Proving the Correctness of Simpson’s 4-slot ACM Using An Assertional Rely-Guarantee Proof Method

N. Henderson
BAE SYSTEMS DCSC, University of Newcastle-Upon-Tyne, UK
neil.henderson@ncl.ac.uk
June 6, 2003

Abstract

This paper describes a rely-guarantee proof to show that Simpson’s 4-slot single-reader, single-writer ACM is Lamport atomic (as described fully in the paper). First an abstract ACM specification is proved Lamport atomic using an exhaustive assertional method. A formal model of Simpson’s 4-slot is then given and this has been proved to be a refinement of the abstract specification using Nipkow’s retrieve relation rule. Simpson’s 4-slot is then shown to be Lamport atomic using an interleaved concurrency rely-guarantee proof method for shared variable concurrency.

Keyword: asynchronous communication, rely-guarantee, assertion networks.

1 Introduction

Inter-process communication is vital in any distributed system. One means of facilitating this communication is by using Asynchronous Communication Mechanisms (ACMs). ACMs are essentially shared variables that can be used to implement such inter-process communication without constraining the timing of the accesses of their reader(s) and writer(s). In this paper a particular implementation of an ACM is proved to be Lamport atomic (this term is fully defined in Section 3). Although, in general, ACMs can have multiple readers and/or writers the ACM considered in this paper only supports a single reader and a single writer. It is possible, however, for the reader of an ACM to end one read and start the next one while a single write is in progress and so multiple reads can overlap with a single write. Similarly multiple writes can overlap a single read. An item written to an ACM may be read by the reader a number of times and it is also possible that items written may not be read at all, because they are overwritten before the reader attempts to read them\(^1\). In the case of Simpson’s 4-slot, as the name implies, there are 4 slots available to hold data. When a read starts the reader is directed to the slot containing the latest complete item of data. If the writer starts to write a new item of data while the read is in progress it is directed to a different slot, to ensure that it does not overwrite part of the item that is being read. Similarly, if a read starts while a write is in progress, the reader is directed to a different slot to the one the writer is accessing. This ensures that the writer does not corrupt any data item while it is being read.

There are many interesting features of ACMs, for example they provide a means of decoupling the temporal interactions between communicating processes and they are robust against deadlock (for example if the writer is held up the reader can re-read the latest item written).

\(^1\)The asynchronous communication that ACMs support is therefore to be distinguished from the model of “asynchronous communication” supported by (infinite) buffers, where all items written are read by the reader (normally in the order that they were written).
Simpson’s 4-slot is an implementation of an ACM that is particularly efficient, which has been developed and used in the defence sector [23, 24, 26]. In [24], Simpson gave implementations of communication mechanisms that are implemented using 1, 2, 3, and 4-slots. The 1-slot mechanism can only be used where it is certain that the reader and writer will not access the shared memory at the same time. This non-interference could happen accidentally, but can only be guaranteed if some type of synchronisation mechanism is used, for example a Hoare monitor [9]. The 2-slot mechanism similarly requires some form of synchronisation, otherwise the mechanism cannot guarantee data coherence or freshness, depending on the implementation (if the reader and writer processes access the shared variable at the same time either: the writer may partially overwrite the item that is being read, and coherence will be lost; or the reader may need to re-read the same item many times, even though newer items have been written, and will not get the freshest item). The 3-slot mechanism almost implements an ACM, but loses coherence and/or freshness if the reader and writer interleave in a particular manner. Simpson gives an additional timing constraint which, if it can be guaranteed, makes the 3-slot behave in the same way as an ACM. He then gave an implementation of an ACM [24], which uses 4-slots to communicate the data. This ACM is challenging to analyse, despite its deceptive brevity, because of the unconstrained manner in which the reader and writer can interact: it is this mechanism that is analysed in this paper.

A proof is given in [8] that the 4-slot is Lamport atomic [14, 15], using Nipkow’s retrieve relation proof rules [16, 17, 13], subject to certain assumptions about the atomicity of the actions of the reader and writer. These assumptions mean that the reader and writer can only interfere with each other at certain points, by grouping a number of actions of the reader and writer into atomic instructions. This formal model of the 4-slot does not capture the fully asynchronous nature of the mechanism.

This paper gives a full correctness proof for the 4-slot that shows it is Lamport atomic, when these atomicity assumptions are relaxed and the read and write actions can interleave in an unconstrained manner, using the rely-guarantee method [10, 12, 11]. The formal model of the 4-slot used for the proofs is in the PVS logic [18], and is the same as that in [8]. The proofs have been completed using the PVS theorem prover [18] with the interleaved concurrency rely-guarantee proof method for shared variable concurrency from [6]. The proofs therefore still assume the individual actions of the reader and writer are atomic, whereas, for example in hardware implementations, the reader and writer can clash when they access control variables in the ACM.

The rest of the paper is organised as follows: Section 2 describes Simpson’s 4-slot implementation; Section 3 gives a correctness proof for an abstract specification of atomicity; Section 4 gives details of a correctness proof for the 4-slot based on the rely-guarantee method from [6]; Section 5 describes related work (in particular correctness proofs for the 4-slot using different proof methods, for example CSP [21, 5, 4, 20], Petri-nets [28] and Role modelling [25]); and Section 6 gives the conclusions from this work.

The contribution of this paper is to show how an ACM implementation can be proved to be Lamport atomic: first by proving correctness to an abstract specification, which treats the ACM as a black box with start and end read and write actions, using Nipkow’s retrieve relation rule [16, 17]; and then using an assertional rely-guarantee proof method [6] to show that it is still atomic when the individual actions of the reader and writer can interleave in an unconstrained manner. The particular ACM used as an example in the paper, i.e. Simpson’s 4-slot, has been analysed using a number of model checking techniques, but the use of the rely-guarantee method, with PVS, has given greater insight into the operation of the ACM than is gained by using a model checker.
2 Simpson’s 4-Slot ACM

In [23, 24] Simpson defined a fully asynchronous communication mechanism that maintained data-coherence and freshness, and which uses four slots for communicating data between the reader and writer.

