
                  (cs'writer'writerSlot = s0 ⇒
                    cs'slots(cs'writer'writerPair, s1)'index ≤ cs'maxFresh-1) ∧
                   (cs'writer'writerSlot = s1 ⇒
                     cs'slots(cs'writer'writerPair, s0)'index ≤
                      cs'maxFresh-1))
```

The assertion when the writer is about to execute *writerIndicatesPair* is once again identical to the previous one except that since it has executed *writerIndicatesSlot*, *wisOccurred* is now true and the local variable *writerSlot* is now equal to the element of the *slotWritten* array for the pair the writer is accessing.

```
writerIndicatesPair_Assertion: [Conc_State → bool] =
    (λ ·(cs: Conc_State):
      cs'nwi = wip ⇒
       cs'wisOccurred ∧
        (¬ cs'pairWritten = cs'pairReading ⇒ cs'pairWritten = cs'writer'writerPair) ∧
         cs'writer'writerSlot = cs'slotWritten(cs'writer'writerPair) ∧
          cs'maxFresh = cs'newMaxFresh-1 ∧
           cs'newMaxFresh = cs'slots(cs'writer'writerPair, cs'writer'writerSlot)'index ∧
            (cs'writer'writerPair = cs'pairWritten ⇒
              (cs'slotWritten(cs'pairWritten) = s0 ⇒
                cs'slots(cs'pairWritten, s1)'index ≤ cs'maxFresh) ∧
               (cs'slotWritten(cs'pairWritten) = s1 ⇒
                 cs'slots(cs'pairWritten, s0)'index ≤ cs'maxFresh) ∧
                (cs'pairWritten = p0 ⇒
                  cs'slots(p1, s0)'index ≤ cs'maxFresh-1 ∧
                   cs'slots(p1, s1)'index ≤ cs'maxFresh-1) ∧
                 (cs'pairWritten = p1 ⇒
                   cs'slots(p0, s0)'index ≤ cs'maxFresh-1 ∧
                    cs'slots(p0, s1)'index ≤ cs'maxFresh-1)) ∧
             (¬ cs'writer'writerPair = cs'pairWritten ⇒
               cs'maxFresh = cs'slots(cs'pairWritten, cs'slotWritten(cs'pairWritten))'index ∧
                (cs'slotWritten(cs'pairWritten) = s0 ⇒
                  cs'slots(cs'pairWritten, s1)'index ≤ cs'maxFresh-1) ∧
                 (cs'slotWritten(cs'pairWritten) = s1 ⇒
                   cs'slots(cs'pairWritten, s0)'index ≤ cs'maxFresh-1) ∧
```

```
(cs'writer'writerSlot = s0 ⇒
  cs'slots(cs'writer'writerPair, s1)'index ≤ cs'maxFresh) ∧
 (cs'writer'writerSlot = s1 ⇒
   cs'slots(cs'writer'writerPair, s0)'index ≤
    cs'maxFresh))
```

The proof obligations for the initialisation functions for the reader and writer (which prove the relevant assertions are established) are as follows:

```
vc_initReader: THEOREM
  ∀ (cs: Conc_State, init: Val, inv: Val, w: WriterState, r: ReaderState):
    init_prot(cs, init, inv, w, r) ⇒ firstReaderChoosesPair_Assertion(cs)

vc_initWriter: THEOREM
  ∀ (cs: Conc_State, init: Val, inv: Val, w: WriterState, r: ReaderState):
    init_prot(cs, init, inv, w, r) ⇒ firstWriterChoosesPair_Assertion(cs)
```

The first proof obligation for each of the locations in the reader and writer networks (*vc1_op_name*) is to establish for each transition in the respective networks that:

1. If the assertion in the start location of the transition associated with each operation holds, and the transition is enabled, that the assertion in the target location of the transition will hold after executing the operation that is associated with the transition. In the case of the four slot the guards for each of the transitions is effectively true i.e. the transition is enabled whenever the component is in the start location of the transition (since the pre-condition for the operation is simply that the program counter for the component is such that the operation is to be executed next).

2. That each of the components does not interfere with the assertions in the network of the other component e.g. if the assertions in the locations of the network of the other component hold before the operation is executed, they will still hold after the operation is executed.

This requires the following proof obligation to be completed for every location in the network of the writer: *vc1_op_name*

$$
\begin{aligned}
&\forall cs1, cs2 : Conc\_State \cdot \\
&\quad pre\_start\_writer\_op\_name(cs1) \wedge \\
&\quad\quad start\_writer\_op\_name\_Assertion(cs1) \wedge \\
&\quad\quad\quad firstReaderChoosesPair\_Assertion(cs1) \wedge \\
&\quad\quad\quad\quad readerChoosesPair\_Assertion(cs1) \wedge \\
&\quad\quad\quad\quad\quad readerIndicatesPair\_Assertion(cs1) \wedge \\
&\quad\quad\quad\quad\quad\quad readerChoosesSlot\_Assertion(cs1) \wedge \\
&\quad\quad\quad\quad\quad\quad\quad read\_Assertion(cs1) \wedge post\_writer\_op\_name(cs1, cs2) \Rightarrow \\
&\quad\quad\quad\quad\quad\quad cs2.nwi = targetLocationInstruction \wedge \\
&\quad\quad\quad\quad\quad\quad\quad target\_writer\_op\_name\_Assertion(cs2) \wedge
\end{aligned}
$$

$firstReaderChoosesPair\_Assertion(cs2) \wedge$
$readerChoosesPair\_Assertion(cs2) \wedge$
$readerIndicatesPair\_Assertion(cs2) \wedge$
$readerChoosesSlot\_Assertion(cs2) \wedge$
$read\_Assertion(cs2)$

Similarly the following proof obligation must be completed for every location in the network of the reader:

$$
\begin{aligned}
&vc1\_op\_name\\
&\quad \forall cs1, cs2 : Conc\_State \cdot\\
&\quad\quad pre\_start\_reader\_op\_name(cs1) \wedge\\
&\quad\quad start\_reader\_op\_name\_Assertion(cs1) \wedge\\
&\quad\quad firstWriterChoosesPair\_Assertion(cs1) \wedge\\
&\quad\quad writerChoosesPair\_Assertion(cs1) \wedge\\
&\quad\quad writerChoosesSlot\_Assertion(cs1) \wedge\\
&\quad\quad write\_Assertion(cs1) \wedge\\
&\quad\quad writerIndicatesSlot\_Assertion(cs1) \wedge\\
&\quad\quad writerIndicatesPair\_Assertion(cs1) \wedge\\
&\quad\quad post\_reader\_op\_name(cs1, cs2) \Rightarrow\\
&\quad\quad cs2.nri = targetLocationInstruction \wedge\\
&\quad\quad target\_reader\_op\_name\_Assertion(cs2) \wedge\\
&\quad\quad firstWriterChoosesPair\_Assertion(cs2) \wedge\\
&\quad\quad writerChoosesPair\_Assertion(cs2) \wedge\\
&\quad\quad writerChoosesSlot\_Assertion(cs2) \wedge\\
&\quad\quad write\_Assertion(cs2) \wedge\\
&\quad\quad writerIndicatesSlot\_Assertion(cs2) \wedge\\
&\quad\quad writerIndicatesPair\_Assertion(cs2)
\end{aligned}
$$

```
vc1_firstReaderChoosesPair: THEOREM
  ∀ (cs1, cs2: Conc_State):
    pre_firstReaderChoosesPair(cs1) ∧
     firstReaderChoosesPair_Assertion(cs1) ∧
      firstWriterChoosesPair_Assertion(cs1) ∧
       writerChoosesPair_Assertion(cs1) ∧
        writerChoosesSlot_Assertion(cs1) ∧
         write_Assertion(cs1) ∧
          writerIndicatesSlot_Assertion(cs1) ∧
           writerIndicatesPair_Assertion(cs1) ∧
            cs2 = firstReaderChoosesPair(cs1) ⇒
     readerIndicatesPair_Assertion(cs2) ∧
      firstWriterChoosesPair_Assertion(cs2) ∧
       writerChoosesPair_Assertion(cs2) ∧
        writerChoosesSlot_Assertion(cs2) ∧
         write_Assertion(cs2) ∧
          writerIndicatesSlot_Assertion(cs2) ∧
           writerIndicatesPair_Assertion(cs2)

vc1_readerChoosesPair: THEOREM
  ∀ (cs1, cs2: Conc_State):
    pre_readerChoosesPair(cs1) ∧
```

```
      readerChoosesPair_Assertion(cs1) ∧
       firstWriterChoosesPair_Assertion(cs1) ∧
        writerChoosesPair_Assertion(cs1) ∧
         writerChoosesSlot_Assertion(cs1) ∧
          write_Assertion(cs1) ∧
           writerIndicatesSlot_Assertion(cs1) ∧
            writerIndicatesPair_Assertion(cs1) ∧
             cs2 = readerChoosesPair(cs1) ⇒
      readerIndicatesPair_Assertion(cs2) ∧
       firstWriterChoosesPair_Assertion(cs2) ∧
        writerChoosesPair_Assertion(cs2) ∧
         writerChoosesSlot_Assertion(cs2) ∧
          write_Assertion(cs2) ∧
           writerIndicatesSlot_Assertion(cs2) ∧
            writerIndicatesPair_Assertion(cs2)

vc1_readerIndicatesPair: THEOREM
  ∀ (cs1, cs2: Conc_State):
    pre_readerIndicatesPair(cs1) ∧
     readerIndicatesPair_Assertion(cs1) ∧
      firstWriterChoosesPair_Assertion(cs1) ∧
       writerChoosesPair_Assertion(cs1) ∧
        writerChoosesSlot_Assertion(cs1) ∧
         write_Assertion(cs1) ∧
          writerIndicatesSlot_Assertion(cs1) ∧
           writerIndicatesPair_Assertion(cs1) ∧
            cs2 = readerIndicatesPair(cs1) ⇒
      readerChoosesSlot_Assertion(cs2) ∧
       firstWriterChoosesPair_Assertion(cs2) ∧
        writerChoosesPair_Assertion(cs2) ∧
         writerChoosesSlot_Assertion(cs2) ∧
          write_Assertion(cs2) ∧
           writerIndicatesSlot_Assertion(cs2) ∧
            writerIndicatesPair_Assertion(cs2)

vc1_readerChoosesSlot: THEOREM
  ∀ (cs1, cs2: Conc_State, v: Val):
    pre_readerChoosesSlot(cs1) ∧
     readerChoosesSlot_Assertion(cs1) ∧
      firstWriterChoosesPair_Assertion(cs1) ∧
       writerChoosesPair_Assertion(cs1) ∧
        writerChoosesSlot_Assertion(cs1) ∧
         write_Assertion(cs1) ∧
          writerIndicatesSlot_Assertion(cs1) ∧
           writerIndicatesPair_Assertion(cs1) ∧
            cs2 = readerChoosesSlot(cs1) ⇒
      read_Assertion(cs2) ∧
       writerChoosesPair_Assertion(cs2) ∧
        writerChoosesPair_Assertion(cs2) ∧
         writerChoosesSlot_Assertion(cs2) ∧
          write_Assertion(cs2) ∧
           writerIndicatesSlot_Assertion(cs2) ∧
            writerIndicatesPair_Assertion(cs2)

vc1_read: THEOREM
  ∀ (cs1, cs2: Conc_State, v: Val):
    pre_read(cs1) ∧
     read_Assertion(cs1) ∧
      firstWriterChoosesPair_Assertion(cs1) ∧
       writerChoosesPair_Assertion(cs1) ∧
        writerChoosesSlot_Assertion(cs1) ∧
         write_Assertion(cs1) ∧
          writerIndicatesSlot_Assertion(cs1) ∧
           writerIndicatesPair_Assertion(cs1) ∧
```

```
            cs2 = read(cs1)'1 ⇒
      readerChoosesPair_Assertion(cs2) ∧
       firstWriterChoosesPair_Assertion(cs2) ∧
        writerChoosesPair_Assertion(cs2) ∧
         writerChoosesSlot_Assertion(cs2) ∧
          write_Assertion(cs2) ∧
           writerIndicatesSlot_Assertion(cs2) ∧
            writerIndicatesPair_Assertion(cs2)

vc1_firstWriterChoosesPair: THEOREM
  ∀ (cs1, cs2: Conc_State, v: Val):
    pre_firstWriterChoosesPair(cs1) ∧
     firstWriterChoosesPair_Assertion(cs1) ∧
      firstReaderChoosesPair_Assertion(cs1) ∧
       readerChoosesPair_Assertion(cs1) ∧
        readerIndicatesPair_Assertion(cs1) ∧
         readerChoosesSlot_Assertion(cs1) ∧
          read_Assertion(cs1) ∧ cs2 =
           firstWriterChoosesPair(cs1) ⇒
     writerChoosesSlot_Assertion(cs2) ∧
      firstReaderChoosesPair_Assertion(cs2) ∧
       readerChoosesPair_Assertion(cs2) ∧
        readerIndicatesPair_Assertion(cs2) ∧
         readerChoosesSlot_Assertion(cs2) ∧
          read_Assertion(cs2)

vc1_writerChoosesPair: THEOREM
  ∀ (cs1, cs2: Conc_State, v: Val):
    pre_writerChoosesPair(cs1) ∧
     writerChoosesPair_Assertion(cs1) ∧
      firstReaderChoosesPair_Assertion(cs1) ∧
       readerChoosesPair_Assertion(cs1) ∧
        readerIndicatesPair_Assertion(cs1) ∧
         readerChoosesSlot_Assertion(cs1) ∧
          read_Assertion(cs1) ∧
           cs2 = writerChoosesPair(cs1) ⇒
     writerChoosesSlot_Assertion(cs2) ∧
      firstReaderChoosesPair_Assertion(cs2) ∧
       readerChoosesPair_Assertion(cs2) ∧
        readerIndicatesPair_Assertion(cs2) ∧
         readerChoosesSlot_Assertion(cs2) ∧
          read_Assertion(cs2)

vc1_writerChoosesSlot: THEOREM
  ∀ (cs1, cs2: Conc_State, v: Val):
    pre_writerChoosesSlot(cs1) ∧
     writerChoosesSlot_Assertion(cs1) ∧
      firstReaderChoosesPair_Assertion(cs1) ∧
       readerChoosesPair_Assertion(cs1) ∧
        readerIndicatesPair_Assertion(cs1) ∧
         readerChoosesSlot_Assertion(cs1) ∧
          read_Assertion(cs1) ∧
           cs2 = writerChoosesSlot(cs1) ⇒
     write_Assertion(cs2) ∧
      firstReaderChoosesPair_Assertion(cs2) ∧
       readerChoosesPair_Assertion(cs2) ∧
        readerIndicatesPair_Assertion(cs2) ∧
         readerChoosesSlot_Assertion(cs2) ∧
          read_Assertion(cs2)

vc1_write: THEOREM
  ∀ (w: write_parameter, cs2: Conc_State, v: Val):
    pre_write(w'p1) ∧
     write_Assertion(w'p1) ∧
```

```
        firstReaderChoosesPair_Assertion(w'p1) ∧
         readerChoosesPair_Assertion(w'p1) ∧
          readerIndicatesPair_Assertion(w'p1) ∧
           readerChoosesSlot_Assertion(w'p1) ∧
            read_Assertion(w'p1) ∧
             cs2 = write(w) ⇒
      writerIndicatesSlot_Assertion(cs2) ∧
       firstReaderChoosesPair_Assertion(cs2) ∧
        readerChoosesPair_Assertion(cs2) ∧
         readerIndicatesPair_Assertion(cs2) ∧
          readerChoosesSlot_Assertion(cs2) ∧
           read_Assertion(cs2)

vc1_writerIndicatesSlot: THEOREM
  ∀ (cs1, cs2: Conc_State, v: Val):
    pre_writerIndicatesSlot(cs1) ∧
     writerIndicatesSlot_Assertion(cs1) ∧
      firstReaderChoosesPair_Assertion(cs1) ∧
       readerChoosesPair_Assertion(cs1) ∧
        readerIndicatesPair_Assertion(cs1) ∧
         readerChoosesSlot_Assertion(cs1) ∧
          read_Assertion(cs1) ∧
           cs2 = writerIndicatesSlot(cs1) ⇒
     writerIndicatesPair_Assertion(cs2) ∧
      firstReaderChoosesPair_Assertion(cs2) ∧
       readerChoosesPair_Assertion(cs2) ∧
        readerIndicatesPair_Assertion(cs2) ∧
         readerChoosesSlot_Assertion(cs2) ∧
          read_Assertion(cs2)

