SRI International

CSL Technical Report ● July 3, 2002

Model Checking Simpson's Four-Slot Fully Asynchronous Communication Mechanism

John Rushby
Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA

This research was supported by NASA Langley Research Center under contract NAS1-20334 and Cooperative Agreement NCC-1-377 with Honeywell Tucson, and by the DARPA MoBIES program under contract F33615-00-C-1700 with US Air Force Research Laboratory.

Abstract

Simpson's four-slot fully asynchronous communication mechanism allows single reader and writer processes to access a shared memory in such a way that interference between concurrent reads and writes is avoided, the reader always accesses the most recent data stored by the writer, and neither process need wait for the other. In computer science parlance, it is a means for implementing a wait-free atomic register.

We use the SAL model checking environment to examine this mechanism and show that concurrent reads and writes are indeed noninterfering but that access to the most recently written data requires the unattractive assumption that some of the control registers are already atomic. We exhibit counterexamples that reveal incorrect operation when the control registers are not atomic, and also when the mechanism is modified (following a suggestion of Simpson) so that control registers are written only when their values will be changed. We do successfully verify the algorithm when its control registers are assumed to be atomic.

The requirement for atomic control registers is unattractive: it means that any application that uses Simpson's mechanism must be accompanied by separate, strong evidence that its implementation of the control registers satisfies this requirement. We recommend formal examination of alternative algorithms that operate under weaker assumptions.

Contents

Chapter 1

Introduction

Embedded control systems must often move data between subsystems that do not share a common clock. Dual-ported memory commonly provides the interface between such subsystems. Because they do not share a clock, it is possible for one subsystem to be reading data from the shared memory at the same time the other is updating it. Without a protocol to ensure some kind of controlled access, it is possible that a read concurrent with a write could obtain partially written data that appears valid but is actually incorrect, and this could lead to incorrect and possibly unsafe operation.

One way to eliminate this danger would be to use semaphores or other locking mechanisms to enforce mutual exclusion on the reading and writing activities: that is, only one of them can be in progress at any one time. The problem with this approach is that one activity can block the other: for example, if the writer is much faster than the reader, it may almost always seize the lock and thereby starve the reader, or delay it beyond its deadline. Priorities, or enforced alternation, change the character of the problem, but do not eliminate it.

As described elsewhere [Rus02], the larger problem with such locking schemes is that they render the flow of control between components *bidirectional*. Kopetz [Kop99] defines interfaces that involve bidirectional flow of control as "composite" and argues convincingly that they should be eschewed in favor of "elementary" interfaces in which control flow is unidirectional. (*Data* flow may be bidirectional, but the task of tolerating external failures, and of application-independence, is greatly simplified by the unidirectional *control* flow of elementary interfaces.)

The need for elementary interfaces leads to protocols for concurrent reading and writing that do not use locks. One approach allows concurrent access to the same memory, but the reader is able to detect if it was changed during its operation; if so, the reader may either retry or reuse its previous data [Lam77]. An improvement over such "lock-free" protocols are "wait-free" protocols that guarantee the reader immediate access to the most recently written data; typically, these algorithms cause concurrent reads and writes to use different areas of memory.

Development and study of protocols for concurrent readers and writers has been pursued separately in the real-time and computer science communities. Anderson [And01] provides a useful summary from the computer science perspective, where the work of Lamport is particularly significant, not only in the development of protocols, but in defining the problem and introducing methods for its analysis [Lam86].

The Time Triggered Architecture (TTA) [TTT01, KG94] uses the Nonblocking Write (NBW) protocol of Kopetz and Reisinger [KR93]. Anderson [And01] observes that these authors were "apparently unaware that Lamport had invented this algorithm 16 years earlier" (referring to the algorithm of [Lam77]). In fact, there are two parts to NBW: the first part is lock-free (and is the same as Lamport's algorithm) but it is not wait-free; the second part adds multiple buffers to make it wait-free (and therefore truly nonblocking). The first wait-free construction was due to Peterson [Pet87], but it assumed atomic control registers; an algorithm that eliminated this assumption was published by Burns and Peterson [BP87], and counterexamples that revealed numerous flaws in that algorithm were described by Haldar and Vidyasankar [HV92]. Correct algorithms for this problem were known in computer science by the late 1980's [Tro89]. Independently, Simpson presented a different, and attractively simple algorithm [Sim90], and this is the one best known in the real-time and avionics communities. It also provides the inspiration for the second part of the NBW construction.