In the 4-slot, bit control variables are used to ensure that the reader and writer are always directed to different slots, so the reader can never read values composed of partial items from more than one write. The 4-slot algorithm is deceptively simple, consisting of only five actions in the write operation and four actions in the read operation, and is shown in Table 1.

Table 1: The 4-slot mechanism

<table>
<thead>
<tr>
<th>Mechanism four slot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>type PairIndex = (p0, p1);</td>
</tr>
<tr>
<td>SlotIndex = (s0, s1);</td>
</tr>
<tr>
<td>var data: array[PairIndex, SlotIndex] of Data;</td>
</tr>
<tr>
<td>slot: array[PairIndex] of SlotIndex;</td>
</tr>
<tr>
<td>latest, reading: PairIndex;</td>
</tr>
<tr>
<td>procedure write (item: data);</td>
</tr>
<tr>
<td>var writepair: PairIndex;</td>
</tr>
<tr>
<td>writeindex: SlotIndex;</td>
</tr>
<tr>
<td>begin</td>
</tr>
<tr>
<td>writepair := not reading; (writerChoosesPair)</td>
</tr>
<tr>
<td>writeindex := not slot[writepair]; (writerChoosesSlot)</td>
</tr>
<tr>
<td>data[writepair, writeindex] := item; (write)</td>
</tr>
<tr>
<td>slot[writepair] := writeindex;</td>
</tr>
<tr>
<td>latest := writepair; (writerIndicatesPair)</td>
</tr>
<tr>
<td>end;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function read: Data;</th>
</tr>
</thead>
<tbody>
<tr>
<td>var readpair: PairIndex;</td>
</tr>
<tr>
<td>readindex: SlotIndex;</td>
</tr>
<tr>
<td>begin</td>
</tr>
<tr>
<td>readpair := latest; (readerChoosesPair)</td>
</tr>
<tr>
<td>reading := readpair; (readerIndicatesPair)</td>
</tr>
<tr>
<td>readindex := slot[readpair]; (readerChoosesSlot)</td>
</tr>
<tr>
<td>read := data[readpair, readindex]; (read)</td>
</tr>
<tr>
<td>end;</td>
</tr>
</tbody>
</table>

Simpson, in a later paper, gave a new algorithm for the 4-slot, which essentially reverses the order in which the reader and writer choose the pair and slot to read from or write to, [26]. That variant is not considered in this paper.

It is the intention of the design of the 4-slot mechanism that the reader and writer cannot access the same slot at the same time i.e. that it maintains coherence of the items communicated. It is also intended to support data freshness i.e. the reader should read the most recently written item.

The requirements for the 4-slot mechanism to maintain data coherence and freshness can be summarised by saying that the 4-slot should be Lamport atomic [8, 26] (the properties of a Lamport atomic ACM are described in the next section).
3 An Abstract Specification of Lamport Atomic ACMs

In [8] an abstract specification of Lamport atomic ACMs was given, which relied on an assumption about the relative speed of the reader and writer\(^2\). This section gives a revised specification which does not rely on this assumption. This specification assumes that the ACM has a single reader and single writer, which means that while multiple reads can overlap with a single write (because one read ends and a new read starts during the write), and vice versa, it is not possible for a read to overlap another read or a write to overlap another write.

The properties of Lamport atomic ACMs are:

1. The writer overwrites items in the ACM. This means that the reader may not read all of the items written if the writer is faster than the reader.
2. The reader may re-read items multiple times, if it is faster than the writer.
3. The reader must read items in the order they are written, so that, once it has read a particular item, it cannot subsequently read one that was written earlier.
4. The reader and writer can access the ACM in a totally asynchronous manner.
5. Reads and writes appear to have occurred in a particular order (as if the entire read and write operations were Hoare atomic [9] and interleaved with each other).

The last property is not characterised directly in the specification. 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. The presence of an item in the sequence models its availability to the reader\(^3\). The model has two booleans to record whether the reader and/or writer are accessing the mechanism, and uses the variables nextIndex, indexRead and firstIndexAvailable to record the indices of the next item to be written, the last item read and the first item available to a read. The specification has four operations, \texttt{start\_read}, \texttt{end\_read}, \texttt{start\_write} and \texttt{end\_write}, each of which is assumed to be atomic, which are given below\(^4\):

\begin{verbatim}
Val: noneMPT\_TYPE

Data: TYPE = [\# index: nat, val: Val \#]

Val\_Sequence: TYPE = \{fin\_seq: finite\_sequence[Data] \mid fin\_seq.length \geq 1\}

\{seq: Val\_Sequence \cup \{d: Data\}: Val\_Sequence = \{\# length := 1, seq := (\lambda x: below[1]): d \#\} \circ seq

Abs\_State: TYPE = [\# vals: Val\_Sequence, 
writerAccess: bool, readerAccess: bool, 
nexIndex: nat, 
indexRead: nat, 
firstIndexAvailable: nat \#]

\textbf{start\_read:} has a precondition that the reader is not already accessing the mechanism. The operation changes the readerAccess boolean to true to indicate that the reader is accessing the ACM, and removes items from the sequence of values that are not available to be read: if the sequence is of length one it is left unchanged, because there is always an item available to be read; and if the length of the sequence is greater than one and the writer
\end{verbatim}

\(^2\)It assumed that if the reader accessed the ACM more than once during a single write, it did not read the item written by the writer in that write access.

\(^3\)Each item is given a unique sequence number in the model, so that it is possible to reason about the order in which these items are written.