vc1_writerIndicatesPair: THEOREM
  ∀ (cs1, cs2: Conc_State, v: Val):
    pre_writerIndicatesPair(cs1) ∧
     writerIndicatesPair_Assertion(cs1) ∧
      firstReaderChoosesPair_Assertion(cs1) ∧
       readerChoosesPair_Assertion(cs1) ∧
        readerIndicatesPair_Assertion(cs1) ∧
         readerChoosesSlot_Assertion(cs1) ∧
          read_Assertion(cs1) ∧
           cs2 = writerIndicatesPair(cs1) ⇒
     writerChoosesPair_Assertion(cs2) ∧
      firstReaderChoosesPair_Assertion(cs2) ∧
       readerChoosesPair_Assertion(cs2) ∧
        readerIndicatesPair_Assertion(cs2) ∧
         readerChoosesSlot_Assertion(cs2) ∧
          read_Assertion(cs2)
```

The remaining proof obligations are first to show that the required guarantee condition holds in the start location for each transition. In this case it follows immediately that the guarantee condition for the ACM holds since it is identical to the guarantee condition for each of the transitions. In the case of the writer the following proof obligations must be discharged:

vc2_op_name

$$\forall cs1 : Conc\_State \cdot$$
$$pre\_start\_writer\_op\_name(cs1) \wedge$$
$$start\_writer\_op\_name\_Assertion(cs1) \wedge$$
$$firstReaderChoosesPair\_Assertion(cs1) \wedge$$
$$readerChoosesPair\_Assertion(cs1) \wedge$$

$$
\begin{array}{l}
\quad readerIndicatesPair\_Assertion(cs1) \wedge \\
\quad\ readerChoosesSlot\_Assertion(cs1) \wedge \\
\quad\quad read\_Assertion(cs1) \Rightarrow \\
\quad (cs1.nri = rd \Rightarrow \\
\quad\ cs1.minFresh \leq cs1.newMaxFresh \wedge \\
\quad\quad cs1.indexRead \geq cs1.minFresh \wedge \\
\quad\quad\ cs1.indexRead \leq cs1.newMaxFresh \wedge \\
\quad\quad\quad cs1.lastIndexRead \leq cs1.indexRead)
\end{array}
$$

It is also necessary to show that the guarantee condition holds in the target location of the transition, as follows:

$$
\begin{array}{l}
vc3\_op\_name \\
\quad \forall cs1, cs2 : Conc\_State \cdot \\
\quad\ pre\_start\_writer\_op\_name(cs1) \wedge \\
\quad\quad start\_writer\_op\_name\_Assertion(cs1) \wedge \\
\quad\quad\ firstReaderChoosesPair\_Assertion(cs1) \wedge \\
\quad\quad\quad readerChoosesPair\_Assertion(cs1) \wedge \\
\quad\quad\quad readerIndicatesPair\_Assertion(cs1) \wedge \\
\quad\quad\quad\ readerChoosesSlot\_Assertion(cs1) \wedge \\
\quad\quad\quad\quad read\_Assertion(cs1) \wedge \\
\quad\quad\quad\quad post\_writer\_op\_name(cs1, cs2) \Rightarrow \\
\quad\quad\quad (cs2.nri = rd \Rightarrow \\
\quad\quad\quad\ cs2.minFresh \leq cs2.newMaxFresh \wedge \\
\quad\quad\quad\quad cs2.indexRead \geq cs2.minFresh \wedge \\
\quad\quad\quad\quad\ cs2.indexRead \leq cs2.newMaxFresh \wedge \\
\quad\quad\quad\quad\quad cs2.lastIndexRead \leq cs2.indexRead)
\end{array}
$$

Similarly, for the reader, the following two proof obligations must be discharged:

$$
\begin{array}{l}
vc2\_op\_name \\
\quad \forall cs1 : Conc\_State \cdot \\
\quad\ pre\_start\_reader\_op\_name(cs1) \wedge \\
\quad\quad start\_reader\_op\_name\_Assertion(cs1) \wedge \\
\quad\quad\ firstWriterChoosesPair\_Assertion(cs1) \wedge \\
\quad\quad\quad writerChoosesPair\_Assertion(cs1) \wedge \\
\quad\quad\quad writerChoosesSlot\_Assertion(cs1) \wedge \\
\quad\quad\quad\ write\_Assertion(cs1) \wedge \\
\quad\quad\quad\quad writerIndicatesSlot\_Assertion(cs1) \wedge \\
\quad\quad\quad\quad\ writerIndicatesPair\_Assertion(cs1) \Rightarrow \\
\quad\quad\quad (cs1.nri = rd \Rightarrow \\
\quad\quad\quad\ cs1.minFresh \leq cs1.newMaxFresh \wedge \\
\quad\quad\quad\quad cs1.indexRead \geq cs1.minFresh \wedge \\
\quad\quad\quad\quad\ cs1.indexRead \leq cs1.newMaxFresh \wedge \\
\quad\quad\quad\quad\quad cs1.lastIndexRead \leq cs1.indexRead)
\end{array}
$$

$$
\begin{array}{l}
vc3\_op\_name \\
\quad \forall cs1, cs2 : Conc\_State \cdot \\
\quad\; pre\_start\_reader\_op\_name(cs1) \wedge \\
\quad\;\; start\_reader\_op\_name\_Assertion(cs1) \wedge \\
\quad\;\;\; firstWriterChoosesPair\_Assertion(cs1) \wedge \\
\quad\;\;\;\; writerChoosesPair\_Assertion(cs1) \wedge \\
\quad\;\;\;\;\; writerChoosesSlot\_Assertion(cs1) \wedge \\
\quad\;\;\;\;\;\; write\_Assertion(cs1) \wedge \\
\quad\;\;\;\;\;\;\; writerIndicatesSlot\_Assertion(cs1) \wedge \\
\quad\;\;\;\;\;\;\;\; writerIndicatesPair\_Assertion(cs1) \wedge \\
\quad\;\;\;\;\;\;\;\;\; post\_reader\_op\_name(cs1, cs2) \Rightarrow \\
\quad\;\;\;\;\;\;\; (cs1.nri = rd \Rightarrow \\
\quad\;\;\;\;\;\;\;\; cs2.minFresh \leq cs2.newMaxFresh \wedge \\
\quad\;\;\;\;\;\;\;\;\; cs2.indexRead \geq cs2.minFresh \wedge \\
\quad\;\;\;\;\;\;\;\;\;\; cs2.indexRead \leq cs2.newMaxFresh \wedge \\
\quad\;\;\;\;\;\;\;\;\;\;\; cs2.lastIndexRead \leq cs2.indexRead)
\end{array}
$$

```
vc2_firstReaderChoosesPair: THEOREM
  ∀ (cs: Conc_State):
    pre_firstReaderChoosesPair(cs) ∧
     firstReaderChoosesPair_Assertion(cs) ∧
      firstWriterChoosesPair_Assertion(cs) ∧
       writerChoosesPair_Assertion(cs) ∧
        writerChoosesSlot_Assertion(cs) ∧
         write_Assertion(cs) ∧
          writerIndicatesSlot_Assertion(cs) ∧
           writerIndicatesPair_Assertion(cs) ⇒
     (cs'nri = rd ⇒
       cs'minFresh ≤ cs'newMaxFresh ∧
        cs'indexRead ≥ cs'minFresh ∧
         cs'indexRead ≤ cs'newMaxFresh ∧
          cs'lastIndexRead ≤ cs'indexRead)

vc3_firstReaderChoosesPair: THEOREM
  ∀ (cs1, cs2: Conc_State):
    pre_firstReaderChoosesPair(cs1) ∧
     firstReaderChoosesPair_Assertion(cs1) ∧
      cs2 = firstReaderChoosesPair(cs1) ∧
       firstWriterChoosesPair_Assertion(cs1) ∧
        writerChoosesPair_Assertion(cs1) ∧
         writerChoosesSlot_Assertion(cs1) ∧
          write_Assertion(cs1) ∧
           writerIndicatesSlot_Assertion(cs1) ∧
            writerIndicatesPair_Assertion(cs1) ⇒
     (cs2'nri = rd ⇒
       cs2'minFresh ≤ cs2'newMaxFresh ∧
        cs2'indexRead ≥ cs2'minFresh ∧
         cs2'indexRead ≤ cs2'newMaxFresh ∧
          cs2'lastIndexRead ≤ cs2'indexRead)

vc2_readerChoosesPair: THEOREM
  ∀ (cs: Conc_State):
    pre_readerChoosesPair(cs) ∧
     readerChoosesPair_Assertion(cs) ∧
      firstWriterChoosesPair_Assertion(cs) ∧
```

```
        writerChoosesPair_Assertion(cs) ∧
         writerChoosesSlot_Assertion(cs) ∧
          write_Assertion(cs) ∧
           writerIndicatesSlot_Assertion(cs) ∧
            writerIndicatesPair_Assertion(cs) ⇒
      (cs'nri = rd ⇒
        cs'minFresh ≤ cs'newMaxFresh ∧
         cs'indexRead ≥ cs'minFresh ∧
          cs'indexRead ≤ cs'newMaxFresh ∧
           cs'lastIndexRead ≤ cs'indexRead)

vc3_readerChoosesPair: THEOREM
  ∀ (cs1, cs2: Conc_State):
    pre_readerChoosesPair(cs1) ∧
     readerChoosesPair_Assertion(cs1) ∧
      cs2 = readerChoosesPair(cs1) ∧
       firstWriterChoosesPair_Assertion(cs1) ∧
        writerChoosesPair_Assertion(cs1) ∧
         writerChoosesSlot_Assertion(cs1) ∧
          write_Assertion(cs1) ∧
           writerIndicatesSlot_Assertion(cs1) ∧
            writerIndicatesPair_Assertion(cs1) ⇒
      (cs2'nri = rd ⇒
        cs2'minFresh ≤ cs2'newMaxFresh ∧
         cs2'indexRead ≥ cs2'minFresh ∧
          cs2'indexRead ≤ cs2'newMaxFresh ∧
           cs2'lastIndexRead ≤ cs2'indexRead)

vc2_readerIndicatesPair: THEOREM
  ∀ (cs: Conc_State):
    pre_readerIndicatesPair(cs) ∧
     readerIndicatesPair_Assertion(cs) ∧
      firstWriterChoosesPair_Assertion(cs) ∧
       writerChoosesPair_Assertion(cs) ∧
        writerChoosesSlot_Assertion(cs) ∧
         write_Assertion(cs) ∧
          writerIndicatesSlot_Assertion(cs) ∧
           writerIndicatesPair_Assertion(cs) ⇒
      (cs'nri = rd ⇒
        cs'minFresh ≤ cs'newMaxFresh ∧
         cs'indexRead ≥ cs'minFresh ∧
          cs'indexRead ≤ cs'newMaxFresh ∧
           cs'lastIndexRead ≤ cs'indexRead)

vc3_readerIndicatesPair: THEOREM
  ∀ (cs1, cs2: Conc_State):
    pre_readerIndicatesPair(cs1) ∧
     readerIndicatesPair_Assertion(cs1) ∧
      cs2 = readerIndicatesPair(cs1) ∧
       firstWriterChoosesPair_Assertion(cs1) ∧
        writerChoosesPair_Assertion(cs1) ∧
         writerChoosesSlot_Assertion(cs1) ∧
          write_Assertion(cs1) ∧
           writerIndicatesSlot_Assertion(cs1) ∧
            writerIndicatesPair_Assertion(cs1) ⇒
      (cs2'nri = rd ⇒
        cs2'minFresh ≤ cs2'newMaxFresh ∧
         cs2'indexRead ≥ cs2'minFresh ∧
          cs2'indexRead ≤ cs2'newMaxFresh ∧
           cs2'lastIndexRead ≤ cs2'indexRead)

vc2_readerChoosesSlot: THEOREM
  ∀ (cs: Conc_State):
    pre_readerChoosesSlot(cs) ∧
```

```
      readerChoosesPair_Assertion(cs) ∧
       firstWriterChoosesPair_Assertion(cs) ∧
        writerChoosesPair_Assertion(cs) ∧
         writerChoosesSlot_Assertion(cs) ∧
          write_Assertion(cs) ∧
           writerIndicatesSlot_Assertion(cs) ∧
            writerIndicatesPair_Assertion(cs) ⇒
      (cs'nri = rd ⇒
        cs'minFresh ≤ cs'newMaxFresh ∧
         cs'indexRead ≥ cs'minFresh ∧
          cs'indexRead ≤ cs'newMaxFresh ∧
           cs'lastIndexRead ≤ cs'indexRead)

vc3_readerChoosesSlot: THEOREM
  ∀ (cs1, cs2: Conc_State):
    pre_readerChoosesSlot(cs1) ∧
     readerChoosesSlot_Assertion(cs1) ∧
      cs2 = readerChoosesSlot(cs1) ∧
       firstWriterChoosesPair_Assertion(cs1) ∧
        writerChoosesPair_Assertion(cs1) ∧
         writerChoosesSlot_Assertion(cs1) ∧
          write_Assertion(cs1) ∧
           writerIndicatesSlot_Assertion(cs1) ∧
            writerIndicatesPair_Assertion(cs1) ⇒
      (cs2'nri = rd ⇒
        cs2'minFresh ≤ cs2'newMaxFresh ∧
         cs2'indexRead ≥ cs2'minFresh ∧
          cs2'indexRead ≤ cs2'newMaxFresh ∧
           cs2'lastIndexRead ≤ cs2'indexRead)

vc2_read: THEOREM
  ∀ (cs: Conc_State, v: Val):
    pre_read(cs) ∧
     read_Assertion(cs) ∧
      firstWriterChoosesPair_Assertion(cs) ∧
       writerChoosesPair_Assertion(cs) ∧
        writerChoosesSlot_Assertion(cs) ∧
         write_Assertion(cs) ∧
          writerIndicatesSlot_Assertion(cs) ∧
           writerIndicatesPair_Assertion(cs) ⇒
      (cs'nri = rd ⇒
        cs'minFresh ≤ cs'newMaxFresh ∧
         cs'indexRead ≥ cs'minFresh ∧
          cs'indexRead ≤ cs'newMaxFresh ∧
           cs'lastIndexRead ≤ cs'indexRead)

vc3_read: THEOREM
  ∀ (cs1, cs2: Conc_State, v: Val):
    pre_read(cs1) ∧
     read_Assertion(cs1) ∧
      cs2 = read(cs1)'1 ∧
       firstWriterChoosesPair_Assertion(cs1) ∧
        writerChoosesPair_Assertion(cs1) ∧
         writerChoosesSlot_Assertion(cs1) ∧
          write_Assertion(cs1) ∧
           writerIndicatesSlot_Assertion(cs1) ∧
            writerIndicatesPair_Assertion(cs1) ⇒
      (cs2'nri = rd ⇒
        cs2'minFresh ≤ cs2'newMaxFresh ∧
         cs2'indexRead ≥ cs2'minFresh ∧
          cs2'indexRead ≤ cs2'newMaxFresh ∧
           cs2'lastIndexRead ≤ cs2'indexRead)

vc2_firstWriterChoosesPair: THEOREM
```

```
  ∀ (cs: Conc_State):
    pre_firstWriterChoosesPair(cs) ∧
     firstWriterChoosesPair_Assertion(cs) ∧
      firstReaderChoosesPair_Assertion(cs) ∧
       readerChoosesPair_Assertion(cs) ∧
        readerIndicatesPair_Assertion(cs) ∧
         readerChoosesSlot_Assertion(cs) ∧
          read_Assertion(cs) ⇒
     (cs'nri = rd ⇒
       cs'minFresh ≤ cs'newMaxFresh ∧
        cs'indexRead ≥ cs'minFresh ∧
         cs'indexRead ≤ cs'newMaxFresh ∧
          cs'lastIndexRead ≤ cs'indexRead)

vc3_firstWriterChoosesPair: THEOREM
  ∀ (cs1, cs2: Conc_State):
    pre_firstWriterChoosesPair(cs1) ∧
     firstWriterChoosesPair_Assertion(cs1) ∧
      cs2 = firstWriterChoosesPair(cs1) ∧
       firstReaderChoosesPair_Assertion(cs1) ∧
        readerChoosesPair_Assertion(cs1) ∧
         readerIndicatesPair_Assertion(cs1) ∧
          readerChoosesSlot_Assertion(cs1) ∧
           read_Assertion(cs1) ⇒
     (cs2'nri = rd ⇒
       cs2'minFresh ≤ cs2'newMaxFresh ∧
        cs2'indexRead ≥ cs2'minFresh ∧
         cs2'indexRead ≤ cs2'newMaxFresh ∧
          cs2'lastIndexRead ≤ cs2'indexRead)

vc2_writerChoosesPair: THEOREM
  ∀ (cs: Conc_State):
    pre_writerChoosesPair(cs) ∧
     writerChoosesPair_Assertion(cs) ∧
      firstReaderChoosesPair_Assertion(cs) ∧
       readerChoosesPair_Assertion(cs) ∧
        readerIndicatesPair_Assertion(cs) ∧
         readerChoosesSlot_Assertion(cs) ∧
          read_Assertion(cs) ⇒
     (cs'nri = rd ⇒
       cs'minFresh ≤ cs'newMaxFresh ∧
        cs'indexRead ≥ cs'minFresh ∧
         cs'indexRead ≤ cs'newMaxFresh ∧
          cs'lastIndexRead ≤ cs'indexRead)

vc3_writerChoosesPair: THEOREM
  ∀ (cs1, cs2: Conc_State):
    pre_writerChoosesPair(cs1) ∧
     writerChoosesPair_Assertion(cs1) ∧
      cs2 = writerChoosesPair(cs1) ∧
       firstReaderChoosesPair_Assertion(cs1) ∧
        readerChoosesPair_Assertion(cs1) ∧
         readerIndicatesPair_Assertion(cs1) ∧
          readerChoosesSlot_Assertion(cs1) ∧
           read_Assertion(cs1) ⇒
     (cs2'nri = rd ⇒
       cs2'minFresh ≤ cs2'newMaxFresh ∧
        cs2'indexRead ≥ cs2'minFresh ∧
         cs2'indexRead ≤ cs2'newMaxFresh ∧
          cs2'lastIndexRead ≤ cs2'indexRead)

vc2_writerChoosesSlot: THEOREM
  ∀ (cs: Conc_State):
    pre_writerChoosesSlot(cs) ∧
```

```
      writerChoosesSlot_Assertion(cs) ∧
       firstReaderChoosesPair_Assertion(cs) ∧
        readerChoosesPair_Assertion(cs) ∧
         readerIndicatesPair_Assertion(cs) ∧
          readerChoosesSlot_Assertion(cs) ∧
           read_Assertion(cs) ⇒
      (cs'nri = rd ⇒
        cs'minFresh ≤ cs'newMaxFresh ∧
         cs'indexRead ≥ cs'minFresh ∧
          cs'indexRead ≤ cs'newMaxFresh ∧
           cs'lastIndexRead ≤ cs'indexRead)

vc3_writerChoosesSlot: THEOREM
  ∀ (cs1, cs2: Conc_State):
    pre_writerChoosesSlot(cs1) ∧
     writerChoosesSlot_Assertion(cs1) ∧
      cs2 = writerChoosesSlot(cs1) ∧
       firstReaderChoosesPair_Assertion(cs1) ∧
        readerChoosesPair_Assertion(cs1) ∧
         readerIndicatesPair_Assertion(cs1) ∧
          readerChoosesSlot_Assertion(cs1) ∧
           read_Assertion(cs1) ⇒
      (cs2'nri = rd ⇒
        cs2'minFresh ≤ cs2'newMaxFresh ∧
         cs2'indexRead ≥ cs2'minFresh ∧
          cs2'indexRead ≤ cs2'newMaxFresh ∧
           cs2'lastIndexRead ≤ cs2'indexRead)

vc2_write: THEOREM
  ∀ (cs: Conc_State):
    pre_write(cs) ∧
     write_Assertion(cs) ∧
      firstReaderChoosesPair_Assertion(cs) ∧
       readerChoosesPair_Assertion(cs) ∧
        readerIndicatesPair_Assertion(cs) ∧
         readerChoosesSlot_Assertion(cs) ∧
          read_Assertion(cs) ⇒
      (cs'nri = rd ⇒
        cs'minFresh ≤ cs'newMaxFresh ∧
         cs'indexRead ≥ cs'minFresh ∧
          cs'indexRead ≤ cs'newMaxFresh ∧
           cs'lastIndexRead ≤ cs'indexRead)

vc3_write: THEOREM
  ∀ (w: write_parameter, cs2: Conc_State):
    pre_write(w'p1) ∧
     write_Assertion(w'p1) ∧
      cs2 = write(w) ∧
       firstReaderChoosesPair_Assertion(w'p1) ∧
        readerChoosesPair_Assertion(w'p1) ∧
         readerIndicatesPair_Assertion(w'p1) ∧
          readerChoosesSlot_Assertion(w'p1) ∧
           read_Assertion(w'p1) ⇒
      (cs2'nri = rd ⇒
        cs2'minFresh ≤ cs2'newMaxFresh ∧
         cs2'indexRead ≥ cs2'minFresh ∧
          cs2'indexRead ≤ cs2'newMaxFresh ∧
           cs2'lastIndexRead ≤ cs2'indexRead)

vc2_writerIndicatesSlot: THEOREM
  ∀ (cs: Conc_State):
    pre_writerIndicatesSlot(cs) ∧
     writerIndicatesSlot_Assertion(cs) ∧
      firstReaderChoosesPair_Assertion(cs) ∧
```