Because its mechanisms are fundamental, because it is widely used, and because it inspires the crucial wait-free aspect of the NBW protocol, we provide a formal examination of Simpson's algorithm. This algorithm has recently also been examined formally by Henderson and Paynter [HP02]; however, they consider only mutual exclusion, and not the additional desirable property that the reader always obtains recently written data. Furthermore, they conduct their examination by theorem proving (in PVS) whereas we note that, apart from the data communicated, the algorithm is finite state and can be examined by model checking. Clark [Cla00] examines several algorithms, including Simpson's, using Petri nets; we compare this work with ours in Chapter 3.

We use the SAL model checking environment to examine Simpson's algorithm and show that concurrent reads and writes are indeed noninterfering but that access to the most recently written data requires either very careful (and, we would argue, nonobvious) interpretation of the algorithm, or else the unwarranted assumption that some of its control registers are already atomic.

Chapter 2

Simpson's Algorithm and Its Formal Examination

Consider two processes—a reader and a writer—that communicate via a shared memory. The two processes and the memory are asynchronous, meaning they have no common clock and there are no assumptions about their relative rates of operation. Periodically, the writer has new data that it writes to the memory, and periodically the reader reads data from the memory. Without coordination, there is the possibility that the reader accesses the memory at the same time the writer is updating it. It is possible that the physics of memory access are such that concurrent reading and writing can cause the reader to obtain unpredictable values: for example, if a memory cell requires some time to "latch" the charge representing its state, or if the data spans several locations and must be read and written in stages. For all these reasons, it is possible that a read concurrent with a write may obtain a value that is only "half written." Lamport [Lam86] distinguishes several kinds of behavior that may obtain when memory (or "registers") are read and written concurrently.

Safe registers return the value most recently written when a read is not concurrent with a write; when a read is concurrent with a write, they can return any value. Strictly, Lamport requires that safe registers return a value consistent with their type—this can always be enforced if there is some way to recognize the type; Henderson and Paynter [HP02] argue that this behavior should be called *typesafe*, and suggest the term *persistent* be used for the weaker assumption that a read concurrent with a write can return absolutely *any* value. We will use the more traditional term, but will note when the weaker assumption is adequate.

Regular registers return the value most recently written when a read is not concurrent with a write; when a read is concurrent with a write, or series of writes, the read may return the value prior to the writes, the final value written, or any of the values written by the intermediate writes.

Atomic registers return the value most recently written when a read is not concurrent with a write; when a read is concurrent with a write, or series of writes, the behavior is consistent with them occurring in some serial order. This is stronger than a regular register in that a series of concurrent reads and writes will never cause a read to return data older than that returned by a previous read.

The challenge, stated in computer science terms, is to find an algorithm that implements an atomic register, using only safe registers. One plausible approach is to use two buffers or, following Simpson's terminology, "slots" so that the writer alternates between the two slots, and the reader accesses whichever of them is *not* the target of the current write (or of the next write, if there is none currently active). The problem here is that if a read overlaps multiple writes, then the second write will use the slot already being used by the reader—thus, the algorithm must be modified so that the writer avoids the slot currently used by the reader. Several control bits are needed to coordinate the reader and the writer in this manner, but all proposed algorithms are vulnerable to unfortunate interleavings that cause the reader and writer to access the same slot concurrently. It seems that additional slots are necessary. Simpson [Sim90] describes a three-slot algorithm, but finds that it suffers from the same vulnerabilities as the two-slot algorithm, so it looks as if at least four slots are necessary. (A three-slot algorithm can be made to work if we allow special hardware "compare-and-swap" instructions [CB97].)