\(^4\)This paper uses the encoding of VDM-SL operations developed in [2].
is not currently accessing the mechanism the sequence is shortened to contain only the latest item, if the writer is accessing the mechanism the latest two items are available to be read. It also sets firstIndexAvailable equal to the index of the first item available to the reader.

\[
\text{pre\_start\_read}(\text{prot\_state}) : \text{bool} = \text{prot\_readerAccess = FALSE}
\]

\[
\text{post\_start\_read}(p : (\text{pre\_start\_read}))(\text{prot\_state}) : \text{bool} = \\
\begin{cases}
    \text{false} & \text{if } p\_\text{val\_length} = 1 \\
    \text{true} & \text{if } p\_\text{writerAccess} \\
    \text{false} & \text{if } p\_\text{readerAccess} := \text{true, firstIndexAvailable := -1}
\end{cases}
\]

\[
\text{end\_read: has a precondition that the reader is accessing the mechanism. The operation} \\
\text{non-deterministically chooses one of the available items from the sequence to read and} \\
\text{shortens the sequence to remove all earlier items to ensure they are not available for future} \\
\text{reads. It also sets the readerAccess boolean to false to show that the read has ended, and} \\
\text{indexRead to the index of the item read (which is the first item in the sequence after} \\
\text{the read has finished).}
\]

\[
\text{pre\_end\_read}(\text{prot\_state}) : \text{bool} = \text{prot\_readerAccess = TRUE}
\]

\[
\text{post\_end\_read}(p : (\text{pre\_end\_read}))(\text{prot\_state, read\_item : Val}) : \text{bool} = \\
\begin{cases}
    \text{false} & \text{if } p\_\text{val\_length} > 1 \\
    \text{true} & \text{if } p\_\text{val\_length} = 1
\end{cases}
\]

\[
\text{start\_write: has a precondition that the writer is not already accessing the ACM. It adds} \\
\text{the item being written to the sequence, increments nextIndex, and sets the writerAccess} \\
\text{boolean to true to show that the writer is accessing the mechanism.}
\]

\[
\text{pre\_start\_write}(\text{prot\_state}) : \text{bool} = \text{prot\_writerAccess = false}
\]

\[
\text{write\_parameter: TYPE = [\# p1 : (\text{pre\_start\_write} \_\text{write\_parameter}) \_\text{val : Val \#}]} \\
\text{post\_start\_write}(p : \text{write\_parameter})(\text{prot\_state}) : \text{bool} = \\
\text{LET new\_item : Data := (\# index := p\_p1\_nextIndex, val := p\_val \#) IN} \\
\text{prot = p\_p1 \_\text{with vals := p\_p1\_vals} \cup \{\_\text{new\_item}\},} \\
\text{writer\_access := true,} \\
\text{next\_index := p\_p1\_next\_index + 1]}
\]

\[
\text{start\_write: [p : write\_parameter \rightarrow (post\_start\_write(p)) ]}
\]

5
**end_write:** has a precondition that the writer is accessing the ACM. If the reader is accessing the mechanism it leaves the sequence unchanged, because there is no way of knowing which item the reader has chosen to read, otherwise it shortens the sequence to one item (because this is the only item available to the next read), and sets the `writerAccess` boolean to false to indicate that the write has finished.

\[
\text{pre\_end\_write} (prot: \text{Abs\_State}) : \text{bool} = \text{prot\_writerAccess = TRUE} \\
\text{post\_end\_write} (p: (\text{pre\_end\_write}))(prot: \text{Abs\_State}) : \text{bool} = \\
(p'\text{readerAccess} = \text{TRUE} \Rightarrow \text{prot} = p \text{ with } [\text{writerAccess} := \text{FALSE}]) \land \\
(p'\text{readerAccess} = \text{FALSE} \Rightarrow \\
\text{prot} = p \text{ with } \text{vals} := p'\text{vals} \limp \text{val}(0, 0) \land \text{writerAccess} := \text{FALSE}) \\
\text{end\_write} : [p: (\text{pre\_end\_write}) \rightarrow (\text{post\_end\_write}(p))] \\
\]

![State Model for the Abstract Specification of an Atomic ACM](image)

Figure 1: State Model for the Abstract Specification of an Atomic ACM

This relatively small specification has been proved Lamport atomic using an exhaustive assertional proof method [3] with PVS. The state space of the model is shown in Figure 1. Each of the states has an assertion associated with it (lambda expressions have been used here so that the same assertions can be used in multiple places in the PVS definitions and then expanded in-line when completing the proofs):

\[
\text{noReader\_noWriter\_assertion: [Abs\_State -> bool]} = \\
\lambda \text{(abs: Abs\_State)}: \\
\text{abs'indexRead} = \text{abs'nextIndex} - \text{abs'vals'length} \land \\
\text{abs'firstIndexAvailable} = \text{abs'nextIndex} - \text{abs'vals'length} \land \\
\text{finseq\_app}(\text{abs'vals})(0)\text{'index} = \text{abs'nextIndex} - 1 \\
\text{reader\_noWriter\_assertion: [Abs\_State -> bool]} = \\
\lambda \text{(abs: Abs\_State)}: \\
\text{abs'indexRead} = \text{abs'nextIndex} - \text{abs'vals'length} \land \\
\text{abs'firstIndexAvailable} = \text{abs'nextIndex} - \text{abs'vals'length} \land \\
\text{finseq\_app}(\text{abs'vals})(0)\text{'index} = \text{abs'nextIndex} - 1 \\
\text{noReader\_writer\_assertion: [Abs\_State -> bool]} = \\
\lambda \text{(abs: Abs\_State)}: \\
\text{abs'indexRead} = \text{abs'nextIndex} - \text{abs'vals'length} \land \\
\text{abs'firstIndexAvailable} = \text{abs'nextIndex} - \text{abs'vals'length} \land \\
\text{finseq\_app}(\text{abs'vals})(0)\text{'index} = \text{abs'nextIndex} - 1 \\
\text{reader\_writer\_assertion: [Abs\_State -> bool]} = \\
\lambda \text{(abs: Abs\_State)}: \\
\text{abs'indexRead} = \text{abs'nextIndex} - \text{abs'vals'length} \land \\
\text{abs'firstIndexAvailable} = \text{abs'nextIndex} - \text{abs'vals'length} \land \\
\text{finseq\_app}(\text{abs'vals})(0)\text{'index} = \text{abs'nextIndex} - 1 \\
\]

6
The following proof has been discharged for each of the operations, from each of the states when they can be executed, to show that the operations do not invalidate the assertions in the respective target states of the transitions:

\[
\forall (s_1, s_2; \text{Abs}_{\text{State}}): \text{pre}(s_1) \land \text{startStateAssertion}(s_1) \\
\land s_2 = \text{op}(s_1) \Rightarrow \text{targetStateAssertion}(s_2)
\]

Finally the specification is proved Lamport atomic by showing that the following assertion always holds when the \textit{endRead} operation is executed:

\[
\text{Lamport}: \text{Abs}_{\text{State}} \rightarrow \text{bool} = \\
\lambda (s_1, s_2; \text{Abs}_{\text{State}}): s_1 \text{indexRead} \leq s_2 \text{indexRead} \\
\land s_2 \text{firstIndexAvailable} \leq s_2 \text{indexRead} \land s_2 \text{nextIndex} - 1 \geq s_2 \text{indexRead}
\]

This is equivalent to a guarantee condition, and is described as follows:

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

2. \textit{firstIndexAvailable} gives the index number of the first item on the list when a read starts (the first item that is available to the reader for that read).