```
        readerChoosesPair_Assertion(cs) ∧
         readerIndicatesPair_Assertion(cs) ∧
          readerChoosesSlot_Assertion(cs) ∧
           read_Assertion(cs) ⇒
      (cs'nri = rd ⇒
        cs'minFresh ≤ cs'newMaxFresh ∧
         cs'indexRead ≥ cs'minFresh ∧
          cs'indexRead ≤ cs'newMaxFresh ∧
           cs'lastIndexRead ≤ cs'indexRead)

vc3_writerIndicatesSlot: THEOREM
  ∀ (cs1, cs2: Conc_State):
    pre_writerIndicatesSlot(cs1) ∧
     writerIndicatesSlot_Assertion(cs1) ∧
      cs2 = writerIndicatesSlot(cs1) ∧
       firstReaderChoosesPair_Assertion(cs1) ∧
        readerChoosesPair_Assertion(cs1) ∧
         readerIndicatesPair_Assertion(cs1) ∧
          readerChoosesSlot_Assertion(cs1) ∧
           read_Assertion(cs1) ⇒
      (cs2'nri = rd ⇒
        cs2'minFresh ≤ cs2'newMaxFresh ∧
         cs2'indexRead ≥ cs2'minFresh ∧
          cs2'indexRead ≤ cs2'newMaxFresh ∧
           cs2'lastIndexRead ≤ cs2'indexRead)

vc2_writerIndicatesPair: THEOREM
  ∀ (cs: Conc_State):
    pre_writerIndicatesPair(cs) ∧
     writerIndicatesPair_Assertion(cs) ∧
      firstReaderChoosesPair_Assertion(cs) ∧
       readerChoosesPair_Assertion(cs) ∧
        readerIndicatesPair_Assertion(cs) ∧
         readerChoosesSlot_Assertion(cs) ∧
          read_Assertion(cs) ⇒
      (cs'nri = rd ⇒
        cs'minFresh ≤ cs'newMaxFresh ∧
         cs'indexRead ≥ cs'minFresh ∧
          cs'indexRead ≤ cs'newMaxFresh ∧
           cs'lastIndexRead ≤ cs'indexRead)

vc3_writerIndicatesPair: THEOREM
  ∀ (cs1, cs2: Conc_State):
    pre_writerIndicatesPair(cs1) ∧
     writerIndicatesPair_Assertion(cs1) ∧
      cs2 = writerIndicatesPair(cs1) ∧
       firstReaderChoosesPair_Assertion(cs1) ∧
        readerChoosesPair_Assertion(cs1) ∧
         readerIndicatesPair_Assertion(cs1) ∧
          readerChoosesSlot_Assertion(cs1) ∧
           read_Assertion(cs1) ⇒
      (cs2'nri = rd ⇒
        cs2'minFresh ≤ cs2'newMaxFresh ∧
         cs2'indexRead ≥ cs2'minFresh ∧
          cs2'indexRead ≤ cs2'newMaxFresh ∧
           cs2'lastIndexRead ≤ cs2'indexRead)

END FOUR_SLOT
```

Appendix I

3-slot ACM Implementations

This appendix gives three formal models of 3-slot ACM implementations: first the implementation from [Sim90a], which is proved to be faulty; second a model which shows that the above implementation is L-atomic, provided that the timing constraint in [Sim90a] can be implemented. Finally an implementation from [XYIS02], where the reader does not keep a local copy of the slot it has chosen to access: it copies the name of the slot directly between the control variables in the mechanism, and uses the value of the *slotReading* control variable to access its chosen slot. This ACM is L-atomic, provided that the access to the control variables is Hoare atomic.

I.1 The Implementation from [Sim90a]

```
THREE_SLOT: THEORY
BEGIN
```

A non-empty type of values that is communicated by the ACM.

```
Val: NONEMPTY_TYPE
```

A *SlotIndex* type to represent the names of the slots in the ACM.

```
SlotIndex: TYPE = {s0, s1, s2}
```

The program counters for the reader and writer.

```
NextReadInstruction: TYPE = {firstRcs, rcs, ris, rd}

NextWriteInstruction: TYPE = {firstWcs, wcs, wr, wis}
```

The locations in the reader and writer assertion networks.

```
ReaderNetworkState: TYPE = {sr, lr1, lr2, lr3, tr}

WriterNetworkState: TYPE = {sw, lw1, lw2, lw3, tw}
```

The local state of the writer and reader.

```
WriterState: TYPE =
[# writerSlot: SlotIndex, currentState: WriterNetworkState #]

ReaderState: TYPE =
[# readerSlot: SlotIndex, currentState: ReaderNetworkState #]
```

The state of the mechanism - the control variables for the writer and reader to record the slot they are accessing, the slots, the program counters for the reader and writer and the reader and writer local states.

```
Conc_State: TYPE =
[# slotWritten: SlotIndex,
   slotReading: SlotIndex,
   slots: [SlotIndex → Val],
   nri: NextReadInstruction,
   nwi: NextWriteInstruction,
   writer: WriterState,
   reader: ReaderState #]
```

The operations of the reader and writer. The first reader operation at start up is *firstReaderChoosesSlot*. The pre-condition of this operation is that the reader is in the initial location in its assertion network (nri = firstRcs): the operation chooses the slot the reader is going to access, and changes the program counter to indicate that the reader can now execute *readerIndicatesSlot*. The *readerChoosesSlot* operation has an identical post-condition, but its pre-condition is that the reader can next execute *readerChoosesSlot*, rather than *firstReaderChoosesSlot*. The pre-conditions for the remaining operations are simply that the reader (or writer) program counter is equal to the correct value for the operation to be executed. These program counters are auxiliary variables that are not part of the implementation. The operations each set the respective program counter to the correct value for the next operation to be executed.

```
pre_firstReaderChoosesSlot(p: Conc_State): bool = p'nri = firstRcs

post_firstReaderChoosesSlot(p: (pre_firstReaderChoosesSlot))(prot: Conc_State): bool =
    prot = p WITH [nri := ris,
                   reader := p'reader WITH [readerSlot := p'slotWritten,
                                            currentState := lr1]]

firstReaderChoosesSlot:
[p: (pre_firstReaderChoosesSlot) → (post_firstReaderChoosesSlot(p))]

pre_readerChoosesSlot(p: Conc_State): bool = p'nri = rcs

post_readerChoosesSlot(p: (pre_readerChoosesSlot))(prot: Conc_State): bool =
    prot = p WITH [nri := ris,
                   reader := p'reader WITH [readerSlot := p'slotWritten,
                                            currentState := lr1]]

readerChoosesSlot:
[p: (pre_readerChoosesSlot) → (post_readerChoosesSlot(p))]
```

The post-condition for the *readerIndicatesSlot* operation is that the reader has indicated the slot it is accessing in the control variable *slotReading*.

```
pre_readerIndicatesSlot(p: Conc_State): bool = p'nri = ris

post_readerIndicatesSlot(p: (pre_readerIndicatesSlot))(prot: Conc_State): bool =
    prot = p WITH [nri := rd,
                   slotReading := p'reader'readerSlot,
                   reader := p'reader WITH [currentState := lr2]]

readerIndicatesSlot:
[p: (pre_readerIndicatesSlot) → (post_readerIndicatesSlot(p))]
```

The post-condition of the *read* operation returns that value read from the ACM.

```
pre_read(p: Conc_State): bool = p'nri = rd

post_read(p: (pre_read))(prot: Conc_State, v: Val): bool =
    v = p'slots(p'reader'readerSlot) ∧
     prot = p WITH [nri := rcs, reader := p'reader WITH [currentState := lr3]]

read: [p: (pre_read) → (post_read(p))]
```

The post-conditions for the *firstWriterChosesSlot* and *writerChoosesSlot* operations are that the writer has chosen the slot it is going to write the new value to. The writer attempts to avoid the slot that the reader is accessing (by choosing to write to a different slot to the one the reader last indicated it was accessing), and also avoids the slot that it last accessed.

```
pre_firstWriterChoosesSlot(p: Conc_State): bool = p'nwi = firstWcs

post_firstWriterChoosesSlot(p: (pre_firstWriterChoosesSlot))(prot: Conc_State): bool =
 (p'slotWritten = s0 ⇒
   (p'slotReading = s0 ⇒
     prot = p WITH [nwi := wr,
                    writer := p'writer WITH
                                [writerSlot := s1, currentState := lw1]]) ∧
     (p'slotReading = s1 ⇒
       prot = p WITH [nwi := wr,
                      writer := p'writer WITH
                                  [writerSlot := s2, currentState := lw1]]) ∧
      (p'slotReading = s2 ⇒
        prot = p WITH [nwi := wr,
                       writer := p'writer WITH
                                   [writerSlot := s1, currentState := lw1]])) ∧
 (p'slotWritten = s1 ⇒
   (p'slotReading = s0 ⇒
     prot = p WITH [nwi := wr,
                    writer := p'writer WITH
                                [writerSlot := s2, currentState := lw1]]) ∧
    (p'slotReading = s1 ⇒
      prot = p WITH [nwi := wr,
                     writer := p'writer WITH
                                 [writerSlot := s2, currentState := lw1]]) ∧
     (p'slotReading = s2 ⇒
       prot = p WITH [nwi := wr,
                      writer := p'writer WITH
                                  [writerSlot := s0, currentState := lw1]])) ∧
 (p'slotWritten = s2 ⇒
```

```
    (p'slotReading = s0 ⇒
      prot = p WITH [nwi := wr,
                     writer := p'writer WITH
                               [writerSlot := s1, currentState := lw1]]) ∧
     (p'slotReading = s1 ⇒
       prot = p WITH [nwi := wr,
                      writer := p'writer WITH
                                [writerSlot := s0, currentState := lw1]]) ∧
      (p'slotReading = s2 ⇒
        prot = p WITH [nwi := wr,
                       writer := p'writer WITH
                                 [writerSlot := s0, currentState := lw1]]))

firstWriterChoosesSlot:
[p: (pre_firstWriterChoosesSlot) → (post_firstWriterChoosesSlot(p))]

pre_writerChoosesSlot(p: Conc_State): bool = p'nwi = wcs

post_writerChoosesSlot(p: (pre_writerChoosesSlot))(prot: Conc_State): bool =
 (p'slotWritten = s0 ⇒
   (p'slotReading = s0 ⇒
     prot = p WITH [nwi := wr,
                    writer := p'writer WITH
                              [writerSlot := s1, currentState := lw1]]) ∧
     (p'slotReading = s1 ⇒
       prot = p WITH [nwi := wr,
                      writer := p'writer WITH
                                [writerSlot := s2, currentState := lw1]]) ∧
      (p'slotReading = s2 ⇒
        prot = p WITH [nwi := wr,
                       writer := p'writer WITH
                                 [writerSlot := s1, currentState := lw1]])) ∧
 (p'slotWritten = s1 ⇒
   (p'slotReading = s0 ⇒
     prot = p WITH [nwi := wr,
                    writer := p'writer WITH
                              [writerSlot := s2, currentState := lw1]]) ∧
    (p'slotReading = s1 ⇒
      prot = p WITH [nwi := wr,
                     writer := p'writer WITH
                               [writerSlot := s2, currentState := lw1]]) ∧
     (p'slotReading = s2 ⇒
       prot = p WITH [nwi := wr,
                      writer := p'writer WITH
                                [writerSlot := s0, currentState := lw1]])) ∧
  (p'slotWritten = s2 ⇒
    (p'slotReading = s0 ⇒
      prot = p WITH [nwi := wr,
                     writer := p'writer WITH
                               [writerSlot := s1, currentState := lw1]]) ∧
     (p'slotReading = s1 ⇒
       prot = p WITH [nwi := wr,
                      writer := p'writer WITH
                                [writerSlot := s0, currentState := lw1]]) ∧
      (p'slotReading = s2 ⇒
        prot = p WITH [nwi := wr,
                       writer := p'writer WITH
                                 [writerSlot := s0, currentState := lw1]]))

writerChoosesSlot:
[p: (pre_writerChoosesSlot) → (post_writerChoosesSlot(p))]
```

The post-condition for the *write* operation is that the writer has written the new item to its chosen slot.

```
pre_write(p: Conc_State): bool = p'nwi = wr

write_parameter: TYPE = [# p1: (pre_write), v: Val #]

post_write(p: write_parameter)(prot: Conc_State): bool =
    prot = p'p1 WITH [nwi := wis,
                      (slots)(p'p1'writer'writerSlot) := p'v,
                      writer := p'p1'writer WITH [currentState := lw2]]

write: [p: write_parameter → (post_write(p))]
```

The post-condition of the *writerIndicatesSlot* operation is that the writer has indicated the slot it has accessed in the relevant control variable in the mechanism.

```
pre_writerIndicatesSlot(p: Conc_State): bool = p'nwi = wis

post_writerIndicatesSlot(p: (pre_writerIndicatesSlot))(prot: Conc_State): bool =
    prot = p WITH [nwi := wis,
                   slotWritten := (p'writer'writerSlot),
                   writer := p'writer WITH [currentState := lw3]]

writerIndicatesSlot:
[p: (pre_writerIndicatesSlot) → (post_writerIndicatesSlot(p))]
```

Initialisation operations for the reader and writer local states, and for the ACM itself.

```
init_writer(w: WriterState): bool = w = w WITH [currentState := sw]

init_reader(r: ReaderState): bool = r = r WITH [currentState := sr]

init_prot(p: Conc_State, init_val: Val, w: WriterState, r: ReaderState): bool =
    p = p WITH [slotWritten := s0,
                slotReading := s0,
                slots := (λ ·(s0: SlotIndex): init_val),
                nri := firstRcs,
                nwi := firstWcs,
                writer := w,
                reader := r]
```

The following are the assertions from the locations in the reader and writer assertion networks.

When the writer is about to execute the *writerChoosesSlot* operation the *slotWritten* control variable is equal to the writer local variable, *writerSlot*.

```
writerChoosesSlot_Assertion: [Conc_State → bool] =
    (λ ·(cs: Conc_State):
      cs'nwi = wcs ⇒ cs'slotWritten = cs'writer'writerSlot
```

When the writer is writing (or about to write) to the mechanism, it has chosen to access a different slot to the one it accessed for the previous write. This same assertion holds when the write is about to execute the *writerIndicatesSlot* operation.

```
write_Assertion: [Conc_State → bool] =
    (λ ·(cs: Conc_State):
      cs'nwi = wr ⇒ ¬ cs'slotWritten = cs'writer'writerSlot

writerIndicatesSlot_Assertion: [Conc_State → bool] =
    (λ ·(cs: Conc_State):
      cs'nwi = wis ⇒ ¬ cs'slotWritten = cs'writer'writerSlot
```

When the reader is about to execute the *readerChoosesSlot* operation the *slotReading* control variable is equal to the reader local variable, *readerSlot*. It is not possible to make any assertions when the reader is about to execute the *readerIndicatesSlot* operation.

```
readerChoosesSlot_Assertion: [Conc_State → bool] =
    (λ ·(cs: Conc_State):
      cs'nri = rcs ⇒ cs'slotReading = cs'reader'readerSlot
```

When the reader is reading (or to about to read) from the mechanism the control variable *slotReading* is equal to the reader local variable, *readerSlot*.

```
read_Assertion: [Conc_State → bool] =
    (λ ·(cs: Conc_State):
      cs'nri = rd ⇒ cs'slotReading = cs'reader'readerSlot
```

The following are the proof obligations that need to be executed to verify that the 3-slot ACM communicates coherent data between the reader and writer (that the reader and writer never access the same slot at the same time). The first proof obligation for each transition (vc1) in the respective assertion networks shows, when the pre-condition for the operation associated with the transition holds and the assertion in the start location of the transition holds, that the assertion in the target location of the operation will hold after the operation is executed. Additionally it shows that the reader operations do not interfere with the assertions in the writer network, and that the writer operations do not interfere with the operations in the writer network. In each case the relevant transition is indicated by the name of its associated operation.

```
vc1_firstWriterChoosesSlot: THEOREM
  ∀ (cs1, cs2: Conc_State):
    pre_firstWriterChoosesSlot(cs1) ∧
     readerChoosesSlot_Assertion(cs1) ∧
      read_Assertion(cs1) ∧
       cs2 = firstWriterChoosesSlot(cs1) ⇒
     write_Assertion(cs2) ∧
      readerChoosesSlot_Assertion(cs2) ∧ read_Assertion(cs2)

vc1_writerChoosesSlot: THEOREM
  ∀ (cs1, cs2: Conc_State):
    pre_writerChoosesSlot(cs1) ∧
     writerChoosesSlot_Assertion(cs1) ∧
      readerChoosesSlot_Assertion(cs1) ∧
       read_Assertion(cs1) ∧
        cs2 = writerChoosesSlot(cs1) ⇒
     write_Assertion(cs2) ∧
```

```
      readerChoosesSlot_Assertion(cs2) ∧ read_Assertion(cs2)

vc1_write: THEOREM
  ∀ (w: write_parameter, cs2: Conc_State):
    pre_write(w`p1) ∧
     write_Assertion(w`p1) ∧
      readerChoosesSlot_Assertion(w`p1) ∧
       read_Assertion(w`p1) ∧
        cs2 = write(w) ⇒
     writerIndicatesSlot_Assertion(cs2) ∧
      readerChoosesSlot_Assertion(cs2) ∧ read_Assertion(cs2)

vc1_writerIndicatesSlot: THEOREM
  ∀ (cs1, cs2: Conc_State):
    pre_writerIndicatesSlot(cs1) ∧
     writerIndicatesSlot_Assertion(cs1) ∧
      readerChoosesSlot_Assertion(cs1) ∧
       read_Assertion(cs1) ∧
        cs2 = writerIndicatesSlot(cs1) ⇒
     writerChoosesSlot_Assertion(cs2) ∧
      readerChoosesSlot_Assertion(cs2) ∧ read_Assertion(cs2)

vc1_readerChoosesSlot: THEOREM
  ∀ (cs1, cs2: Conc_State):
    pre_readerChoosesSlot(cs1) ∧
     readerChoosesSlot_Assertion(cs1) ∧
      writerChoosesSlot_Assertion(cs1) ∧
       write_Assertion(cs1) ∧
        writerIndicatesSlot_Assertion(cs1) ∧
         cs2 = readerChoosesSlot(cs1) ⇒
     writerChoosesSlot_Assertion(cs2) ∧
      write_Assertion(cs2) ∧ writerIndicatesSlot_Assertion(cs2)

vc1_readerIndicatesSlot: THEOREM
  ∀ (cs1, cs2: Conc_State):
    pre_readerIndicatesSlot(cs1) ∧
     writerChoosesSlot_Assertion(cs1) ∧
      write_Assertion(cs1) ∧
       writerIndicatesSlot_Assertion(cs1) ∧
        cs2 = readerIndicatesSlot(cs1) ⇒
     read_Assertion(cs2) ∧
      writerChoosesSlot_Assertion(cs2) ∧
       write_Assertion(cs2) ∧ writerIndicatesSlot_Assertion(cs2)

vc1_read: THEOREM
  ∀ (cs1, cs2: Conc_State):
    pre_read(cs1) ∧
     read_Assertion(cs1) ∧
      writerChoosesSlot_Assertion(cs1) ∧
       write_Assertion(cs1) ∧
        writerIndicatesSlot_Assertion(cs1) ∧
         cs2 = read(cs1)`1 ⇒
     readerChoosesSlot_Assertion(cs2) ∧
      writerChoosesSlot_Assertion(cs2) ∧
       write_Assertion(cs2) ∧ writerIndicatesSlot_Assertion(cs2)
```

The remaining proof obligations are to show that the guarantee condition holds in the start location of each operation, and that it also holds after each of the operations is executed. The guarantee condition is that the reader and writer will access different slots when they are reading from and writing to the ACM. Stated formally:

$nwi = wr \wedge nri = rd \Rightarrow readerSlot \neq writerSlot$

It is not possible to complete this proof obligation to show that the guarantee condition holds when the *readerIndicatesSlot* operation is executed.