3. \textit{indexRead} is the index number of the item that is read.

4. The above assertion guarantees 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, less than the index to be used for the next item written, and that it is the same or a later item than that read last time.

The remainder of the properties that are required to guarantee Lamport atomicity are encoded directly into the specification, 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; and 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.

The specification in this section corrects that in [8]. The proofs described there, that used Nipkow's retrieve relation proof rule [16, 17] to show the 4-slot is a refinement of this specification, have been repeated for this revised specification. These proofs, however, are insufficient to show that the 4-slot implementation is Lamport atomic when the reader and writer can access the mechanism in a totally asynchronous manner. It is not clear how the implementation could be related directly to this specification using refinement. The exhaustive proof method used to show the specification is Lamport atomic could be used to prove the implementation is also Lamport 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. Verification proofs for each of the states in the entire state space would then need to be discharged\footnote{It is anticipated that this would more than double the number of verification proofs that would need to be discharged to prove the 4-slot implementation is Lamport atomic compared to the proof method described in Section 4.}. While it is anticipated that these proofs would be easier to discharge than those for the rely-guarantee method described in Section 4, it would be a non trivial task to ensure that the entire state space is explored correctly. For these reasons this is not considered to be a practical solution, so it was necessary to explore other proof methods to show that the 4-slot implementation is Lamport atomic. The next section describes such a method, using an assertional rely-guarantee proof method for shared variable concurrency.
4 A Rely-Guarantee Proof for Simpson’s 4-slot

This section presents a proof that Simpson’s 4-slot ACM is Lamport atomic when the read and write actions can interleave in an unconstrained manner, using an interleaved concurrency assertional rely-guarantee method for shared variables from [6]. This involves producing assertion networks for the reader and writer, as described in Section 4.1, and showing that the read operations do not interfere with the writer and that the write operations do not interfere with the reader (the Acel semantics [1] described in [6]). The use of this proof method reduces the number of verification proofs that it is necessary to discharge to a manageable number: it is necessary to discharge three verification proofs for each of the states in the reader and writer networks, a total of 33 proofs. The correctness proof is split into two parts: the first part shows that the mechanism maintains coherence of the data items transmitted, and the second part proves that it is Lamport atomic. The proofs, which have all been discharged using the PVS theorem prover [18, 19], are based on the model of the 4-slot in the PVS logic given in Appendix A6.

4.1 Assertion Networks for the Reader and Writer

![Assertion Network for the Reader](image)

Figure 2: Assertion Network for the Reader

The assertion networks for the reader and writer are shown in Figure 2 and Figure 3 respectively. The networks both contain a transition labelled false, which leads to their respective termination states. This is to indicate that the reader and writer algorithms do not terminate once they have started. Assertions are associated with the states in the network and these are described in Section 4.2.

![Assertion Network for the Writer](image)

Figure 3: Assertion Network for the Writer

4.2 Correctness Proofs

This section gives details of the proofs for coherence and Lamport atomicity. The proofs in both cases follow the same format:

- For each write operation it is necessary to show:

\[
\forall (c1, c2: \text{Conc State}): \text{pre}\_\text{startState}(c1) \land \\
\text{startState}\_\text{Assertion}(c1) \land \text{(readerChoosePair}\_\text{Assertion}(c1)) \land \\
\text{(writerChoosePair}\_\text{Assertion}(c1)) \land \\
\text{post}\_\text{startState}(c2) \land \\
\text{assertion}\_\text{trans}(c1, c2)
\]

6The interested reader can download all of the PVS models and proof files described in this paper from http://homepages.cs.ncl.ac.uk/neil.henderson/fme2003.
(readerChoosesSlotAssertion(cs1)) \land
(readAssertion(cs1)) \land cs2 = writerOp(cs1) \Rightarrow
(targetStateAssertion(cs2)) \land (readerChoosesPairAssertion(cs2)) \land
(readerChoosesSlotAssertion(cs2)) \land (readAssertion(cs2))

i.e. that if the pre-condition of the operation and the assertion in the start state hold, then the assertion in the target state will hold after the operation has been executed; and furthermore that the operation does not interfere with any of the assertions in the states in the assertion network of the reader.

- Similarly for each read operation:

\[
\forall (cs1, cs2: ConcState): \text{preStartState}(cs1) \land
\text{startStateAssertion}(cs1) \land \text{writerChoosePairAssertion}(cs1) \land
\text{writerChoosesSlotAssertion}(cs1) \land \text{writeAssertion}(cs1) \land
\text{writerIndicatesSlotAssertion}(cs1) \land \text{writerIndicatesPairAssertion}(cs1) \land
\text{cs2 = readerOp}(cs1) \Rightarrow
\text{targetStateAssertion}(cs2) \land \text{writerChoosesPairAssertion}(cs2) \land
\text{writerChoosesSlotAssertion}(cs2) \land \text{writeAssertion}(cs2) \land
\text{writerIndicatesSlotAssertion}(cs2) \land \text{writerIndicatesPairAssertion}(cs2)
\]

i.e. if the pre-condition and the assertion in the start state hold, the assertion of the target state will hold after the operation has been executed. In addition the reader will not interfere with any of the assertions in the assertion network of the writer.

- It is then necessary to show for each transition that the assertions in the source and target states of the assertion network are strong enough to meet the guarantee condition. In the case of the coherence proof the interesting proofs are when the reader and writer are both accessing the slots in the ACM, and for the Lamport atomic proof the interesting proof is for end read.

It is not necessary to make any assertions in the coherence proof for the reader network states sr, lr1 and lr4 (when the reader is about to execute firstReaderChoosesPair, readerIndicatesPair, and readerChoosesPair respectively) and the writer network state lw5 (when the writer is about to execute writeChoosesPair) other than the respective values of the auxiliary variables next read instruction (nri) and next write instruction (nwi). These values are, therefore, merely stated in line in the proofs.

4.2.1 The Proof of Coherence

The proofs of coherence rely on the following assertions in the respective states of the network:

- readerChoosesSlotAssertion: \[\text{[ConcState \to bool]} = \lambda (cs: \text{ConcState}): \text{cs'}nri = \text{cs}' \Rightarrow \text{cs}' \text{pairReading} = \text{cs}' \text{reader}' \text{readerPair} \]

- readAssertion: \[\text{[ConcState \to bool]} = \lambda (cs: \text{ConcState}): \text{cs'}nri = \text{rd} \Rightarrow
\text{cs}' \text{pairReading} = \text{cs}' \text{reader}' \text{readerPair} \land
\text{cs}' \text{reader}' \text{readerPair} = \text{cs}' \text{writer}' \text{writerPair} \Rightarrow
(\neg \text{cs}' \text{wis Occurred} \Rightarrow \text{cs}' \text{reader}' \text{readerSlot} = \text{cs}' \text{slotWritten}(\text{cs}' \text{reader}' \text{readerPair})) \land
(\neg \text{cs}' \text{wis Occurred} \Rightarrow
(\text{cs} \text{rsSinceWis} \Rightarrow \text{cs}' \text{reader}' \text{readerSlot} =
\text{cs}' \text{slotWritten}(\text{cs}' \text{reader}' \text{readerPair})) \land
(\neg \text{cs} \text{rsSinceWis} \Rightarrow \neg \text{cs}' \text{reader}' \text{readerSlot} = \text{cs}' \text{slotWritten}(\text{cs}' \text{reader}' \text{readerPair}))))

- firstWriterChoosesPairAssertion: \[\text{[ConcState \to bool]} = \lambda (cs: \text{ConcState}): \text{cs'}nwi = \text{firstWcp} \Rightarrow \neg \text{cs}' \text{wis Occurred} \]

- writerChoosesSlotAssertion: \[\text{[ConcState \to bool]} = \lambda (cs: \text{ConcState}): \text{cs'}nwi = \text{wes} \Rightarrow \neg \text{cs}' \text{wis Occurred} \land
writeAssertion:[ConcState→bool]⇝
λ(cs: ConcState): cs'wmi = wr ⇒ ¬cs'wIsOccurred ∧
¬cs'writer'writerSlot = cs'slotWritten(cs'writer'writerPair)

writerIndicatesSlotAssertion:[ConcState→bool]⇝
λ(cs: ConcState): cs'wmi = wsi ⇒ ¬cs'wIsOccurred ∧
¬cs'writer'writerSlot = cs'slotWritten(cs'writer'writerPair)

writerIndicatesPairAssertion:[ConcState→bool]⇝
λ(cs: ConcState): cs'wip = wpi ⇒ ¬cs'wIsOccurred ∧
¬cs'writer'writerSlot = cs'slotWritten(cs'writer'writerPair)

These 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 interesting assertion is that for read, because the reader cannot rely on the local copy of the control variable for the slot it is accessing being the same as the value recorded in the mechanism. This is because only the writer has write access to the control variable in the mechanism and can change it after the reader has accessed it, but before the reader has used its local copy to access the relevant slot. Three different cases for this value therefore need to be considered when the reader is reading:

1. The writer has not got as far as indicating the slot it is writing to in the pair it is accessing in the current write operation. 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, in which case either:
   - The reader chose the slot to read from after the writer had indicated the new slot it had written to, in which case the reader’s local copy of the value will be the same as that in the control variable. The reader will access the same slot as the writer, but this is fine because the writer has finished writing the data. The reader may effectively get the latest data before it has been fully released.
   - 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. The reader will get the last item fully released.

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.8

The guarantee condition that needs to be established is that the reader and writer do not access the same slot in the mechanism at the same time, in order to preserve coherence of data. This requires that the assertions in the networks are sufficient to prove the following:

\[∀(cs: ConcState):\]
\[cs'nri = rd ∧ cs'nwi = wr ⇒\]
\[((¬cs'reader'readerPair = cs'writer'writerPair) ∨\]
\[(¬cs'reader'readerSlot = cs'writer'writerSlot)\]

This assertion guarantees that the reader and writer are in different pairs or different slots in the same pair in the mechanism when they are reading and writing data respectively. The reader and writer both rely on the other to only access the slot they have chosen.

---

7 The only cases of interest are when the reader and writer are in the same pair, otherwise they are by definition accessing different slots.

8 The reader cannot subsequently follow the writer to the new pair until the writer has indicated it has changed to the new pair, which is last action in the next complete write access.
4.2.2 The Proof of Atomicity

The proofs of Lamport atomicity use auxiliary variables to record extra information about the data items that are available to the reader in a similar method to the exhaustive proof for the abstract specification.

**newMaxFresh**: Incremented by the writer at start write, to indicate the index of the item to be written.

**maxFresh**: Used by the writer to indicate the index of the last item written.

**minFresh**: Used by the reader to record the index of the first item available at the start of a read.

**indexRead**: Used by the reader to record the index of the item read. It is compared with minFresh and newMaxFresh to ensure that the item is fresh, and with its previous value to ensure that the items are read in sequence.