```
vc2_firstReaderChoosesSlot: THEOREM
  ∀ (cs1: Conc_State):
    pre_firstReaderChoosesSlot(cs1) ∧
     writerChoosesSlot_Assertion(cs1) ∧
      write_Assertion(cs1) ∧
       writerIndicatesSlot_Assertion(cs1) ⇒
     (cs1'nwi = wr ∧ cs1'nri = rd ⇒
       ¬ cs1'reader'readerSlot = cs1'writer'writerSlot)

vc3_firstReaderChoosesSlot: THEOREM
  ∀ (cs1: Conc_State):
    pre_firstReaderChoosesSlot(cs1) ∧
     writerChoosesSlot_Assertion(cs1) ∧
      write_Assertion(cs1) ∧
       writerIndicatesSlot_Assertion(cs1) ⇒
     (firstReaderChoosesSlot(cs1)'nwi = wr ∧ firstReaderChoosesSlot(cs1)'nri = rd ⇒
       ¬ (firstReaderChoosesSlot(cs1)'reader'readerSlot =
            firstReaderChoosesSlot(cs1)'writer'writerSlot))

vc2_readerChoosesSlot: THEOREM
  ∀ (cs1: Conc_State):
    pre_readerChoosesSlot(cs1) ∧
     readerChoosesSlot_Assertion(cs1) ∧
      writerChoosesSlot_Assertion(cs1) ∧
       write_Assertion(cs1) ∧ writerIndicatesSlot_Assertion(cs1) ⇒
     (cs1'nwi = wr ∧ cs1'nri = rd ⇒
       ¬ cs1'reader'readerSlot = cs1'writer'writerSlot)

vc3_readerChoosesSlot: THEOREM
  ∀ (cs1: Conc_State):
    pre_readerChoosesSlot(cs1) ∧
     readerChoosesSlot_Assertion(cs1) ∧
      writerChoosesSlot_Assertion(cs1) ∧
       write_Assertion(cs1) ∧ writerIndicatesSlot_Assertion(cs1) ⇒
     (readerChoosesSlot(cs1)'nwi = wr ∧ readerChoosesSlot(cs1)'nri = rd ⇒
       ¬ (readerChoosesSlot(cs1)'reader'readerSlot =
            readerChoosesSlot(cs1)'writer'writerSlot))

vc2_readerIndicatesSlot: THEOREM
  ∀ (cs1: Conc_State):
    pre_readerIndicatesSlot(cs1) ∧
     writerChoosesSlot_Assertion(cs1) ∧
      write_Assertion(cs1) ∧
       writerIndicatesSlot_Assertion(cs1) ⇒
     (cs1'nwi = wr ∧ cs1'nri = rd ⇒
       ¬ cs1'reader'readerSlot = cs1'writer'writerSlot)

vc3_readerIndicatesSlot: THEOREM
  ∀ (cs1: Conc_State):
    pre_readerIndicatesSlot(cs1) ∧
     writerChoosesSlot_Assertion(cs1) ∧
      write_Assertion(cs1) ∧
       writerIndicatesSlot_Assertion(cs1) ⇒
     (readerIndicatesSlot(cs1)'nwi = wr ∧ readerIndicatesSlot(cs1)'nri = rd ⇒
       ¬ (readerIndicatesSlot(cs1)'reader'readerSlot =
            readerIndicatesSlot(cs1)'writer'writerSlot))

vc2_read: THEOREM
  ∀ (cs1: Conc_State):
```

```
    pre_read(cs1) ∧
     read_Assertion(cs1) ∧
      writerChoosesSlot_Assertion(cs1) ∧
       write_Assertion(cs1) ∧ writerIndicatesSlot_Assertion(cs1) ⇒
     (cs1'nwi = wr ∧ cs1'nri = rd ⇒
       ¬ cs1'reader'readerSlot = cs1'writer'writerSlot)

vc3_read: THEOREM
  ∀ (cs1: Conc_State):
    pre_read(cs1) ∧
     read_Assertion(cs1) ∧
      writerChoosesSlot_Assertion(cs1) ∧
       write_Assertion(cs1) ∧ writerIndicatesSlot_Assertion(cs1) ⇒
     (read(cs1)'1'nwi = wr ∧ read(cs1)'1'nri = rd ⇒
       ¬ (read(cs1)'1'reader'readerSlot =
            read(cs1)'1'writer'writerSlot))

vc2_firstWriterChoosesSlot: THEOREM
  ∀ (cs1: Conc_State):
    pre_firstWriterChoosesSlot(cs1) ∧
     readerChoosesSlot_Assertion(cs1) ∧
      read_Assertion(cs1) ⇒
     (cs1'nwi = wr ∧ cs1'nri = rd ⇒
       ¬ cs1'reader'readerSlot = cs1'writer'writerSlot)

vc3_firstWriterChoosesSlot: THEOREM
  ∀ (cs1: Conc_State):
    pre_firstWriterChoosesSlot(cs1) ∧
     readerChoosesSlot_Assertion(cs1) ∧
      read_Assertion(cs1) ⇒
     (firstWriterChoosesSlot(cs1)'nwi = wr ∧
       firstWriterChoosesSlot(cs1)'nri = rd ⇒
       ¬ (firstWriterChoosesSlot(cs1)'reader'readerSlot =
            firstWriterChoosesSlot(cs1)'writer'writerSlot))

vc2_writerChoosesSlot: THEOREM
  ∀ (cs1: Conc_State):
    pre_writerChoosesSlot(cs1) ∧
     writerChoosesSlot_Assertion(cs1) ∧
      readerChoosesSlot_Assertion(cs1) ∧
       read_Assertion(cs1) ⇒
     (cs1'nwi = wr ∧
       cs1'nri = rd ⇒
       ¬ cs1'reader'readerSlot = cs1'writer'writerSlot)

vc3_writerChoosesSlot: THEOREM
  ∀ (cs1: Conc_State):
    pre_writerChoosesSlot(cs1) ∧
     writerChoosesSlot_Assertion(cs1) ∧
      readerChoosesSlot_Assertion(cs1) ∧
       read_Assertion(cs1) ⇒
     (writerChoosesSlot(cs1)'nwi = wr ∧
       writerChoosesSlot(cs1)'nri = rd ⇒
       ¬ (writerChoosesSlot(cs1)'reader'readerSlot =
            writerChoosesSlot(cs1)'writer'writerSlot))

vc2_write: THEOREM
  ∀ (w: write_parameter):
    pre_write(w'p1) ∧
     write_Assertion(w'p1) ∧
      readerChoosesSlot_Assertion(w'p1) ∧
       read_Assertion(w'p1) ⇒
     (w'p1'nwi = wr ∧ w'p1'nri = rd ⇒
```

```
        ¬ w'p1'reader'readerSlot = w'p1'writer'writerSlot)

vc3_write: THEOREM
  ∀ (w: write_parameter):
    pre_write(w'p1) ∧
     writerIndicatesSlot_Assertion(w'p1) ∧
      readerChoosesSlot_Assertion(w'p1) ∧
       read_Assertion(w'p1) ⇒
     (write(w)'nwi = wr ∧ write(w)'nri = rd ⇒
       ¬ (write(w)'reader'readerSlot = write(w)'writer'writerSlot))

vc2_writerIndicatesSlot: THEOREM
  ∀ (cs1: Conc_State):
    pre_writerIndicatesSlot(cs1) ∧
     writerIndicatesSlot_Assertion(cs1) ∧ readerChoosesSlot_Assertion(cs1) ∧ read_Assertion(cs1) ⇒
     (cs1'nwi = wr ∧ cs1'nri = rd ⇒
       ¬ cs1'reader'readerSlot = cs1'writer'writerSlot)

vc3_writerIndicatesSlot: THEOREM
  ∀ (cs1: Conc_State):
    pre_writerIndicatesSlot(cs1) ∧
     writerIndicatesSlot_Assertion(cs1) ∧
      readerChoosesSlot_Assertion(cs1) ∧
       read_Assertion(cs1) ⇒
     (writerIndicatesSlot(cs1)'nwi = wr ∧ writerIndicatesSlot(cs1)'nri = rd ⇒
       ¬ (writerIndicatesSlot(cs1)'reader'readerSlot =
            writerIndicatesSlot(cs1)'writer'writerSlot))
```

In order to demonstrate that the 3-slot implementation may allow the reader and writer to access the same slot at the same time it is possible to find a number of witness values (when attempting to complete the proof obligation to show that the guarantee condition holds after executing *readerIndicatesSlot* above) so that the following proof can be completed.

```
vc3_readerIndicatesSlot_Incorrect: THEOREM
  ∃ (cs1: Conc_State):
    pre_readerIndicatesSlot(cs1) ∧
     writerChoosesSlot_Assertion(cs1) ∧
      write_Assertion(cs1) ∧
       writerIndicatesSlot_Assertion(cs1) ⇒
     readerIndicatesSlot(cs1)'nwi = wr ∧
      readerIndicatesSlot(cs1)'nri = rd ∧
       readerIndicatesSlot(cs1)'reader'readerSlot =
        readerIndicatesSlot(cs1)'writer'writerSlot

END THREE_SLOT
```

I.2 Introducing a Timing Constraint

The model in this section is almost identical to the one in the previous section. The only difference is that it models the introduction of the timing constraint from [Sim90a], and verifies that if the timing constraint can be implemented, the ACM communicates coherent data between its reader and writer. The timing constraint is that

> ... the interval between control operations in the read function is always shorter than the interval between writes ...

The constraint attempts to ensure that, if the reader chooses a new slot before the writer executes the *writerIndicatesSlot* operation, it is not possible for the writer to execute the start of the next write before the reader indicates the slot it has chosen. This will avoid the faulty operation of the ACM described in the last section, which allows the reader and writer to access the same slot at the same time.

The timing constraint is modelled by adding two auxiliary variables to the model, and amending the pre-conditions of some of the operations as follows:

1. The additional auxiliary variables are *wisSinceRcs*, which is true if the last occurrence of *writerIndicatesSlot* is after the last occurrence of *readerChoosesSlot* and false otherwise (set to true by *writerIndicatesSlot* and false by *readerChoosesSlot*), and *risSinceWis*, which is true if the last occurrence of *readerIndicatesSlot* is after the last occurrence of *writerIndicatesSlot* (set to true by *readerIndicatesSlot* and false by *writerIndicatesSlot*).

2. The pre-condition of amended to include *wisSinceRcs* $\Rightarrow$ *risSinceWis*, so that, when *writerIndicatesSlot* occurs after *readerChoosesSlot*, *writerChoosesSlot* cannot be executed unless there has been a subsequent *readerIndicatesSlot*. This avoids the incorrect operation of the ACM, and all of the proof obligations can be discharged.

The following model of this revised implementation is identical to the previous one, except for the inclusion of the additional auxiliary variables described above, and the amended pre-condition for the amended pre-condition for the *writerChoosesSlot* operation.

```
THREE_SLOT: THEORY
 BEGIN

  Val: NONEMPTY_TYPE

  v1: Val

  SlotIndex: TYPE = {s0, s1, s2}

  NextReadInstruction: TYPE = {firstRcs, rcs, ris, rd}

  NextWriteInstruction: TYPE = {firstWcs, wcs, wr, wis}

  ReaderNetworkState: TYPE = {sr, lr1, lr2, lr3, tr}

  WriterNetworkState: TYPE = {sw, lw1, lw2, lw3, tw}

  WriterState: TYPE =
  [# writerSlot: SlotIndex, currentState: WriterNetworkState #]

  ReaderState: TYPE =
  [# readerSlot: SlotIndex, currentState: ReaderNetworkState #]
```

```
Conc_State: TYPE =
[# slotWritten: SlotIndex,
   slotReading: SlotIndex,
   slots: [SlotIndex → Val],
   nri: NextReadInstruction,
   nwi: NextWriteInstruction,
   writer: WriterState,
   reader: ReaderState,
   wisSinceRcs: bool,
   risSinceWis: bool #]

pre_firstReaderChoosesSlot(p: Conc_State): bool = p'nri = firstRcs

post_firstReaderChoosesSlot(p: (pre_firstReaderChoosesSlot))(prot: Conc_State): bool =
    prot = p WITH [nri := ris,
                   reader := p'reader WITH [readerSlot := p'slotWritten, currentState := lr1],
                   wisSinceRcs := FALSE]

firstReaderChoosesSlot:
[p: (pre_firstReaderChoosesSlot) → (post_firstReaderChoosesSlot(p))]

pre_readerChoosesSlot(p: Conc_State): bool = p'nri = rcs

post_readerChoosesSlot(p: (pre_readerChoosesSlot))(prot: Conc_State): bool =
    prot = p WITH [nri := ris,
                   reader := p'reader WITH [readerSlot := p'slotWritten, currentState := lr1],
                   wisSinceRcs := FALSE]

readerChoosesSlot:
[p: (pre_readerChoosesSlot) → (post_readerChoosesSlot(p))]

pre_readerIndicatesSlot(p: Conc_State): bool = p'nri = ris

post_readerIndicatesSlot(p: (pre_readerIndicatesSlot))(prot: Conc_State): bool =
    prot = p WITH [nri := rd,
                   slotReading := p'reader'readerSlot,
                   reader := p'reader WITH [currentState := lr2],
                   risSinceWis := TRUE]

readerIndicatesSlot:
[p: (pre_readerIndicatesSlot) → (post_readerIndicatesSlot(p))]

pre_read(p: Conc_State): bool = p'nri = rd

post_read(p: (pre_read))(prot: Conc_State, v: Val): bool =
    v = p'slots(p'reader'readerSlot) ∧
     prot = p WITH [nri := rcs, reader := p'reader WITH [currentState := lr3]]

read: [p: (pre_read) → (post_read(p))]

pre_firstWriterChoosesSlot(p: Conc_State): bool = p'nwi = firstWcs

post_firstWriterChoosesSlot(p: (pre_firstWriterChoosesSlot))(prot: Conc_State): bool =
 (p'slotWritten = s0 ⇒
   (p'slotReading = s0 ⇒
     prot = p WITH [nwi := wr,
                    writer := p'writer WITH
                                [writerSlot := s1, currentState := lw1]]) ∧
    (p'slotReading = s1 ⇒
      prot = p WITH [nwi := wr,
                     writer := p'writer WITH
                                 [writerSlot := s2, currentState := lw1]]) ∧
     (p'slotReading = s2 ⇒
```

```
        prot = p WITH [nwi := wr,
                       writer := p'writer WITH
                                 [writerSlot := s1, currentState := lw1]]]) ∧
 (p'slotWritten = s1 ⇒
   (p'slotReading = s0 ⇒
     prot = p WITH [nwi := wr,
                    writer := p'writer WITH
                              [writerSlot := s2, currentState := lw1]]) ∧
    (p'slotReading = s1 ⇒
      prot = p WITH [nwi := wr,
                     writer := p'writer WITH
                               [writerSlot := s2, currentState := lw1]]) ∧
     (p'slotReading = s2 ⇒
       prot = p WITH [nwi := wr,
                      writer := p'writer WITH
                                [writerSlot := s0, currentState := lw1]])) ∧
 (p'slotWritten = s2 ⇒
   (p'slotReading = s0 ⇒
     prot = p WITH [nwi := wr,
                    writer := p'writer WITH
                              [writerSlot := s1, currentState := lw1]]) ∧
    (p'slotReading = s1 ⇒
      prot = p WITH [nwi := wr,
                     writer := p'writer WITH
                               [writerSlot := s0, currentState := lw1]]) ∧
     (p'slotReading = s2 ⇒
       prot = p WITH [nwi := wr,
                      writer := p'writer WITH
                                [writerSlot := s0, currentState := lw1]])))

firstWriterChoosesSlot:
[p: (pre_firstWriterChoosesSlot) → (post_firstWriterChoosesSlot(p))]

pre_writerChoosesSlot(p: Conc_State): bool =
    p'nwi = wcs ∧ (p'wisSinceRcs ⇒ p'risSinceWis)

post_writerChoosesSlot(p: (pre_writerChoosesSlot))(prot: Conc_State): bool =
 (p'slotWritten = s0 ⇒
   (p'slotReading = s0 ⇒
     prot = p WITH [nwi := wr,
                    writer := p'writer WITH
                              [writerSlot := s1, currentState := lw1]]) ∧
    (p'slotReading = s1 ⇒
      prot = p WITH [nwi := wr,
                     writer := p'writer WITH
                               [writerSlot := s2, currentState := lw1]]) ∧
     (p'slotReading = s2 ⇒
       prot = p WITH [nwi := wr,
                      writer := p'writer WITH
                                [writerSlot := s1, currentState := lw1]])) ∧
 (p'slotWritten = s1 ⇒
   (p'slotReading = s0 ⇒
     prot = p WITH [nwi := wr,
                    writer := p'writer WITH
                              [writerSlot := s2, currentState := lw1]]) ∧
    (p'slotReading = s1 ⇒
      prot = p WITH [nwi := wr,
                     writer := p'writer WITH
                               [writerSlot := s2, currentState := lw1]]) ∧
     (p'slotReading = s2 ⇒
       prot = p WITH [nwi := wr,
                      writer := p'writer WITH
                                [writerSlot := s0, currentState := lw1]])) ∧
 (p'slotWritten = s2 ⇒
```