**lastIndexRead**: Used by the reader to record the index of the last item read.

An example assertion from the assertion network for the reader, that is used to prove atomicity is:

```plaintext
readerChoosesPair_Assertion: [Conc_State -> bool] ==
  \lambda (cs: Conc_State).
  cs'new = rcp \Rightarrow cs'treader'treaderPair = cs'tpairReading \land
  (cs'tpairReading = cs'tpairWritten \land
   cs'treader'treaderPair = cs'twriter'twriterPair \land
   cs'treader'treaderPair = cs'tpairReading \Rightarrow
   (\neg cs'twinOccurred \Rightarrow
    cs'tminFresh \leq cs'tmaxFresh \land
    cs'tindexRead \leq cs'tmaxFresh \land
    cs'tindexRead \geq cs'tminFresh \land cs'tlastIndexRead \leq cs'tindexRead) \land
    (\neg cs'twriteOccurred \Rightarrow
     cs'tminFresh \leq cs'tnewMaxFresh \land
     cs'tindexRead \geq cs'tminFresh \land cs'tlastIndexRead \leq cs'tindexRead) \land
    (\neg cs'treadOccurred \Rightarrow
     cs'tminFresh \leq cs'tnewMaxFresh \land
     cs'tindexRead \leq cs'tminFresh \land
     cs'tlastIndexRead \leq cs'tindexRead) \land
     (cs'tpairReading = cs'tpairWritten \land
      \neg cs'treader'treaderPair = cs'twriter'twriterPair \land
      cs'treader'treaderPair = cs'tpairReading \Rightarrow
      cs'tminFresh \leq cs'tmaxFresh \land
      cs'tindexRead \leq cs'tminFresh \land
      cs'tlastIndexRead \leq cs'tindexRead) \land
      (\neg cs'tpairReading = cs'tpairWritten \land
       \neg cs'treader'treaderPair = cs'twriter'twriterPair \land
       cs'treader'treaderPair = cs'tpairReading \Rightarrow
       cs'tminFresh \leq cs'tmaxFresh \land
       cs'tindexRead \leq cs'tminFresh \land
       cs'tlastIndexRead \leq cs'tindexRead) \land
       (cs'tpairWritten = p_0 \Rightarrow
        cs'tminFresh \leq cs'tslot(p_1, cs'tslotWritten(p_1)) \land
        (cs'tpairWritten = p_1 \Rightarrow
         cs'tminFresh \leq cs'tslot(p_0, cs'tslotWritten(p_0)))
```
This assertion describes the different relationships between the auxiliary variables, depending on the values of the control variables in the mechanism, that are used to prove atomicity of the 4-slot, when the reader is about to execute the \texttt{readerChoosesPair} operation.

An example of assertion for the writer, when it is about to execute the \texttt{writerIndicatesPair} operation, which gives the values of the indices of the data items in the slots, depending on whether the writer has changed pair for this write or not, is:

\begin{verbatim}
writerIndicatesPair_Assertion: [Conc_Slot \rightarrow \text{bool}] =
\begin{array}{c}
\lambda (cs: \text{Conc_Slot}):
\begin{array}{l}
\text{cs'swi = wip} \Rightarrow \text{cs'wRead occurred} \land \\
\rightarrow \text{cs'pairWritten = cs'pairReading} \Rightarrow \\
\text{cs'PairWritten = cs'writer'writerPair} \land \\
\text{cs'writer'writerSlot} = \text{cs'slotWritten}(\text{cs'writer'writerPair}) \land \\
\text{cs'maxFresh} = \text{cs'newMaxFresh} - 1 \land \\
cs'newMaxFresh = \text{cs'slots}(\text{cs'writer'writerPair}, \\
\text{cs'writer'writerSlot})'\text{index} \land \\
(\text{cs'writer'writerPair} = \text{cs'pairWritten}) \Rightarrow \\
(\text{cs'slotWritten}(\text{cs'pairWritten}) = s0) \Rightarrow \\
\text{cs'slots}(\text{cs'pairWritten}, s1)'\text{index} \leq \text{cs'maxFresh} \land \\
(\text{cs'slotWritten}(\text{cs'pairWritten}) = s1) \Rightarrow \\
\text{cs'slots}(\text{cs'pairWritten}, s0)'\text{index} \leq \text{cs'maxFresh} \land \\
(\text{cs'pairWritten} = p0) \Rightarrow \\
\text{cs'slots}(p1, s0)'\text{index} \leq \text{cs'maxFresh} - 1 \land \\
\text{cs'slots}(p1, s1)'\text{index} \leq \text{cs'maxFresh} - 1) \land \\
(\text{cs'pairWritten} = p1) \Rightarrow \\
\text{cs'slots}(p0, s0)'\text{index} \leq \text{cs'maxFresh} - 1 \land \\
\text{cs'slots}(p0, s1)'\text{index} \leq \text{cs'maxFresh} - 1)) \land \\
(\rightarrow \text{cs'writer'writerPair} = \text{cs'pairWritten}) \Rightarrow \\
\text{cs'maxFresh} = \text{cs'slots}(\text{cs'pairWritten}, \\
\text{cs'slotWritten}(\text{cs'pairWritten})'\text{index} \land \\
(\text{cs'slotWritten}(\text{cs'pairWritten}) = s0) \Rightarrow \\
\text{cs'slots}(\text{cs'pairWritten}, s1)'\text{index} \leq \text{cs'maxFresh} - 1) \land \\
(\text{cs'slotWritten}(\text{cs'pairWritten}) = s1) \Rightarrow \\
\text{cs'slots}(\text{cs'pairWritten}, s0)'\text{index} \leq \text{cs'maxFresh} - 1) \land \\
(\text{cs'writer'writerSlot} = s0) \Rightarrow \\
\text{cs'slots}(\text{cs'writer'writerPair}, s1)'\text{index} \leq \text{cs'maxFresh} \land \\
(\text{cs'writer'writerSlot} = s1) \Rightarrow \\
\text{cs'slots}(\text{cs'writer'writerPair}, s0)'\text{index} \leq \\
\text{cs'maxFresh}))
\end{array}
\end{array}
\end{verbatim}

This specifies, for example the slot that has been written to contains the item with the latest index, and, if the writer has changed pairs both of the slots in the opposite pair contain items with indices at least one less than the index of the latest item.