```
    (p'slotReading = s0 ⇒
      prot = p WITH [nwi := wr,
                     writer := p'writer WITH
                               [writerSlot := s1, currentState := lw1]]) ∧
     (p'slotReading = s1 ⇒
       prot = p WITH [nwi := wr,
                      writer := p'writer WITH
                               [writerSlot := s0, currentState := lw1]]) ∧
      (p'slotReading = s2 ⇒
        prot = p WITH [nwi := wr,
                       writer := p'writer WITH
                                 [writerSlot := s0, currentState := lw1]]))

writerChoosesSlot:
[p: (pre_writerChoosesSlot) → (post_writerChoosesSlot(p))]

pre_write(p: Conc_State): bool = p'nwi = wr

write_parameter: TYPE = [# p1: (pre_write), v: Val #]

post_write(p: write_parameter)(prot: Conc_State): bool =
    prot = p'p1 WITH [nwi := wis,
                      (slots)(p'p1'writer'writerSlot) := p'v,
                      writer := p'p1'writer WITH [currentState := lw2]]

write: [p: write_parameter → (post_write(p))]

pre_writerIndicatesSlot(p: Conc_State): bool = p'nwi = wis

post_writerIndicatesSlot(p: (pre_writerIndicatesSlot))(prot: Conc_State): bool =
    prot = p WITH [nwi := wis,
                   slotWritten := (p'writer'writerSlot),
                   writer := p'writer WITH [currentState := lw3],
                   wisSinceRcs := TRUE,
                   risSinceWis := FALSE]

writerIndicatesSlot:
[p: (pre_writerIndicatesSlot) → (post_writerIndicatesSlot(p))]

init_writer(w: WriterState): bool = w = w WITH [currentState := sw]

init_reader(r: ReaderState): bool = r = r WITH [currentState := sr]

init_prot(p: Conc_State, init_val: Val, inv_val: Val, w: WriterState, r: ReaderState): bool =
    p = p WITH [slotWritten := s0,
                slotReading := s0,
                (slots)(s0) := init_val,
                (slots)(s1) := inv_val,
                (slots)(s2) := inv_val,
                nri := firstRcs,
                nwi := firstWcs,
                writer := w,
                reader := r,
                wisSinceRcs := FALSE,
                risSinceWis := FALSE]

firstWriterChoosesSlot_Assertion: [Conc_State → bool] =
    (λ  ·(cs: Conc_State): cs'nwi = firstWcs ⇒ ¬ cs'wisSinceRcs

writerChoosesSlot_Assertion: [Conc_State → bool] =
    (λ  ·(cs: Conc_State):
      cs'nwi = wcs ⇒ cs'slotWritten = cs'writer'writerSlot

write_Assertion: [Conc_State → bool] =
```

```
    (λ ·(cs: Conc_State):
      cs'nwi = wr ⇒
       (cs'wisSinceRcs ⇒ cs'risSinceWis) ∧
        ¬ cs'slotWritten = cs'writer'writerSlot

writerIndicatesSlot_Assertion: [Conc_State → bool] =
    (λ ·(cs: Conc_State):
      cs'nwi = wis ⇒
       (cs'wisSinceRcs ⇒ cs'risSinceWis) ∧
        ¬ cs'slotWritten = cs'writer'writerSlot

readerChoosesSlot_Assertion: [Conc_State → bool] =
    (λ ·(cs: Conc_State):
      cs'nri = rcs ⇒
       (cs'wisSinceRcs ⇒ ¬ cs'writer'writerSlot = cs'reader'readerSlot) ∧
        (¬ cs'wisSinceRcs ⇒ cs'slotWritten = cs'reader'readerSlot) ∧
         cs'slotReading = cs'reader'readerSlot

readerIndicatesSlot_Assertion: [Conc_State → bool] =
    (λ ·(cs: Conc_State):
      cs'nri = ris ⇒
       (cs'wisSinceRcs ⇒ ¬ cs'risSinceWis ∧
        ¬ cs'writer'writerSlot = cs'reader'readerSlot) ∧
        (¬ cs'wisSinceRcs ⇒ cs'slotWritten = cs'reader'readerSlot)

read_Assertion: [Conc_State → bool] =
    (λ ·(cs: Conc_State):
      cs'nri = rd ⇒
       cs'slotReading = cs'reader'readerSlot ∧
        (cs'wisSinceRcs ⇒ ¬ cs'writer'writerSlot = cs'reader'readerSlot) ∧
         (¬ cs'wisSinceRcs ⇒ cs'slotWritten = cs'reader'readerSlot)

vc_initWriter: THEOREM
  ∀ (cs: Conc_State, init: Val, inv: Val, w: WriterState, r: ReaderState):
    init_prot(cs, init, inv, w, r) ⇒ firstWriterChoosesSlot_Assertion(cs)

vc1_firstWriterChoosesSlot: THEOREM
  ∀ (cs1, cs2: Conc_State):
    pre_firstWriterChoosesSlot(cs1) ∧
     firstWriterChoosesSlot_Assertion(cs1) ∧
      readerChoosesSlot_Assertion(cs1) ∧
       readerIndicatesSlot_Assertion(cs1) ∧
        read_Assertion(cs1) ∧ cs2 = firstWriterChoosesSlot(cs1) ⇒
     write_Assertion(cs2) ∧
      readerChoosesSlot_Assertion(cs2) ∧
       readerIndicatesSlot_Assertion(cs2) ∧ read_Assertion(cs2)

vc1_writerChoosesSlot: THEOREM
  ∀ (cs1, cs2: Conc_State):
    pre_writerChoosesSlot(cs1) ∧
     writerChoosesSlot_Assertion(cs1) ∧
      readerChoosesSlot_Assertion(cs1) ∧
       readerIndicatesSlot_Assertion(cs1) ∧
        read_Assertion(cs1) ∧
         cs2 = writerChoosesSlot(cs1) ⇒
     write_Assertion(cs2) ∧
      readerChoosesSlot_Assertion(cs2) ∧
       readerIndicatesSlot_Assertion(cs2) ∧ read_Assertion(cs2)

vc1_write: THEOREM
  ∀ (w: write_parameter, cs2: Conc_State):
    pre_write(w'p1) ∧
     write_Assertion(w'p1) ∧
      readerChoosesSlot_Assertion(w'p1) ∧
```

```
        readerIndicatesSlot_Assertion(w'p1)  ∧
         read_Assertion(w'p1) ∧
          cs2 = write(w) ⇒
      writerIndicatesSlot_Assertion(cs2) ∧
       readerChoosesSlot_Assertion(cs2) ∧
        readerIndicatesSlot_Assertion(cs2) ∧ read_Assertion(cs2)

vc1_writerIndicatesSlot: THEOREM
  ∀ (cs1, cs2: Conc_State):
    pre_writerIndicatesSlot(cs1) ∧
     writerIndicatesSlot_Assertion(cs1) ∧
      readerChoosesSlot_Assertion(cs1) ∧
       readerIndicatesSlot_Assertion(cs1) ∧
        read_Assertion(cs1) ∧
         cs2 = writerIndicatesSlot(cs1) ⇒
      writerChoosesSlot_Assertion(cs2) ∧
       readerChoosesSlot_Assertion(cs2) ∧
        readerIndicatesSlot_Assertion(cs2) ∧ read_Assertion(cs2)

vc1_firstReaderChoosesSlot: THEOREM
  ∀ (cs1, cs2: Conc_State):
    pre_firstReaderChoosesSlot(cs1) ∧
     firstWriterChoosesSlot_Assertion(cs1) ∧
      writerChoosesSlot_Assertion(cs1) ∧
       write_Assertion(cs1) ∧
        writerIndicatesSlot_Assertion(cs1) ∧
         cs2 = firstReaderChoosesSlot(cs1) ⇒
      readerIndicatesSlot_Assertion(cs2) ∧
       firstWriterChoosesSlot_Assertion(cs2) ∧
        writerChoosesSlot_Assertion(cs2) ∧
         write_Assertion(cs2) ∧ writerIndicatesSlot_Assertion(cs2)

vc1_readerChoosesSlot: THEOREM
  ∀ (cs1, cs2: Conc_State):
    pre_readerChoosesSlot(cs1) ∧
     firstWriterChoosesSlot_Assertion(cs1) ∧
      writerChoosesSlot_Assertion(cs1) ∧
       write_Assertion(cs1) ∧
        writerIndicatesSlot_Assertion(cs1) ∧
         cs2 = readerChoosesSlot(cs1) ⇒
      readerIndicatesSlot_Assertion(cs2) ∧
       firstWriterChoosesSlot_Assertion(cs2) ∧
        writerChoosesSlot_Assertion(cs2) ∧
         write_Assertion(cs2) ∧ writerIndicatesSlot_Assertion(cs2)

vc1_readerIndicatesSlot: THEOREM
  ∀ (cs1, cs2: Conc_State):
    pre_readerIndicatesSlot(cs1) ∧
     readerIndicatesSlot_Assertion(cs1) ∧
      firstWriterChoosesSlot_Assertion(cs1) ∧
       writerChoosesSlot_Assertion(cs1) ∧
        write_Assertion(cs1) ∧
         writerIndicatesSlot_Assertion(cs1) ∧
          cs2 = readerIndicatesSlot(cs1) ⇒
      read_Assertion(cs2) ∧
       firstWriterChoosesSlot_Assertion(cs2) ∧
        writerChoosesSlot_Assertion(cs2) ∧
         write_Assertion(cs2) ∧ writerIndicatesSlot_Assertion(cs2)

vc1_read: THEOREM
  ∀ (cs1, cs2: Conc_State):
    pre_read(cs1) ∧
     read_Assertion(cs1) ∧
      firstWriterChoosesSlot_Assertion(cs1) ∧
```

```
        writerChoosesSlot_Assertion(cs1) ∧
         write_Assertion(cs1) ∧
          writerIndicatesSlot_Assertion(cs1) ∧
           cs2 = read(cs1)'1 ⇒
      readerChoosesSlot_Assertion(cs2) ∧
       firstWriterChoosesSlot_Assertion(cs2) ∧
        writerChoosesSlot_Assertion(cs2) ∧
         write_Assertion(cs2) ∧ writerIndicatesSlot_Assertion(cs2)

vc2_firstReaderChoosesSlot: THEOREM
  ∀ (cs1: Conc_State):
    pre_firstReaderChoosesSlot(cs1) ∧
     firstWriterChoosesSlot_Assertion(cs1) ∧
      writerChoosesSlot_Assertion(cs1) ∧
       write_Assertion(cs1) ∧ writerIndicatesSlot_Assertion(cs1) ⇒
     (cs1'nwi = wr ∧ cs1'nri = rd ⇒
       ¬ cs1'reader'readerSlot = cs1'writer'writerSlot)

vc3_firstReaderChoosesesSlot: THEOREM
  ∀ (cs1: Conc_State):
    pre_firstReaderChoosesSlot(cs1) ∧
     firstWriterChoosesSlot_Assertion(cs1) ∧
      writerChoosesSlot_Assertion(cs1) ∧
       write_Assertion(cs1) ∧ writerIndicatesSlot_Assertion(cs1) ⇒
     (firstReaderChoosesSlot(cs1)'nwi = wr ∧ firstReaderChoosesSlot(cs1)'nri = rd ⇒
       ¬ (firstReaderChoosesSlot(cs1)'reader'readerSlot =
            firstReaderChoosesSlot(cs1)'writer'writerSlot))

vc2_readerChoosesSlot: THEOREM
  ∀ (cs1: Conc_State):
    pre_readerChoosesSlot(cs1) ∧
     readerChoosesSlot_Assertion(cs1) ∧
      firstWriterChoosesSlot_Assertion(cs1) ∧
       writerChoosesSlot_Assertion(cs1) ∧
        write_Assertion(cs1) ∧ writerIndicatesSlot_Assertion(cs1) ⇒
     (cs1'nwi = wr ∧ cs1'nri = rd ⇒
       ¬ cs1'reader'readerSlot = cs1'writer'writerSlot)

vc3_readerChoosesesSlot: THEOREM
  ∀ (cs1: Conc_State):
    pre_readerChoosesSlot(cs1) ∧
     readerChoosesSlot_Assertion(cs1) ∧
      firstWriterChoosesSlot_Assertion(cs1) ∧
       writerChoosesSlot_Assertion(cs1) ∧
        write_Assertion(cs1) ∧ writerIndicatesSlot_Assertion(cs1) ⇒
     (readerChoosesSlot(cs1)'nwi = wr ∧ readerChoosesSlot(cs1)'nri = rd ⇒
       ¬ (readerChoosesSlot(cs1)'reader'readerSlot =
            readerChoosesSlot(cs1)'writer'writerSlot))

vc2_readerIndicatesSlot: THEOREM
  ∀ (cs1: Conc_State):
    pre_readerChoosesSlot(cs1) ∧
     readerChoosesSlot_Assertion(cs1) ∧
      firstWriterChoosesSlot_Assertion(cs1) ∧
       writerChoosesSlot_Assertion(cs1) ∧
        write_Assertion(cs1) ∧ writerIndicatesSlot_Assertion(cs1) ⇒
     (cs1'nwi = wr ∧ cs1'nri = rd ⇒
       ¬ cs1'reader'readerSlot = cs1'writer'writerSlot)

vc3_readerIndicatesSlot: THEOREM
  ∀ (cs1: Conc_State):
    pre_readerIndicatesSlot(cs1) ∧
     readerIndicatesSlot_Assertion(cs1) ∧
      firstWriterChoosesSlot_Assertion(cs1) ∧
```

```
        writerChoosesSlot_Assertion(cs1) ∧
         write_Assertion(cs1) ∧ writerIndicatesSlot_Assertion(cs1) ⇒
       (readerIndicatesSlot(cs1)'nwi = wr ∧ readerIndicatesSlot(cs1)'nri = rd ⇒
         ¬ (readerIndicatesSlot(cs1)'reader'readerSlot =
             readerIndicatesSlot(cs1)'writer'writerSlot))

vc2_read: THEOREM
  ∀ (cs1: Conc_State):
    pre_read(cs1) ∧
     read_Assertion(cs1) ∧
      firstWriterChoosesSlot_Assertion(cs1) ∧
       writerChoosesSlot_Assertion(cs1) ∧
        write_Assertion(cs1) ∧ writerIndicatesSlot_Assertion(cs1) ⇒
     (cs1'nwi = wr ∧ cs1'nri = rd ⇒
       ¬ cs1'reader'readerSlot = cs1'writer'writerSlot)

vc3_read: THEOREM
  ∀ (cs1: Conc_State):
    pre_read(cs1) ∧
     read_Assertion(cs1) ∧
      firstWriterChoosesSlot_Assertion(cs1) ∧
       writerChoosesSlot_Assertion(cs1) ∧
        write_Assertion(cs1) ∧ writerIndicatesSlot_Assertion(cs1) ⇒
     (read(cs1)'1'nwi = wr ∧ read(cs1)'1'nri = rd ⇒
       ¬ (read(cs1)'1'reader'readerSlot =
           read(cs1)'1'writer'writerSlot))

vc2_firstWriterChoosesSlot: THEOREM
  ∀ (cs1: Conc_State):
    pre_firstWriterChoosesSlot(cs1) ∧
     firstWriterChoosesSlot_Assertion(cs1) ∧
      readerChoosesSlot_Assertion(cs1) ∧
       readerIndicatesSlot_Assertion(cs1) ∧
        read_Assertion(cs1) ⇒
     (cs1'nwi = wr ∧ cs1'nri = rd ⇒
       ¬ cs1'reader'readerSlot = cs1'writer'writerSlot)

vc3_firstWriterChoosesSlot: THEOREM
  ∀ (cs1: Conc_State):
    pre_firstWriterChoosesSlot(cs1) ∧
     firstWriterChoosesSlot_Assertion(cs1) ∧
      readerChoosesSlot_Assertion(cs1) ∧
       readerIndicatesSlot_Assertion(cs1) ∧
        read_Assertion(cs1) ⇒
     (firstWriterChoosesSlot(cs1)'nwi = wr ∧ firstWriterChoosesSlot(cs1)'nri = rd ⇒
       ¬ (firstWriterChoosesSlot(cs1)'reader'readerSlot =
           firstWriterChoosesSlot(cs1)'writer'writerSlot))

vc2_writerChoosesSlot: THEOREM
  ∀ (cs1: Conc_State):
    pre_writerChoosesSlot(cs1) ∧
     writerChoosesSlot_Assertion(cs1) ∧
      readerChoosesSlot_Assertion(cs1) ∧
       readerIndicatesSlot_Assertion(cs1) ∧
        read_Assertion(cs1) ⇒
     (cs1'nwi = wr ∧ cs1'nri = rd ⇒
       ¬ cs1'reader'readerSlot = cs1'writer'writerSlot)

vc3_writerChoosesSlot: THEOREM
  ∀ (cs1: Conc_State):
    pre_writerChoosesSlot(cs1) ∧
     writerChoosesSlot_Assertion(cs1) ∧
      readerChoosesSlot_Assertion(cs1) ∧
       readerIndicatesSlot_Assertion(cs1) ∧
```

```
        read_Assertion(cs1) ⇒
      (writerChoosesSlot(cs1)'nwi = wr ∧ writerChoosesSlot(cs1)'nri = rd ⇒
        ¬ (writerChoosesSlot(cs1)'reader'readerSlot =
            writerChoosesSlot(cs1)'writer'writerSlot))

vc2_write: THEOREM
  ∀ (w: write_parameter):
    pre_write(w'p1) ∧
     write_Assertion(w'p1) ∧
      readerChoosesSlot_Assertion(w'p1) ∧
       readerIndicatesSlot_Assertion(w'p1) ∧ read_Assertion(w'p1) ⇒
     (w'p1'nwi = wr ∧ w'p1'nri = rd ⇒
       ¬ w'p1'reader'readerSlot = w'p1'writer'writerSlot)

vc3_write: THEOREM
  ∀ (w: write_parameter):
    pre_write(w'p1) ∧
     write_Assertion(w'p1) ∧
      readerChoosesSlot_Assertion(w'p1) ∧
       readerIndicatesSlot_Assertion(w'p1) ∧ read_Assertion(w'p1) ⇒
     (write(w)'nwi = wr ∧ write(w)'nri = rd ⇒
       ¬ write(w)'reader'readerSlot = write(w)'writer'writerSlot)

vc2_writerIndicatesSlot: THEOREM
  ∀ (cs1: Conc_State):
    pre_writerIndicatesSlot(cs1) ∧
     writerIndicatesSlot_Assertion(cs1) ∧
      readerChoosesSlot_Assertion(cs1) ∧
       readerIndicatesSlot_Assertion(cs1) ∧
        read_Assertion(cs1) ⇒
     (cs1'nwi = wr ∧ cs1'nri = rd ⇒
       ¬ cs1'reader'readerSlot = cs1'writer'writerSlot)

vc3_writerIndicatesSlot: THEOREM
  ∀ (cs1: Conc_State):
    pre_writerIndicatesSlot(cs1) ∧
     writerIndicatesSlot_Assertion(cs1) ∧
      readerChoosesSlot_Assertion(cs1) ∧
       readerIndicatesSlot_Assertion(cs1) ∧
        read_Assertion(cs1) ⇒
     (writerIndicatesSlot(cs1)'nwi = wr ∧ writerIndicatesSlot(cs1)'nri = rd ⇒
       ¬ (writerIndicatesSlot(cs1)'reader'readerSlot =
           writerIndicatesSlot(cs1)'writer'writerSlot))
END THREE_SLOT
```

I.3 A Revised 3-slot ACM Implementation

This section gives a model of the revised 3-slot implementation from [XYIS02], which is similar to the implementation from [Sim90a]. This implementation requires Hoare atomic access to the control variables, and the reader, rather than copying the name of the slot it is going to access to a local variable and then indicating the slot it has chosen to the relevant control variable, copies the new value direct to the control variable. It then uses the value of the *slotReading* control variable when it accesses the ACM during the read operation. The reader therefore only has two operations, *readerIndicatesSlot* and read. It is again possible to discharge all of the proof obligations for this

ACM. The ACM is not fully asynchronous. but the time taken to access the control variables is very short compared to the time to read and write data, and the penalty of Hoare atomic access to the control variables is considered by the authors of the paper to be a worthwhile trade off in order to obtain an otherwise very efficient implementation.

```
THREE_SLOT: THEORY
BEGIN

 Val: NONEMPTY_TYPE

 v1: Val

 SlotIndex: TYPE = {s0, s1, s2}

 NextReadInstruction: TYPE = {firstRis, ris, rd}

 NextWriteInstruction: TYPE = {firstWcs, wcs, wr, wis}

 ReaderNetworkState: TYPE = {sr, lr1, lr2, tr}

 WriterNetworkState: TYPE = {sw, lw1, lw2, lw3, tw}

 WriterState: TYPE =
 [# writerSlot: SlotIndex, currentState: WriterNetworkState #]

 ReaderState: TYPE = [# currentState: ReaderNetworkState #]

 Conc_State: TYPE =
 [# slotWritten: SlotIndex,
    slotReading: SlotIndex,
    slots: [SlotIndex → Val],
    nri: NextReadInstruction,
    nwi: NextWriteInstruction,
    writer: WriterState,
    reader: ReaderState #]
```

The *firstReaderIndicatesSlot* and *readerIndicatesSlot* operations indicate the slot the reader is going to access, by copying the value of the control variable *slotWritten* to the control variable *pairReading*.