Lamport atomicity requires that the following guarantee condition must be implied by the assertions in the networks when the transitions are executed:

\begin{verbatim}
\forall (cs: \text{Conc_Slot}) \cdot \text{cs'mi = ri} \Rightarrow \\
\text{cs'minFresh} \leq \text{cs'newMaxFresh} \land \\
\text{cs'indexRead} \leq \text{cs'newMaxFresh} \land \\
\text{cs'indexRead} \geq \text{cs'minFresh} \land \\
\text{cs'lastIndexRead} \leq \text{cs'indexRead})
\end{verbatim}

Where \text{minFresh} is the index of the first item available to the reader when it starts, \text{newMaxFresh} is the index of last item possibly written when the read finishes, \text{indexRead} gives the index of the item read, and \text{lastIndexRead} gives the index of the item read the last time\textsuperscript{9}. This requires two verification proofs for each transition, one to show that the guarantee condition holds before the operation associated with the transition is executed and the other to show that it still holds afterwards.

The use of the PVS theorem prover for discharging the verification proofs described in this section has been advantageous for a number of reasons. First PVS can be used to expose errors in the model. For example an error in the part of the model being verified may be indicated if part or all of a proof is unexpectedly discharged, or it is not possible to discharge all or part of a proof in the expected manner. For this reason it is advisable to work out the required tactics

\textsuperscript{9}The reader must read data items in the order they are written and always reads a fresh item.
to discharge the proofs in advance (apart from saving time that may be wasted in following the incorrect tactics). Second PVS can be used to validate a partial model that does not describe all of the required behaviours of the required system. It is then easier to revise the proofs as the model is extended than it would be with hand written proofs. This helps to build confidence in the model as it evolves. The use of PVS has also increased confidence in the correctness of the final model and proofs.

5 Related Work

The proofs given in this paper are based the assumption that the actions of the reader and writer are Hoare atomic, although they can interleave in an unrestricted manner. It is recognised that this assumption does not hold in many implementations. When it does not hold it is possible for the reader and writer to clash on reading and writing a particular control variable, or to attempt to read a control variable in the mechanism when it is changing. The control variables are single bits, but there is no guarantee that the reader of the variable will get the value written in these situations. It is also possible that reading a changing value will cause the reader to become metastable, in other words it may take the reader an arbitrary length of time to decide whether it has read a zero or a one. The author has been involved in work which models the 4-slot that recognises that the reader and writer can clash on accesses to control variables. These models, in CSP, take account of metastability effects that arise in such circumstances. This work [20] has shown, using the FDR model checker [7], that the 4-slot mechanism still preserves coherence of data, and is Lamport atomic even when metastability occurs. This positive result is important because in [22] Rushby has shown that the 4-slot is not Lamport atomic if it is implemented with Lamport safe control variables.

Simpson’s role model method [25, 27] has been used to prove correctness of the 4-slot, but this relies on an assumption about the behaviour of the read and write actions in the presence of metastability and clashes on control variables: that the reader of the variable gets the value either before or after the writer to the variable occurred. The 4-slot has also been proved correct using Petri-nets [29], and timed CSP [5, 4].

The method described in this paper has also been used with PVS to prove a 3-slot ACM implementation incorrect, because coherence of data is lost. The use of PVS helped to identify the sequence of actions of the reader and writer that would result in concurrent access by the reader and writer to the same data slot. It was then possible to show that if a timing constraint could be implemented to prevent this particular sequence of the actions the implementation would maintain coherence of the data transmitted\(^\text{10}\).

6 Conclusions

This paper has presented a rely-guarantee proof of the correctness of Simpson’s 4-slot ACM using the interleaved concurrency method for shared variable concurrency from [6]. The algorithm is deceptively simple, however the proofs are complex because of the unconstrained manner in which the read and write operations can interleave. The ACM has been proved correct using a number of different model checking methods, but the use of the rely-guarantee has provided much greater insight into how the algorithm operates than those methods. In addition the use of the interleaved concurrency method from [6] reduces the number of verification proofs to a manageable number (3 proofs for each of the states in the assertion networks in this case).

\(^{10}\)A revised 3-slot ACM implementation, from [30], that combines two of the actions of the original implementation into a single action to prevent the incorrect interleaving of actions has also been proved correct using PVS.
It was shown in [8] that the 4-slot implementation could be proved to be Lamport atomic with respect to an abstract implementation, subject to certain operations in the implementation being combined into atomic operations. A revised abstract specification is given in this paper with an exhaustive proof that this specification exhibits the desired Lamport atomic properties.

The paper then describes a proof using a rely-guarantee proof method for shared variable concurrency that shows that the implementation is Lamport atomic when the individual operations interleave in an unconstrained manner. The proofs described are sufficient to demonstrate correctness of the 4-slot when the control variables are Hoare atomic variables. In any implementation where this assumption does not hold it needs to be recognised that asynchronous accesses to control variables can result in an attempt to read such a variable when it is changing. In these circumstances it is possible that the reader will not return the value written, or that the reader may take an arbitrary length of time to decide whether it has read a zero or a one. It may be possible to extend the proofs to show correctness in these circumstances, but this would make the assertions significantly more complex, and the proofs will be daunting, if not intractable.

This paper shows that it is possible to use the rely-guarantee method to prove properties of asynchronous networks of processes even when they are correct due to emergent properties of their asynchronous operation, rather than due to specific guarantees provided by their component processes. It may be easier to prove such properties by model checking, but the advantage of the proof method described here is that it gives much greater insight into the operation of the algorithm. The requirement to give assertions that hold in the states of the transitions of the reader and writer require an understanding of how the two processes interact, and this level of understanding could not be obtained simply by model checking the correctness of the mechanism. For example, in completing the proofs it was shown that the reader can read an item coherently before it has been fully released by the writer: after writerIndicatesSlot but before writerIndicatesPair, in certain circumstances when the reader and writer are accessing the same pair of slots. In addition the requirement to prove that the assertions hold in the target state of each transition, after executing the operation associated with the transition, provided the assertions hold in the source node of the transition gives increased confidence in the correctness of the model. A model checker will give counter examples for an incorrect implementation, however it will only give a positive result when the implementation is correct. This may lead to inefficient implementations being used, while more elegant implementations, that could be found with a better understanding of how the reader and writer interact, are overlooked.

The rely-guarantee method described in this paper has allowed a full proof of correctness of the 4-slot implementation with respect to an infinite state specification of atomicity (an unbounded number of reads can overlap with a single write and vice versa). Such a proof is not possible with a model checker without some form of data abstraction, and there is a danger that such an abstraction could leave out the very property that invalidates the proofs.

The use of PVS to validate the models and discharge the proof obligations has increased confidence in the correctness of the models and proof, and assisted in finding and correcting errors in the models. It was also easier to validate partial models and extend the proofs as the models evolved than would be the case if the proofs were discharged by hand. In addition, if a minor modification were made to the protocol the existing proofs could be modified more easily to check the correctness of the amended implementation, than with hand written proofs.

7 Acknowledgements

The BAE SYSTEMS Dependable Computing Systems Centre funded this research. This work has benefited from conversations with Profs. C.B. Jones and H.R. Simpson and Drs. J.M. Armstrong and S.E. Paynter.
References


16
A  Formal Model of the 4-slot Implementation in PVS

This appendix contains the model of the 4-slot algorithm, in the PVS logic, that has been used in all of the proofs (apart from differences in auxiliary variables).

Val: `NONEMPTY_TYPE
Data: TYPE = [\* index: nat, val: Val \*]
PairIndex: TYPE = \{p0, p1\}
SlotIndex: TYPE = \{s0, s1\}
NextReadInstruction: TYPE = \{firstRcp, rcp, rip, rcs, rd\}
NextWriteInstruction: TYPE = \{firstWcp, wcp, wcs, wr, wis, wip\}
WriterState: TYPE = \# writerPair: PairIndex, writerSlot: SlotIndex, currentState: WriterNetworkState \#]
ReaderState: TYPE = \# readerPair: PairIndex, readerSlot: SlotIndex, currentState: ReaderNetworkState \#]

Conc_State: TYPE = \# pairWritten: PairIndex, slotWritten: [PairIndex \to SlotIndex], lastSlotWritten: [PairIndex \to SlotIndex], pairReading: PairIndex, slots: [PairIndex, SlotIndex \to Data], nri: NextReadInstruction, nwi: NextWriteInstruction, writer: WriterState, reader: ReaderState, maxFresh: nat \#]

The NextReadInstruction and NextWriteInstruction variables are used to model the progress of the reader and writer respectively through their algorithms. The operations of the reader and writer are encoded as follows\(^\text{11}\):

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 [pairPair := p.pairWritten, currentState := ir1]]

readerChoosesPair: \# (pre_readerChoosesPair) \to (post_readerChoosesPair(p))\]

pre_readerIndicatesPair(p: Conc_State): bool = p.nri = rip

post_readerIndicatesPair(p: (pre_readerIndicatesPair)(prot: Conc_State)): bool =
with [currentState := ir2]]

readerIndicatesPair: \# (pre_readerIndicatesPair) \to (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 [slotPair := p.slotWritten(p.reader.readerPair), currentState := ir3]]

readerChoosesSlot: \# (pre_readerChoosesSlot) \to (post_readerChoosesSlot(p))\]

\(^{11}\)The firstReaderChoosesPair and firstWriterChoosesPair operations are identical to the readerChoosesPair and writerChoosesPair operations, except for assignments to auxiliary variables, and are not shown here.
pre_read(p: Conc_State): bool = p'row = rd

post_read(p: (pre_read)) (prot: Conc_State, v: Val): bool =
    v = p'slots p'reader'readerPair, p'reader'readerSlot) v ∧
    prot = p WITH [row := rcp, reader := p'reader WITH [currentState := lr]]
read: [p: (pre_read) → (post_read(p))]

pre_writerChoosesPair(p: Conc_State): bool = p'niwi = wcp

post_writerChoosesPair(p: (pre_writerChoosesPair)) (prot: Conc_State): bool =
    (p'pairReading = p0 ⇒ prot = p WITH [niwi := wcs, writer := p'writer WITH [writerPair := p1, currentState := lw]]) ∧
    (p'pairReading = p1 ⇒ prot = p WITH [niwi := wcs, writer := p'writer WITH [writerPair := p0, currentState := lw]])
writerChoosesPair: [p: (pre_writerChoosesPair) → (post_writerChoosesPair(p))]

pre_writerChoosesSlot(p: Conc_State): bool = p'niwi = wcs

post_writerChoosesSlot(p: (pre_writerChoosesSlot)) (prot: Conc_State): bool =
    (p'slotWritten p'writer'writerPair = s0 ⇒
        prot = p WITH [niwi := wr, writer := p'writer
        WITH [writerSlot := s1, currentState := lw]]) ∧
    (p'slotWritten p'writer'writerPair = s1 ⇒
        prot = p WITH [niwi := wr, writer := p'writer
        WITH [writerSlot := s0, currentState := lw]])
writerChoosesSlot: [p: (pre_writerChoosesSlot) → (post_writerChoosesSlot(p))]

write: [p: writeParameter → (post_write(p))]}

writeParameter: TYPE = [# p1: (pre_write), v: Val #]

pre_write(p: Conc_State): bool = p'niwi = wr

post_write(p: writeParameter) (prot: Conc_State): bool =
    prot = p'p1 WITH [niwi := wr, (slots) p'p1'writer'writerPair, p'p1'writer'writerSlot)
     := (# index := p'p1'maxFresh, val := p'v #),
    writer := p'p1'writer WITH [currentState := lw]]
write: [p: writeParameter → (post_write(p))]

pre_writerIndicatesSlot(p: Conc_State): bool = p'niwi = wip

post_writerIndicatesSlot(p: (pre_writerIndicatesSlot)) (prot: Conc_State): bool =
    prot = p WITH [niwi := wip, (slotWritten) p'writer'writerPair := (p'writer'writerSlot),
    writer := p'writer WITH [currentState := lw]]
writerIndicatesSlot: [p: (pre_writerIndicatesSlot) → (post_writerIndicatesSlot(p))]

pre_writerIndicatesPair(p: Conc_State): bool = p'niwi = wip

post_writerIndicatesPair(p: (pre_writerIndicatesPair)) (prot: Conc_State): bool =
    prot = p WITH [niwi := wip, pairWritten := p'writer'writerPair, writer := p'writer WITH [currentState := lw]]
writerIndicatesPair: [p: (pre_writerIndicatesPair) → (post_writerIndicatesPair(p))]