```
 pre_firstReaderIndicatesSlot(p: Conc_State): bool = p'nri = firstRis

 post_firstReaderIndicatesSlot(p: (pre_firstReaderIndicatesSlot))(prot: Conc_State): bool =
     prot = p WITH [nri := rd,
                    slotReading := p'slotWritten,
                    reader := p'reader WITH [currentState := lr1]]

 firstReaderIndicatesSlot:
 [p: (pre_firstReaderIndicatesSlot) → (post_firstReaderIndicatesSlot(p))]

 pre_readerIndicatesSlot(p: Conc_State): bool = p'nri = ris

 post_readerIndicatesSlot(p: (pre_readerIndicatesSlot))(prot: Conc_State): bool =
     prot = p WITH [nri := rd,
                    slotReading := p'slotWritten,
                    reader := p'reader WITH [currentState := lr1]]

 readerIndicatesSlot:
 [p: (pre_readerIndicatesSlot) → (post_readerIndicatesSlot(p))]
```

The *read* operation uses the value of the control variable *pairReading* to decide which slot the reader is going to access.

```
pre_read(p: Conc_State): bool = p'nri = rd

post_read(p: (pre_read))(prot: Conc_State, v: Val): bool =
    v = p'slots(p'slotReading) ∧
     prot = p WITH [nri := ris, reader := p'reader WITH [currentState := lr2]]

read: [p: (pre_read) → (post_read(p))]
```

The writer operations are identical to the ones for the implementation from [Sim90a], given in Appendix I.1.

```
pre_firstWriterChoosesSlot(p: Conc_State): bool = p'nwi = firstWcs

post_firstWriterChoosesSlot(p: (pre_firstWriterChoosesSlot))(prot: Conc_State): bool =
 (p'slotWritten = s0 ⇒
   (p'slotReading = s0 ⇒
     prot = p WITH [nwi := wr,
                    writer := p'writer WITH
                                [writerSlot := s1, currentState := lw1]]) ∧
     (p'slotReading = s1 ⇒
       prot = p WITH [nwi := wr,
                      writer := p'writer WITH
                                  [writerSlot := s2, currentState := lw1]]) ∧
      (p'slotReading = s2 ⇒
        prot = p WITH [nwi := wr,
                       writer := p'writer WITH
                                   [writerSlot := s1, currentState := lw1]])) ∧
 (p'slotWritten = s1 ⇒
   (p'slotReading = s0 ⇒
     prot = p WITH [nwi := wr,
                    writer := p'writer WITH
                                [writerSlot := s2, currentState := lw1]]) ∧
    (p'slotReading = s1 ⇒
      prot = p WITH [nwi := wr,
                     writer := p'writer WITH
                                 [writerSlot := s2, currentState := lw1]]) ∧
     (p'slotReading = s2 ⇒
       prot = p WITH [nwi := wr,
                      writer := p'writer WITH
                                  [writerSlot := s0, currentState := lw1]])) ∧
  (p'slotWritten = s2 ⇒
    (p'slotReading = s0 ⇒
      prot = p WITH [nwi := wr,
                     writer := p'writer WITH
                                 [writerSlot := s1, currentState := lw1]]) ∧
     (p'slotReading = s1 ⇒
       prot = p WITH [nwi := wr,
                      writer := p'writer WITH
                                  [writerSlot := s0, currentState := lw1]]) ∧
      (p'slotReading = s2 ⇒
        prot = p WITH [nwi := wr,
                       writer := p'writer WITH
                                   [writerSlot := s0, currentState := lw1]]))

firstWriterChoosesSlot:
[p: (pre_firstWriterChoosesSlot) → (post_firstWriterChoosesSlot(p))]

pre_writerChoosesSlot(p: Conc_State): bool = p'nwi = wcs
```

```
post_writerChoosesSlot(p: (pre_writerChoosesSlot))(prot: Conc_State): bool =
 (p'slotWritten = s0 ⇒
   (p'slotReading = s0 ⇒
     prot = p WITH [nwi := wr,
                    writer := p'writer WITH
                                [writerSlot := s1, currentState := lw1]]) ∧
    (p'slotReading = s1 ⇒
      prot = p WITH [nwi := wr,
                     writer := p'writer WITH
                                 [writerSlot := s2, currentState := lw1]]) ∧
     (p'slotReading = s2 ⇒
       prot = p WITH [nwi := wr,
                      writer := p'writer WITH
                                  [writerSlot := s1, currentState := lw1]])) ∧
 (p'slotWritten = s1 ⇒
   (p'slotReading = s0 ⇒
     prot = p WITH [nwi := wr,
                    writer := p'writer WITH
                                [writerSlot := s2, currentState := lw1]]) ∧
    (p'slotReading = s1 ⇒
      prot = p WITH [nwi := wr,
                     writer := p'writer WITH
                                 [writerSlot := s2, currentState := lw1]]) ∧
     (p'slotReading = s2 ⇒
       prot = p WITH [nwi := wr,
                      writer := p'writer WITH
                                  [writerSlot := s0, currentState := lw1]])) ∧
 (p'slotWritten = s2 ⇒
   (p'slotReading = s0 ⇒
     prot = p WITH [nwi := wr,
                    writer := p'writer WITH
                                [writerSlot := s1, currentState := lw1]]) ∧
    (p'slotReading = s1 ⇒
      prot = p WITH [nwi := wr,
                     writer := p'writer WITH
                                 [writerSlot := s0, currentState := lw1]]) ∧
     (p'slotReading = s2 ⇒
       prot = p WITH [nwi := wr,
                      writer := p'writer WITH
                                  [writerSlot := s0, currentState := lw1]])))

writerChoosesSlot:
[p: (pre_writerChoosesSlot) → (post_writerChoosesSlot(p))]

pre_write(p: Conc_State): bool = p'nwi = wr

write_parameter: TYPE = [# p1: (pre_write), v: Val #]

post_write(p: write_parameter)(prot: Conc_State): bool =
    prot = p'p1 WITH [nwi := wis,
                      (slots)(p'p1'writer'writerSlot) := p'v,
                      writer := p'p1'writer WITH [currentState := lw2]]

write: [p: write_parameter → (post_write(p))]

pre_writerIndicatesSlot(p: Conc_State): bool = p'nwi = wis

post_writerIndicatesSlot(p: (pre_writerIndicatesSlot))(prot: Conc_State): bool =
    prot = p WITH [nwi := wis,
                   slotWritten := (p'writer'writerSlot),
                   writer := p'writer WITH [currentState := lw3]]

writerIndicatesSlot:
```

```
[p: (pre_writerIndicatesSlot) → (post_writerIndicatesSlot(p))]

init_writer(w: WriterState): bool = w = w WITH [currentState := sw]

init_reader(r: ReaderState): bool = r = r WITH [currentState := sr]

init_prot(p: Conc_State, init_val: Val, w: WriterState, r: ReaderState): bool =
    p = p WITH [slotWritten := s0,
                slotReading := s0,
                slots := ((λ ·(s0: SlotIndex): init_val),
                nri := firstRis,
                nwi := firstWcs,
                writer := w,
                reader := r]
```

In this model it is not possible to make any assertions about the reader, since it has no local variables, and the assertions for the locations in the writer assertion network are the same as for the two previous versions of the 3-slot implementation. It is possible to discharge the proof obligations to show that this version of the ACM communicates coherent data.

```
writerChoosesSlot_Assertion: [Conc_State → bool] =
    (λ ·(cs: Conc_State):
      cs'nwi = wcs ⇒ cs'slotWritten = cs'writer'writerSlot

write_Assertion: [Conc_State → bool] =
    (λ ·(cs: Conc_State):
      cs'nwi = wr ⇒
       ¬ cs'slotWritten = cs'writer'writerSlot ∧
        ¬ cs'writer'writerSlot = cs'slotReading

writerIndicatesSlot_Assertion: [Conc_State → bool] =
    (λ ·(cs: Conc_State):
      cs'nwi = wis ⇒
       ¬ cs'slotWritten = cs'writer'writerSlot ∧
        ¬ cs'writer'writerSlot = cs'slotReading

vc1_firstWriterChoosesSlot: THEOREM
  ∀ (cs1, cs2: Conc_State):
    pre_firstWriterChoosesSlot(cs1) ∧
     cs2 = firstWriterChoosesSlot(cs1) ⇒
     write_Assertion(cs2)

vc1_writerChoosesSlot: THEOREM
  ∀ (cs1, cs2: Conc_State):
    pre_writerChoosesSlot(cs1) ∧
     writerChoosesSlot_Assertion(cs1) ∧
      cs2 = writerChoosesSlot(cs1)
     ⇒ write_Assertion(cs2)

vc1_write: THEOREM
  ∀ (w: write_parameter, cs2: Conc_State):
    pre_write(w'p1) ∧
     write_Assertion(w'p1) ∧
      cs2 = write(w) ⇒
     writerIndicatesSlot_Assertion(cs2)

vc1_writerIndicatesSlot: THEOREM
  ∀ (cs1, cs2: Conc_State):
    pre_writerIndicatesSlot(cs1) ∧
     writerIndicatesSlot_Assertion(cs1) ∧
```

```
      cs2 = writerIndicatesSlot(cs1)
    ⇒ writerChoosesSlot_Assertion(cs2)

vc1_firstReaderIndicatesSlot: THEOREM
  ∀ (cs1, cs2: Conc_State):
    pre_firstReaderIndicatesSlot(cs1) ∧
     writerChoosesSlot_Assertion(cs1) ∧
      write_Assertion(cs1) ∧
       writerIndicatesSlot_Assertion(cs1) ∧
        cs2 = firstReaderIndicatesSlot(cs1) ⇒
     writerChoosesSlot_Assertion(cs2) ∧
      write_Assertion(cs2) ∧ writerIndicatesSlot_Assertion(cs2)

vc1_readerIndicatesSlot: THEOREM
  ∀ (cs1, cs2: Conc_State):
    pre_readerIndicatesSlot(cs1) ∧
     writerChoosesSlot_Assertion(cs1) ∧
      write_Assertion(cs1) ∧
       writerIndicatesSlot_Assertion(cs1) ∧
        cs2 = readerIndicatesSlot(cs1) ⇒
     writerChoosesSlot_Assertion(cs2) ∧
      write_Assertion(cs2) ∧ writerIndicatesSlot_Assertion(cs2)

vc1_read: THEOREM
  ∀ (cs1, cs2: Conc_State):
    pre_read(cs1) ∧
     writerChoosesSlot_Assertion(cs1) ∧
      write_Assertion(cs1) ∧
       writerIndicatesSlot_Assertion(cs1) ∧
        cs2 = read(cs1)`1 ⇒
     writerChoosesSlot_Assertion(cs2) ∧
      write_Assertion(cs2) ∧ writerIndicatesSlot_Assertion(cs2)

vc2_firstReaderIndicatesSlot: THEOREM
  ∀ (cs1: Conc_State):
    pre_firstReaderIndicatesSlot(cs1) ∧
     writerChoosesSlot_Assertion(cs1) ∧
      write_Assertion(cs1) ∧
       writerIndicatesSlot_Assertion(cs1) ⇒
     (cs1`nwi = wr ∧ cs1`nri = rd ⇒
       ¬ cs1`slotReading = cs1`writer`writerSlot)

vc3_firstReaderIndicatesSlot: THEOREM
  ∀ (cs1: Conc_State):
    pre_firstReaderIndicatesSlot(cs1) ∧
     writerChoosesSlot_Assertion(cs1) ∧
      write_Assertion(cs1) ∧
       writerIndicatesSlot_Assertion(cs1) ⇒
     (firstReaderIndicatesSlot(cs1)`nwi = wr ∧ firstReaderIndicatesSlot(cs1)`nri = rd ⇒
       ¬ (firstReaderIndicatesSlot(cs1)`slotReading =
            firstReaderIndicatesSlot(cs1)`writer`writerSlot))

vc2_readerIndicatesSlot: THEOREM
  ∀ (cs1: Conc_State):
    pre_readerIndicatesSlot(cs1) ∧
     writerChoosesSlot_Assertion(cs1) ∧
      write_Assertion(cs1) ∧
       writerIndicatesSlot_Assertion(cs1) ⇒
     (cs1`nwi = wr ∧ cs1`nri = rd ⇒
       ¬ cs1`slotReading = cs1`writer`writerSlot)

vc3_readerIndicatesSlot: THEOREM
  ∀ (cs1: Conc_State):
    pre_readerIndicatesSlot(cs1) ∧
```

```
      writerChoosesSlot_Assertion(cs1) ∧
       write_Assertion(cs1) ∧
        writerIndicatesSlot_Assertion(cs1) ⇒
      (readerIndicatesSlot(cs1)'nwi = wr ∧ readerIndicatesSlot(cs1)'nri = rd ⇒
        ¬ (readerIndicatesSlot(cs1)'slotReading =
            readerIndicatesSlot(cs1)'writer'writerSlot))

vc2_read: THEOREM
  ∀ (cs1: Conc_State):
    pre_read(cs1) ∧
     writerChoosesSlot_Assertion(cs1) ∧
      write_Assertion(cs1) ∧
       writerIndicatesSlot_Assertion(cs1) ⇒
     (cs1'nwi = wr ∧ cs1'nri = rd ⇒
       ¬ cs1'slotReading = cs1'writer'writerSlot)

vc3_read: THEOREM
  ∀ (cs1: Conc_State):
    pre_read(cs1) ∧
     writerChoosesSlot_Assertion(cs1) ∧
      write_Assertion(cs1) ∧
       writerIndicatesSlot_Assertion(cs1) ⇒
     (read(cs1)'1'nwi = wr ∧ read(cs1)'1'nri = rd ⇒
       ¬ (read(cs1)'1'slotReading = read(cs1)'1'writer'writerSlot))

vc2_firstWriterChoosesSlot: THEOREM
  ∀ (cs1: Conc_State):
    pre_firstWriterChoosesSlot(cs1) ⇒
     (cs1'nwi = wr ∧ cs1'nri = rd ⇒
       ¬ cs1'slotReading = cs1'writer'writerSlot)

vc3_firstWriterChoosesSlot: THEOREM
  ∀ (cs1: Conc_State):
    pre_firstWriterChoosesSlot(cs1) ⇒
     (firstWriterChoosesSlot(cs1)'nwi = wr ∧ firstWriterChoosesSlot(cs1)'nri = rd ⇒
       ¬ (firstWriterChoosesSlot(cs1)'slotReading =
           firstWriterChoosesSlot(cs1)'writer'writerSlot))

vc2_writerChoosesSlot: THEOREM
  ∀ (cs1: Conc_State):
    pre_writerChoosesSlot(cs1) ∧ writerChoosesSlot_Assertion(cs1) ⇒
     (cs1'nwi = wr ∧ cs1'nri = rd ⇒
       ¬ cs1'slotReading = cs1'writer'writerSlot)

vc3_writerChoosesSlot: THEOREM
  ∀ (cs1: Conc_State):
    pre_writerChoosesSlot(cs1) ∧ writerChoosesSlot_Assertion(cs1) ⇒
     (writerChoosesSlot(cs1)'nwi = wr ∧ writerChoosesSlot(cs1)'nri = rd ⇒
       ¬ (writerChoosesSlot(cs1)'slotReading =
           writerChoosesSlot(cs1)'writer'writerSlot))

vc2_write: THEOREM
  ∀ (w: write_parameter):
    pre_write(w'p1) ∧ write_Assertion(w'p1) ⇒
     (w'p1'nwi = wr ∧ w'p1'nri = rd ⇒
       ¬ w'p1'slotReading = w'p1'writer'writerSlot)

vc3_write: THEOREM
  ∀ (w: write_parameter):
    pre_write(w'p1) ∧ writerIndicatesSlot_Assertion(w'p1) ⇒
     (write(w)'nwi = wr ∧ write(w)'nri = rd ⇒
       ¬ (write(w)'slotReading = write(w)'writer'writerSlot))

vc2_writerIndicatesSlot: THEOREM
```

```
    ∀ (cs1: Conc_State):
      pre_writerIndicatesSlot(cs1) ∧ writerIndicatesSlot_Assertion(cs1) ⇒
       (cs1'nwi = wr ∧ cs1'nri = rd ⇒
         ¬ cs1'slotReading = cs1'writer'writerSlot)

  vc3_writerIndicatesSlot: THEOREM
    ∀ (cs1: Conc_State):
      pre_writerIndicatesSlot(cs1) ∧ writerIndicatesSlot_Assertion(cs1) ⇒
       (writerIndicatesSlot(cs1)'nwi = wr ∧ writerIndicatesSlot(cs1)'nri = rd ⇒
         ¬ (writerIndicatesSlot(cs1)'slotReading =
              writerIndicatesSlot(cs1)'writer'writerSlot))

END THREE_SLOT
```

Appendix J

Modelling Metastability Using CSP

This appendix gives the complete model of the 4-slot in machine readable CSP (CSP_M), that has been used with the FDR model checker to explore the behaviour of the ACM in the presence of metastability (using a number of different models of bit control variables that model the effects of metastability in different ways, and also model the different methods for containing the effects of metastability as described in Chapter 7). The results of model checking these models are described in Section 7.5.2, and it has been shown that the 4-slot implementation is L-atomic provided the effects of metastability can be contained. The model is as follows:

Data types to:

1. Define the maximum number of values that can be communicated by the ACM in the model (so that the model can be represented by a finite state machine).

2. Represent the values that the bits, and pair and slot indices, can take.

```
max_no_of_values = 10
data_values = {1..max_no_of_values}
datatype bit_values = b0 | b1 | d
datatype slot_index = s1 | s2 | s12
datatype pair_index = p1 | p2 | p12
```

Processes to convert between slot/pair indices and the values of the bit control variables.

```
bs(b0) = s1  -- convert bit values to slot indexes
bs(b1) = s2
bs(d) = s12
bp(b0) = p1  -- convert bit values to pair indexes
bp(b1) = p2
bp(d) = p12
sb(s1) = b0  -- convert slot indexes to bit values
```

```
sb(s2) = b1
sb(s12) = d
pb(p1) = b0  -- convert pair indexes to bit values
pb(p2) = b1
pb(p12) = d
toggle(b0) = b1  -- toggle (invert) bit values
toggle(b1) = b0
toggle(d) = d
```

Declarations for the CSP channels that are required in the models.

```
datatype atomic_operations = atomic_rd | atomic_wr
channel pool :  atomic_operations.data_values

datatype slot_operations =
      sr_slot | er_slot.data_values | sw_slot.data_values | ew_slot

channel slots : pair_index.slot_index.slot_operations

channel slot_written_pair, read_slot_pair : pair_index
channel slot_written_slot, read_slot_slot : slot_index
channel slot_written_val, read_slot_val : data_values

datatype shared_bit_operations = sr | er.bit_values | sw.bit_values | ew

datatype local_bit_operations = set.bit_values | get.bit_values

channel reading, latest : shared_bit_operations

channel writers_slots : pair_index.shared_bit_operations

channel start_write, end_read : data_values

channel clash_bang, mono_bang,  dither, start_read, end_write

channel start_write_slots, end_write_slots, start_read_slots, end_read_slots

channel LB_write_pair, LB_write_slot,
       LB_read_pair, LB_read_slot : local_bit_operations
```

– Incoherence Specification –

An ACM that refines this specification does not transmit coherent data between its reader and writer. When the reader and writer access the same slot at the same time a single clash_bang is output and the process stops.

```
Incoherence_Spec = clash_bang -> STOP
```

– Monotonic Activities –

A process that transmits a monotonically increasing integer value between its reader and writer. An ACM that refines this specification maintains (a possibly partial) ordering of data between its reader and writer - the reader reads the items in the order they were written, but may not read all of the items.

```
Write_Act(n) = start_write!n -> if n == max_no_of_values then STOP
                                else end_write -> Write_Act(n+1)

Read_Act(old_x) = start_read -> end_read?x ->
                  if x < old_x then mono_bang -> STOP else Read_Act(x)
```

– Hoare Atomic variable Definition —

An ACM that only allows Hoare atomic (complete) writes and reads.

```
H_Atomic_Var(var_name, val) =
     var_name.atomic_wr?x -> H_Atomic_Var(var_name, x) □
     var_name.atomic_rd!val -> H_Atomic_Var(var_name, val)
```

– Atomic Shared Variable "Pool" Specification –

A definition of a L-atomic ACM. An ACM that refines this specification is L-atomic.

```
Read = start_read -> pool.atomic_rd?val -> end_read!val -> Read

Write = start_write?val -> pool.atomic_wr!val -> end_write -> Write

Pool_State = H_Atomic_Var(pool, 1)

Pool_Spec = (((Read ||| Write) [| {| pool |} |] Pool_State) \{| pool |})
```

– Semi-Regular-ACM Specification –

The definition of a semi-regular ACM - one where the reader can only read values that have been previously written. The process creates a set of all of the values that have already been written (and the initial value) and ensures that the reader only reads values from that set.

```
SemiRegACM(vals) =
   start_write?x -> SemiRegACM_w(union({x}, vals)) □
   start_read -> SemiRegACM_r(vals)

SemiRegACM_w(vals) =
   end_write -> SemiRegACM(vals) □
   start_read -> SemiRegACM_wr(vals)

SemiRegACM_r(vals) =
   start_write?x -> SemiRegACM_wr(union({x}, vals)) □
   ([] z : vals @ end_read!z -> SemiRegACM(vals))

SemiRegACM_wr(vals) =
   end_write -> SemiRegACM_r(vals) □
   ([] z : vals @ end_read!z -> SemiRegACM_w(vals))


SemiRegACM_Spec = SemiRegACM({1})
```

– Regular-ACM Specification –

Specification of a regular ACM - one where the reader can only read the item written immediately before the read started or one of the values that is written by a write that occurs concurrently with the read. It creates a set of values that are written while the read is in progress plus the value written immediately before the read starts, and ensures that the reader can only read one of these values.

```
RegACM(val) =
   start_write?x -> RegACM_w(union({x}, {val}), x) []
   start_read -> RegACM_r(val)

RegACM_w(vals, x) =
   end_write -> RegACM(x) []
   start_read -> RegACM_wr(vals, x)

RegACM_r(val) =
   start_write?x -> RegACM_wr(union({x}, {val}), x) []
   end_read!val -> RegACM(val)

RegACM_wr(vals, x) =
   end_write -> RegACM_r_clashed(vals, x) []
   ([] z : vals @ end_read!z -> RegACM_w(vals, x))

RegACM_r_clashed(vals, x) =
   start_write?z -> RegACM_wr(union({z}, vals), z) []
   ([] z : vals @ end_read!z -> RegACM(x))


RegACM_Spec = RegACM(1)
```

– Non-Atomic Slots with deadlock/bang behaviour if multiply accessed –

The following is a non-atomic variable that deadlocks (after performing a detectable clash_bang operation) should it not be accessed atomically: it is used for modelling the slots in the four slot ACM. If presented with a dithering value "d", it non-deterministically resolves it.

```
Slot(pair_name, slot_name, val) =
      slots.pair_name.slot_name.sw_slot?x ->
         (slots.pair_name.slot_name.ew_slot -> Slot(pair_name, slot_name, x)
          []
          slots.pair_name.slot_name.sr_slot -> clash_bang -> STOP)
      []
      slots.pair_name.slot_name.sr_slot ->
         (slots.pair_name.slot_name.sw_slot?x -> clash_bang -> STOP
          []
          slots.pair_name.slot_name.er_slot!val ->
                              Slot(pair_name, slot_name, val))

the_actual_slots = (Slot(p1, s1, 1) ||| Slot(p1, s2, 1) |||
                    Slot(p2, s1, 1) ||| Slot(p2, s2, 1))



write_slots =
   start_write_slots ->
   slot_written_pair?pair ->
   slot_written_slot?slot ->
   slot_written_val?val ->
   (if pair == p12 then
      (if slot == s12 then
         (slot_written_proc(p1, s1, val) |~|
          slot_written_proc(p1, s2, val) |~|
          slot_written_proc(p2, s1, val) |~|
          slot_written_proc(p2, s2, val))
      else
         (slot_written_proc(p1, slot, val) |~|
          slot_written_proc(p2, slot, val)))
```

```
    else
       (if slot == s12 then
          (slot_written_proc(pair, s1, val) |~|
           slot_written_proc(pair, s2, val))
       else
          slot_written_proc(pair, slot, val)))


slot_written_proc(pair, slot, val) =
    slots.pair.slot!sw_slot!val ->
    slots.pair.slot.ew_slot ->
    end_write_slots ->
    write_slots


read_slots =
   start_read_slots ->
   read_slot_pair?pair ->
   read_slot_slot?slot ->
   (if pair == p12 then
      (if slot == s12 then
         (read_slot_proc(p1, s1) |~|
          read_slot_proc(p1, s2) |~|
          read_slot_proc(p2, s1) |~|
          read_slot_proc(p2, s2))
      else
         (read_slot_proc(p1, slot) |~|
          read_slot_proc(p2, slot)))
   else
      (if slot == s12 then
         (read_slot_proc(pair, s1) |~|
          read_slot_proc(pair, s2))
      else
         read_slot_proc(pair, slot)))


read_slot_proc(pair, slot) =
    slots.pair.slot.sr_slot ->
    slots.pair.slot.er_slot?val ->
    read_slot_val!val ->
    end_read_slots -> read_slots


the_slots = (read_slots ||| write_slots) [| {| slots |} |] the_actual_slots
                \ {| slots |}
```

– A (highly metastable) local bit variable 1 –

This model of a local bit allows multiple accesses by a reader while its value is metastable (potentially infinite metastability). If the reader accesses it while the value is metastable (d) it non-deterministically returns one of the two possible valid values (0 or 1) or the metastable value, d.

```
LB1(var_name, val) = if val == d then
             (LB1(var_name, b0) |~| LB1(var_name, b1) |~|
             (var_name.set?x -> LB1(var_name, x) □
              var_name.get!val -> LB1(var_name, val)))
          else
             (var_name.set?x -> LB1(var_name, x) □
              var_name.get!val -> LB1(var_name, val))
```

```
the_writers_local_bits1 =
      LB1(LB_write_pair, b0) ||| LB1(LB_write_slot, b0)

the_readers_local_bits1 =
     LB1(LB_read_pair, b0) ||| LB1(LB_read_slot, b0)
```

– A (limited metastable) local bit variable 2 –

This model of a local bit non-deterministically returns one of the valid values(0 or 1), if the reader accesses it while it is metastable. This models the engineering solution that can be employed to contain metastability - that it is possible to make the reader wait for a short time before using the value read. This allows a metastable value to resolve to a valid one with very high probability.

```
LB2(var_name, val) = var_name.set?x ->
          (if x == d then
              (LB2(var_name, b0) |~| LB2(var_name, b1))
           else
              LB2(var_name, x))
          [] var_name.get!val -> LB2(var_name, val)


the_writers_local_bits2 =
      LB2(LB_write_pair, b0) ||| LB2(LB_write_slot, b0)

the_readers_local_bits2 =
     LB2(LB_read_pair, b0) ||| LB2(LB_read_slot, b0)
```

The various different models of bits that have been used to investigate properties of the 4-slot implementation.

— BIT VARIABLES: BIT0 —

This model is included for completeness and models Hoare atomic access to the variable.

```
BIT0(var_name, val) =
    var_name.sw?x -> var_name.ew -> BIT0(var_name, x) □
    var_name.sr ->  var_name.er!val -> BIT0(var_name, val)


BITs0 =
   (||| x : {reading, latest, writers_slots.p1,
                                  writers_slots.p2} @ BIT0(x, b0))
```

— BIT VARIABLES: BIT1 —

A model of a type-safe bit. It allows arbitrary clashes between the reader and writer, and non-deterministically returns a 0 or a 1 to the reader when a clash occurs. Metastability is ignored.

```
BIT1(var_name, val) = var_name.sw?x -> BIT1_w(var_name, val, x) □
                      var_name.sr -> BIT1_r(var_name, val)

BIT1_w(var_name, val, x) = var_name.ew -> BIT1(var_name, x) □
                           var_name.sr -> BIT1_wr(var_name, val, x)

BIT1_r(var_name, val) = var_name.sw?x -> BIT1_wr(var_name, val, x) □
                        var_name.er!val -> BIT1(var_name, val)

BIT1_wr(var_name, val, x) = var_name.ew -> BIT1_r_clashed(var_name, x) □
                  (var_name.er!b0 -> BIT1_w(var_name, val, x) |~|
                   var_name.er!b1 -> BIT1_w(var_name, val, x))

BIT1_r_clashed(var_name, val) = var_name.sw?x -> BIT1_wr(var_name, val, x) □
                        (var_name.er!b0 -> BIT1(var_name, val) |~|
                         var_name.er!b1 -> BIT1(var_name, val))

BITs1 =
   (||| x : {reading, latest, writers_slots.p1,
                                    writers_slots.p2} @ BIT1(x, b0))
```

— BIT VARIABLES: BIT2 —

This model is the same as BIT1 except that the bit remains stable when it is overwritten with the same value. This means that it deterministically returns the value that it contains, provided it is being overwritten with the same value, when the reader and writer access the variable concurrently.

```
BIT2(var_name, val) =
     var_name.sw?x -> (if x == val then
                 BIT2_w_stable(var_name, val)
             else
                 BIT2_w(var_name, val, x))
     □ var_name.sr -> BIT2_r(var_name, val)

BIT2_w(var_name, val, x) = var_name.ew -> BIT2(var_name, x) □
                           var_name.sr -> BIT2_wr(var_name, val, x)

BIT2_r(var_name, val) =
     var_name.sw?x -> (if x == val then
                 BIT2_wr_stable(var_name, val)
             else
                 BIT2_wr(var_name, val, x))
     □ var_name.er!val -> BIT2(var_name, val)

BIT2_wr(var_name, val, x) = var_name.ew -> BIT2_r_clashed(var_name, x) □
                  (var_name.er!b0 -> BIT2_w(var_name, val, x) |~|
                   var_name.er!b1 -> BIT2_w(var_name, val, x))

BIT2_r_clashed(var_name, val) = var_name.sw?x -> BIT2_wr(var_name, val, x) □
                      (var_name.er!b0 -> BIT2(var_name, val) |~|
                       var_name.er!b1 -> BIT2(var_name, val))

BIT2_w_stable(var_name, val) = var_name.ew -> BIT2(var_name, val) □
                               var_name.sr -> BIT2_wr_stable(var_name, val)

BIT2_wr_stable(var_name, val) = var_name.ew -> BIT2_r(var_name, val) □
                                var_name.er!val -> BIT2_w_stable(var_name, val)

BITs2 =
   (||| x : {reading, latest, writers_slots.p1,
```

```
writers_slots.p2} @ BIT2(x, b0))
```

— BIT VARIABLES: BIT3 —

As BIT2 except metastability causes arbitrary *clock stretching*. This is the method for containing metastability proposed in [Cha87] where the clock of the reader can be arbitrarily stopped when it detects it is reading a metastable value, to allow the value to resolve to a stable one. The clock is then restarted and may be out of phase when compared to the period before it was stopped.

```
BIT3(var_name, val) =
     var_name.sw?x -> (if x == val then
                BIT3_w_stable(var_name, val)
            else
                BIT3_w(var_name, val, x))
     [] var_name.sr -> BIT3_r(var_name, val)

BIT3_w(var_name, val, x) = var_name.ew -> BIT3(var_name, x) []
                           var_name.sr -> BIT3_wr(var_name, val, x)

BIT3_r(var_name, val) =
     var_name.sw?x -> (if x == val then
                BIT3_wr_stable(var_name, val)
            else
                BIT3_wr(var_name, val, x))
     [] var_name.er!val -> BIT3(var_name, val)

BIT3_wr(var_name, val, x) = var_name.ew -> BIT3_r_clashed(var_name, x) []
                  (var_name.er!b0 -> BIT3_w(var_name, val, x) |~|
                   var_name.er!b1 -> BIT3_w(var_name, val, x) |~|
                   dither -> BIT3_wr(var_name, val, x))

BIT3_r_clashed(var_name, val) = var_name.sw?x -> BIT3_wr(var_name, val, x) []
                     (var_name.er!b0 -> BIT3(var_name, val) |~|
                      var_name.er!b1 -> BIT3(var_name, val) |~|
                      dither -> BIT3_r_clashed(var_name, val))

BIT3_w_stable(var_name, val) = var_name.ew -> BIT3(var_name, val) []
                               var_name.sr -> BIT3_wr_stable(var_name, val)

BIT3_wr_stable(var_name, val) = var_name.ew -> BIT3_r(var_name, val) []
                                var_name.er!val -> BIT3_w_stable(var_name, val)

BITs3 =
   (||| x : {reading, latest, writers_slots.p1,
               writers_slots.p2} @ (BIT3(x, b0) \ {| dither |} ))
```

The remaining bit models use the local bits (LB1 and LB2) to store the values that are read and then re-read the values from the local bits before using them to access the ACM. This is a more realistic model of the behaviour of the ACM implementation

— BIT VARIABLES: BIT4 —

The BIT4 model is the first to explicitly include the possibility that the reader of the bit may return a metastable value (d). It may use either of the

local bit models (LB1, which allows the reader to return the metastable value multiple times, and LB2 which non-deterministically resolves metastable values as they are read). It allows the reader to clash multiple times with a single write. The value contained in the variable remains stable when overwritten with the same value.

```
BIT4(var_name, val) =
     var_name.sw?x -> (if x == val then
                  BIT4_w_stable(var_name, val)
               else
                  BIT4_w(var_name, val, x))
     [] var_name.sr -> BIT4_r(var_name, val)

BIT4_w(var_name, val, x) = var_name.ew -> BIT4(var_name, x) []
                           var_name.sr -> BIT4_wr(var_name, val, x)

BIT4_r(var_name, val) =
     var_name.sw?x -> (if x == val then
                  BIT4_wr_stable(var_name, val)
               else
                  BIT4_wr(var_name, val, x))
     [] var_name.er!val -> BIT4(var_name, val)

BIT4_wr(var_name, val, x) = var_name.ew -> BIT4_r_clashed(var_name, x) []
                  (var_name.er!b0 -> BIT4_w(var_name, val, x) |~|
                   var_name.er!b1 -> BIT4_w(var_name, val, x) |~|
                   var_name.er!d -> BIT4_w(var_name, val, x))

BIT4_r_clashed(var_name, val) = var_name.sw?x -> BIT4_wr(var_name, val, x) []
                      (var_name.er!b0 -> BIT4(var_name, val) |~|
                       var_name.er!b1 -> BIT4(var_name, val) |~|
                       var_name.er!d -> BIT4(var_name, val))

BIT4_w_stable(var_name, val) = var_name.ew -> BIT4(var_name, val) []
                               var_name.sr -> BIT4_wr_stable(var_name, val)

BIT4_wr_stable(var_name, val) = var_name.ew -> BIT4_r(var_name, val) []
                                var_name.er!val -> BIT4_w_stable(var_name, val)

BITs4 =
   (||| x : {reading, latest, writers_slots.p1,
                                   writers_slots.p2} @ BIT4(x, b0))
```

— BIT VARIABLES: BIT5 —

As BIT4, except that it disallows multiple clashes with a single write.

```
BIT5(var_name, val) =
   var_name.sw?x -> (if x == val then
               BIT5_w_stable(var_name, val)
            else
                BIT5_w(var_name, val, x))
   [] var_name.sr -> BIT5_r(var_name, val)

BIT5_w(var_name, val, x) = var_name.ew -> BIT5(var_name, x) []
                           var_name.sr -> BIT5_wr(var_name, val, x)

BIT5_r(var_name, val) =
   var_name.sw?x -> (if x == val then
                  BIT5_wr_stable(var_name, val)
            else
```

```
                BIT5_wr(var_name, val, x))
   [] var_name.er!val -> BIT5(var_name, val)

BIT5_wr(var_name, val, x) = var_name.ew -> BIT5_r_clashed(var_name, x) []
   (var_name.er!b0 -> BIT5_w_r_occured(var_name, val, x) |~|
    var_name.er!b1 -> BIT5_w_r_occured(var_name, val, x) |~|
    var_name.er!d -> BIT5_w_r_occured(var_name, val, x))

BIT5_r_clashed(var_name, val) =
   var_name.er!b0 -> BIT5(var_name, val) |~|
   var_name.er!b1 -> BIT5(var_name, val) |~|
   var_name.er!d -> BIT5(var_name, val)

BIT5_w_stable(var_name, val) = var_name.ew -> BIT5(var_name, val) []
                               var_name.sr -> BIT5_wr_stable(var_name, val)

BIT5_wr_stable(var_name, val) = var_name.ew -> BIT5_r(var_name, val) []
                                var_name.er!val -> BIT5_w_stable(var_name, val)

BIT5_w_r_occured(var_name, val, x) = var_name.ew -> BIT5(var_name, x)


BITs5 =
   (||| x : {reading, latest, writers_slots.p1,
                                   writers_slots.p2} @ BIT5(x, b0))
```

— BIT VARIABLES: BIT6 —

As BIT5 except that the value contained in the bit flickers when overwritten with the same value.

```
BIT6(var_name, val) =
   var_name.sw?x -> BIT6_w(var_name, val, x)
   [] var_name.sr -> BIT6_r(var_name, val)

BIT6_w(var_name, val, x) = var_name.ew -> BIT6(var_name, x) []
                           var_name.sr -> BIT6_wr(var_name, val, x)

BIT6_r(var_name, val) =
   var_name.sw?x -> BIT6_wr(var_name, val, x)
   [] var_name.er!val -> BIT6(var_name, val)

BIT6_wr(var_name, val, x) = var_name.ew -> BIT6_r_clashed(var_name, x) []
   (var_name.er!b0 -> BIT6_w_r_occured(var_name, val, x) |~|
    var_name.er!b1 -> BIT6_w_r_occured(var_name, val, x) |~|
    var_name.er!d -> BIT6_w_r_occured(var_name, val, x))

BIT6_r_clashed(var_name, val) =
   var_name.er!b0 -> BIT6(var_name, val) |~|
   var_name.er!b1 -> BIT6(var_name, val) |~|
   var_name.er!d -> BIT6(var_name, val)


BIT6_w_r_occured(var_name, val, x) = var_name.ew -> BIT6(var_name, x)


BITs6 =
   (||| x : {reading, latest, writers_slots.p1,
                                   writers_slots.p2} @ BIT6(x, b0))
```

———— Four-Slot Writer and Reader Algorithms ——

In each case there are two versions of the algorithms, one that does not use the local bit variables and the other that does. The second (local bit) version of the algorithm has two variants, one to use the LB1 model and the other for the LB2 model.

```
Fourslot_Writer =
    start_write?val ->
    reading.sr ->
    reading.er?not_pair_written ->
    writers_slots.bp(toggle(not_pair_written)).sr ->
    writers_slots.bp(toggle(not_pair_written)).er?not_slot_written ->
    start_write_slots ->
    slot_written_pair!bp(toggle(not_pair_written)) ->
    slot_written_slot!bs(toggle(not_slot_written)) ->
    slot_written_val!val ->
    end_write_slots ->
    writers_slots.bp(toggle(not_pair_written)).sw!toggle(not_slot_written) ->
    writers_slots.bp(toggle(not_pair_written)).ew ->
    latest.sw!toggle(not_pair_written) ->
    latest.ew ->
    end_write ->
    Fourslot_Writer

Fourslot_Writer_LB =
    start_write?val ->
    reading.sr ->
    reading.er?not_pair_written ->
    LB_write_pair.set!toggle(not_pair_written) ->
    LB_write_pair.get?pair_written ->
    writers_slots.bp(pair_written).sr ->
    writers_slots.bp(pair_written).er?not_slot_written ->
    LB_write_slot.set!toggle(not_slot_written) ->
    LB_write_slot.get?slot_written ->
    LB_write_pair.get?pair_written ->
    start_write_slots ->
    slot_written_pair!bp(pair_written) ->
    slot_written_slot!bs(slot_written) ->
    slot_written_val!val ->
    end_write_slots ->
    LB_write_pair.get?pair_written ->
    LB_write_slot.get?slot_written ->
    writers_slots.bp(pair_written).sw!slot_written ->
    writers_slots.bp(pair_written).ew ->
    LB_write_pair.get?pair_written ->
    latest.sw!pair_written ->
    latest.ew ->
    end_write ->
    Fourslot_Writer_LB

Writer_LB1 =
      Fourslot_Writer_LB [| {| LB_write_pair, LB_write_slot |} |]
            the_writers_local_bits1 \ {| LB_write_pair, LB_write_slot |}

Writer_LB2 =
      Fourslot_Writer_LB [| {| LB_write_pair, LB_write_slot |} |]
            the_writers_local_bits2 \ {| LB_write_pair, LB_write_slot |}
```

```
Fourslot_Reader =
    start_read ->
    latest.sr ->
    latest.er?read_pair ->
    reading.sw!read_pair ->
    reading.ew ->
    writers_slots.bp(read_pair).sr ->
    writers_slots.bp(read_pair).er?read_slot ->
    start_read_slots ->
    read_slot_pair!bp(read_pair) ->
    read_slot_slot!bs(read_slot) ->
    read_slot_val?val ->
    end_read_slots ->
    end_read!val ->
    Fourslot_Reader


Fourslot_Reader_LB =
    start_read ->
    latest.sr ->
    latest.er?read_pair ->
    LB_read_pair.set!read_pair ->
    LB_read_pair.get?read_pair ->
    reading.sw!read_pair ->
    reading.ew ->
    LB_read_pair.get?read_pair ->
    writers_slots.bp(read_pair).sr ->
    writers_slots.bp(read_pair).er?read_slot ->
    LB_read_slot.set!read_slot ->
    LB_read_slot.get?read_slot ->
    LB_read_pair.get?read_pair ->
    start_read_slots ->
    read_slot_pair!bp(read_pair) ->
    read_slot_slot!bs(read_slot) ->
    read_slot_val?val ->
    end_read_slots ->
    end_read!val ->
    Fourslot_Reader_LB


Reader_LB1 = Fourslot_Reader_LB [| {| LB_read_pair, LB_read_slot |} |]
                the_readers_local_bits1 \ {| LB_read_pair, LB_read_slot |}


Reader_LB2 = Fourslot_Reader_LB [| {| LB_read_pair, LB_read_slot |} |]
                the_readers_local_bits2 \ {| LB_read_pair, LB_read_slot |}
```

Four_Slot Definitions

The definitions of the models of the 4-slot algorithms with the different versions of the models of bits and local variables.

```
Four_Slot_BIT0 =(((Fourslot_Writer ||| Fourslot_Reader)
                    [| {| read_slot_pair, read_slot_slot, read_slot_val,
                          slot_written_pair, slot_written_slot, slot_written_val,
                          writers_slots, reading, latest,
                          start_write_slots, end_write_slots,
                          start_read_slots, end_read_slots |} |]
                   (the_slots ||| BITs0))
```

```
                          \ {| read_slot_pair, read_slot_slot,
                               read_slot_val, slot_written_pair,
                               slot_written_slot, slot_written_val,
                               writers_slots, reading, latest,
                               start_write_slots, end_write_slots,
                               start_read_slots, end_read_slots |})

Four_Slot_BIT1 =(((Fourslot_Writer ||| Fourslot_Reader)
                     [| {| read_slot_pair, read_slot_slot, read_slot_val,
                        slot_written_pair, slot_written_slot, slot_written_val,
                        writers_slots, reading, latest,
                        start_write_slots, end_write_slots,
                        start_read_slots, end_read_slots |} |]
                    (the_slots ||| BITs1))
                        \ {| read_slot_pair, read_slot_slot,
                             read_slot_val, slot_written_pair,
                             slot_written_slot, slot_written_val,
                             writers_slots, reading, latest,
                             start_write_slots, end_write_slots,
                             start_read_slots, end_read_slots |})

Four_Slot_BIT2 = (((Fourslot_Writer ||| Fourslot_Reader)
                     [| {| read_slot_pair, read_slot_slot, read_slot_val,
                        slot_written_pair, slot_written_slot, slot_written_val,
                        writers_slots, reading, latest,
                        start_write_slots, end_write_slots,
                        start_read_slots, end_read_slots |} |]
                    (the_slots ||| BITs2))
                        \ {| read_slot_pair, read_slot_slot,
                             read_slot_val, slot_written_pair,
                             slot_written_slot, slot_written_val,
                             writers_slots, reading, latest,
                             start_write_slots, end_write_slots,
                             start_read_slots, end_read_slots |})

Four_Slot_BIT3 = (((Fourslot_Writer ||| Fourslot_Reader)
                     [| {| read_slot_pair, read_slot_slot, read_slot_val,
                        slot_written_pair, slot_written_slot, slot_written_val,
                        writers_slots, reading, latest,
                        start_write_slots, end_write_slots,
                        start_read_slots, end_read_slots |} |]
                    (the_slots ||| BITs3))
                        \ {| read_slot_pair, read_slot_slot,
                             read_slot_val, slot_written_pair,
                             slot_written_slot, slot_written_val,
                             writers_slots, reading, latest,
                             start_write_slots, end_write_slots,
                             start_read_slots, end_read_slots |})


Four_Slot_BIT4_LB1 = (((Writer_LB1 ||| Reader_LB1)
                     [| {| read_slot_pair, read_slot_slot, read_slot_val,
                        slot_written_pair, slot_written_slot, slot_written_val,
                        writers_slots, reading, latest,
                        start_write_slots, end_write_slots,
                        start_read_slots, end_read_slots |} |]
                    (the_slots ||| BITs4))
                        \ {| read_slot_pair, read_slot_slot,
                             read_slot_val, slot_written_pair,
                             slot_written_slot, slot_written_val,
                             writers_slots, reading, latest,
                             start_write_slots, end_write_slots,
                             start_read_slots, end_read_slots |})
```

```
Four_Slot_BIT4_LB2 = (((Writer_LB2 ||| Reader_LB2)
                      [| {| read_slot_pair, read_slot_slot, read_slot_val,
                            slot_written_pair, slot_written_slot, slot_written_val,
                            writers_slots, reading, latest,
                            start_write_slots, end_write_slots,
                            start_read_slots, end_read_slots |} |]
                         (the_slots ||| BITs4))
                             \ {| read_slot_pair, read_slot_slot,
                                  read_slot_val, slot_written_pair,
                                  slot_written_slot, slot_written_val,
                                  writers_slots, reading, latest,
                                  start_write_slots, end_write_slots,
                                  start_read_slots, end_read_slots |})

Four_Slot_BIT5_LB1 = (((Writer_LB1 ||| Reader_LB1)
                      [| {| read_slot_pair, read_slot_slot, read_slot_val,
                            slot_written_pair, slot_written_slot, slot_written_val,
                            writers_slots, reading, latest,
                            start_write_slots, end_write_slots,
                            start_read_slots, end_read_slots |} |]
                         (the_slots ||| BITs5))
                             \ {| read_slot_pair, read_slot_slot,
                                  read_slot_val, slot_written_pair,
                                  slot_written_slot, slot_written_val,
                                  writers_slots, reading, latest,
                                  start_write_slots, end_write_slots,
                                  start_read_slots, end_read_slots |})

Four_Slot_BIT5_LB2 = (((Writer_LB2 ||| Reader_LB2)
                      [| {| read_slot_pair, read_slot_slot, read_slot_val,
                            slot_written_pair, slot_written_slot, slot_written_val,
                            writers_slots, reading, latest,
                            start_write_slots, end_write_slots,
                            start_read_slots, end_read_slots |} |]
                         (the_slots ||| BITs5))
                             \ {| read_slot_pair, read_slot_slot,
                                  read_slot_val, slot_written_pair,
                                  slot_written_slot, slot_written_val,
                                  writers_slots, reading, latest,
                                  start_write_slots, end_write_slots,
                                  start_read_slots, end_read_slots |})

Four_Slot_BIT6_LB1 = (((Writer_LB1 ||| Reader_LB1)
                      [| {| read_slot_pair, read_slot_slot, read_slot_val,
                            slot_written_pair, slot_written_slot, slot_written_val,
                            writers_slots, reading, latest,
                            start_write_slots, end_write_slots,
                            start_read_slots, end_read_slots |} |]
                         (the_slots ||| BITs6))
                             \ {| read_slot_pair, read_slot_slot,
                                  read_slot_val, slot_written_pair,
                                  slot_written_slot, slot_written_val,
                                  writers_slots, reading, latest,
                                  start_write_slots, end_write_slots,
                                  start_read_slots, end_read_slots |})

Four_Slot_BIT6_LB2 = (((Writer_LB2 ||| Reader_LB2)
                      [| {| read_slot_pair, read_slot_slot, read_slot_val,
                            slot_written_pair, slot_written_slot, slot_written_val,
                            writers_slots, reading, latest,
                            start_write_slots, end_write_slots,
                            start_read_slots, end_read_slots |} |]
                         (the_slots ||| BITs6))
                             \ {| read_slot_pair, read_slot_slot,
```

```
                    read_slot_val, slot_written_pair,
                    slot_written_slot, slot_written_val,
                    writers_slots, reading, latest,
                    start_write_slots, end_write_slots,
                    start_read_slots, end_read_slots |})
```

— Monotonic Implementations —

Definitions of the monotonic ACM (that transmits a monotonically increasing set of values) with the different versions of the bit models and local variables.

```
Mono_BIT0  = ((Write_Act(1) ||| Read_Act(0))
                    [| {| start_write, end_read, start_read, end_write |} |]
              Four_Slot_BIT0)
                    \ {| start_write, end_read, start_read, end_write |}

Mono_BIT1  = ((Write_Act(1) ||| Read_Act(0))
                    [| {| start_write, end_read, start_read, end_write |} |]
              Four_Slot_BIT1)
                    \ {| start_write, end_read, start_read, end_write |}


Mono_BIT2  = ((Write_Act(1) ||| Read_Act(0))
                    [| {| start_write, end_read, start_read, end_write |} |]
              Four_Slot_BIT2)
                    \ {| start_write, end_read, start_read, end_write |}


Mono_BIT3  = ((Write_Act(1) ||| Read_Act(0))
                    [| {| start_write, end_read, start_read, end_write |} |]
              Four_Slot_BIT3)
                    \ {| start_write, end_read, start_read, end_write |}


Mono_BIT4_LB1  = ((Write_Act(1) ||| Read_Act(0))
                    [| {| start_write, end_read, start_read, end_write |} |]
                 Four_Slot_BIT4_LB1)
                    \ {| start_write, end_read, start_read, end_write |}

Mono_BIT4_LB2  = ((Write_Act(1) ||| Read_Act(0))
                    [| {| start_write, end_read, start_read, end_write |} |]
                 Four_Slot_BIT4_LB2)
                    \ {| start_write, end_read, start_read, end_write |}

Mono_BIT5_LB1  = ((Write_Act(1) ||| Read_Act(0))
                    [| {| start_write, end_read, start_read, end_write |} |]
                 Four_Slot_BIT5_LB1)
                    \ {| start_write, end_read, start_read, end_write |}

Mono_BIT5_LB2  = ((Write_Act(1) ||| Read_Act(0))
                    [| {| start_write, end_read, start_read, end_write |} |]
                 Four_Slot_BIT5_LB2)
                    \ {| start_write, end_read, start_read, end_write |}

Mono_BIT6_LB1  = ((Write_Act(1) ||| Read_Act(0))
                    [| {| start_write, end_read, start_read, end_write |} |]
                 Four_Slot_BIT6_LB1)
                    \ {| start_write, end_read, start_read, end_write |}
```

```
Mono_BIT6_LB2  = ((Write_Act(1) ||| Read_Act(0))
                     [| {| start_write, end_read, start_read, end_write |} |]
                  Four_Slot_BIT6_LB2)
                       \ {| start_write, end_read, start_read, end_write |}
```

— Assertions —

The assertions that have been used with FDR to investigate properties of the 4-slot implementation.

```
-- assert (Four_Slot_BIT0 \ {| start_write, end_read, start_read, end_write |})
--          [T=  Incoherence_Spec
assert SemiRegACM_Spec [T= Four_Slot_BIT0
-- assert SemiRegACM_Spec [F= Four_Slot_BIT0
assert RegACM_Spec [T= Four_Slot_BIT0
-- assert RegACM_Spec [F= Four_Slot_BIT0
assert STOP [T= Mono_BIT0
assert Pool_Spec [T=  Four_Slot_BIT0
-- assert Pool_Spec [F=  Four_Slot_BIT0

----
-- assert (Four_Slot_BIT1 \ {| start_write, end_read, start_read, end_write |})
--          [T=  Incoherence_Spec
assert SemiRegACM_Spec [T= Four_Slot_BIT1
-- assert SemiRegACM_Spec [F= Four_Slot_BIT1
assert RegACM_Spec [T= Four_Slot_BIT1
-- assert RegACM_Spec [F= Four_Slot_BIT1
assert STOP [T= Mono_BIT1
assert Pool_Spec [T=  Four_Slot_BIT1
-- assert Pool_Spec [F=  Four_Slot_BIT1

----
--assert (Four_Slot_BIT2 \ {| start_write, end_read, start_read, end_write |})
--          [T=  Incoherence_Spec
assert SemiRegACM_Spec [T= Four_Slot_BIT2
-- assert SemiRegACM_Spec [F= Four_Slot_BIT2
assert RegACM_Spec [T= Four_Slot_BIT2
-- assert RegACM_Spec [F= Four_Slot_BIT2
assert STOP [T= Mono_BIT2
assert Pool_Spec [T=  Four_Slot_BIT2
-- assert Pool_Spec [F=  Four_Slot_BIT2

----
--assert (Four_Slot_BIT3 \ {| start_write, end_read, start_read, end_write |})
--          [T=  Incoherence_Spec
assert SemiRegACM_Spec [T= Four_Slot_BIT3
-- assert SemiRegACM_Spec [F= Four_Slot_BIT3
assert RegACM_Spec [T= Four_Slot_BIT3
-- assert RegACM_Spec [F= Four_Slot_BIT3
assert STOP [T= Mono_BIT3
assert Pool_Spec [T=  Four_Slot_BIT3
-- assert Pool_Spec [F=  Four_Slot_BIT3

----
--assert (Four_Slot_BIT4_LB1 \ {| start_write, end_read, start_read, end_write |})
--          [T=  Incoherence_Spec
assert SemiRegACM_Spec [T= Four_Slot_BIT4_LB1
-- assert SemiRegACM_Spec [F= Four_Slot_BIT4_LB1
```

```
assert RegACM_Spec [T= Four_Slot_BIT4_LB1
-- assert RegACM_Spec [F= Four_Slot_BIT4_LB1
assert STOP [T= Mono_BIT4_LB1
assert Pool_Spec [T=  Four_Slot_BIT4_LB1
-- assert Pool_Spec [F=  Four_Slot_BIT4_LB1

----
--assert (Four_Slot_BIT4_LB2 \ {| start_write, end_read, start_read, end_write |})
--           [T=  Incoherence_Spec
assert SemiRegACM_Spec [T= Four_Slot_BIT4_LB2
-- assert SemiRegACM_Spec [F= Four_Slot_BIT4_LB2
assert RegACM_Spec [T= Four_Slot_BIT4_LB2
-- assert RegACM_Spec [F= Four_Slot_BIT4_LB2
assert STOP [T= Mono_BIT4_LB2
assert Pool_Spec [T=  Four_Slot_BIT4_LB2
-- assert Pool_Spec [F=  Four_Slot_BIT4_LB2

----
--assert (Four_Slot_BIT5_LB1 \ {| start_write, end_read, start_read, end_write |})
--           [T=  Incoherence_Spec
assert SemiRegACM_Spec [T= Four_Slot_BIT5_LB1
-- assert SemiRegACM_Spec [F= Four_Slot_BIT5_LB1
assert RegACM_Spec [T= Four_Slot_BIT5_LB1
-- assert RegACM_Spec [F= Four_Slot_BIT5_LB1
assert STOP [T= Mono_BIT5_LB1
assert Pool_Spec [T=  Four_Slot_BIT5_LB1
-- assert Pool_Spec [F=  Four_Slot_BIT5_LB1

----
--assert (Four_Slot_BIT5_LB2 \ {| start_write, end_read, start_read, end_write |})
--           [T=  Incoherence_Spec
assert SemiRegACM_Spec [T= Four_Slot_BIT5_LB2
-- assert SemiRegACM_Spec [F= Four_Slot_BIT5_LB2
assert RegACM_Spec [T= Four_Slot_BIT5_LB2
-- assert RegACM_Spec [F= Four_Slot_BIT5_LB2
assert STOP [T= Mono_BIT5_LB2
assert Pool_Spec [T=  Four_Slot_BIT5_LB2
-- assert Pool_Spec [F=  Four_Slot_BIT5_LB2

----
--assert (Four_Slot_BIT6_LB1 \ {| start_write, end_read, start_read, end_write |})
--           [T=  Incoherence_Spec
assert SemiRegACM_Spec [T= Four_Slot_BIT6_LB1
-- assert SemiRegACM_Spec [F= Four_Slot_BIT6_LB1
assert RegACM_Spec [T= Four_Slot_BIT6_LB1
-- assert RegACM_Spec [F= Four_Slot_BIT6_LB1
assert STOP [T= Mono_BIT6_LB1
assert Pool_Spec [T=  Four_Slot_BIT6_LB1
-- assert Pool_Spec [F=  Four_Slot_BIT6_LB1

----
--assert (Four_Slot_BIT6_LB2 \ {| start_write, end_read, start_read, end_write |})
--           [T=  Incoherence_Spec
assert SemiRegACM_Spec [T= Four_Slot_BIT6_LB2
-- assert SemiRegACM_Spec [F= Four_Slot_BIT6_LB2
assert RegACM_Spec [T= Four_Slot_BIT6_LB2
-- assert RegACM_Spec [F= Four_Slot_BIT6_LB2
assert STOP [T= Mono_BIT6_LB2
assert Pool_Spec [T=  Four_Slot_BIT6_LB2
-- assert Pool_Spec [F=  Four_Slot_BIT6_LB2
```