Simpson's four-slot algorithm arranges the slots as two pairs; we can call the pairs `left` and `right`, and within each pair refer to the `top` and `bottom` slots. The writer operates on one pair, alternating between the `top` and `bottom` slot. A control bit `latest` indicates to which pair the writer most recently wrote, and another bit `index` associated with the pair indicates which slot of the pair was most recently written; both these control bits are written only by the writer. The reader uses a control bit `reading` to indicate from which pair it is reading; it sets this equal to `latest` when it begins a read; conversely the writer chooses the opposite pair when it starts a write operation.

The intuition is that the reader always reads the slot most recently completed by the writer; the writer finishes writing the other slot of that pair (if it is operating concurrently) and subsequently moves to the other pair. The algorithm can be described as follows (in generic pseudocode), where we use Booleans (rather than left-right, and top-bottom) for the control registers.

```
item: TYPE                                                    1

slot: ARRAY [boolean, boolean] OF item;
index: ARRAY [boolean] OF boolean;
latest, reading: boolean;

PROCESS writer(in: item);           PROCESS reader(out: item);
  wpair, windex: boolean;             rpair, rindex: boolean;
BEGIN                               BEGIN
  wpair := !reading;                  rpair := latest;
  windex := !index[wpair];            reading := rpair;
  slot[wpair][windex] := in;          rindex := index[rpair];
  index[wpair] := windex;             out := slot[rpair][rindex];
  latest := wpair;                  END;
END;
```

It is important to recognize that the individual assignment statements from concurrent executions of `reader` and `writer` may interleave or overlap arbitrarily, so the choice of the exact order in which the control variables are updated in each procedure is crucial. It is also important to recognize that the Boolean control variables, as well as the slot registers are assumed only to be safe—so that if the first statement of `reader` occurs concurrently with the last of `writer`, then the value read from `latest` by `reader` may be unrelated to that written by `writer`.

2.1 Formal Specification

We will conduct our formal examination of Simpson's algorithm using the SALenv model checking environment for SRI's SAL system [BGL+00]. This is the first published investigation conducted using SALenv, so it is presented in tutorial detail. The SAL language and semantics are described in its reference manual [DHOS01], but the following description is given in more operational terms.

The SAL notion corresponding to a process such as `reader` or `writer` is a `MODULE`, within which assignments to variables are described either as (variable-wise) definitions, or as guarded commands. The definitions within a module are executed atomically whereas we need to allow the possibility that assignments by the `reader` are interleaved with those from the `writer`, so guarded commands are the appropriate construction here. Guarded commands are given within a `TRANSITION`, so that a first attempt to encode the `reader` might look as follows (where the guards on the guarded commands are absent for the time being).

```
reader: MODULE                                                    2
BEGIN
% variable declarations and initializations
% omitted for the present
TRANSITION
  [
    rpair’ = latest
  []
    reading’ = rpair
  []
    rindex’ = index[rpair]
  []
    out’ = slot[rpair][rindex]
  ]
END;
```

Notice that the assignment statements are replaced by equations where the destination variables are primed: this is because a SAL transition specifies the relation between values of variables in the *new* (or *next*) state and those in the *old* (or *previous*) state; primes indicate values in the new state while undecorated names refer to those in the old state. Next, observe that semicolons between assignments are replaced by the `[]` symbol. This indicates choice and is needed to specify that the assignments within the `reader` are not performed as a single atomic step, but may be interleaved with those of the `writer`. Moreover, the elements of the choice are not ordered: at any step, the system may choose any element for execution. To force the execution to be sequential, we need to introduce a *program counter* for the reader `rpc` and to rewrite the specification appropriately. The `rpc` variable has no special significance to SAL, so we must explicitly include it in guards to select the correct assignment, and must explicitly update its value after each step. This leads to the following revised specification.

```
reader: MODULE
BEGIN
% variable declarations and initializations
% omitted for the present
TRANSITION
  [
    rpc = 0 --> rpair' = latest;
                rpc' = 1
  []
    rpc = 1 --> reading' = rpair;
                rpc' = 2
  []
    rpc = 2 --> rindex' = index[rpair];
                rpc' = 3
  []
    rpc = 3 --> out' = slot[rpair][rindex];
                rpc' = 0
  ]
END;
```

The *guards* such as `rpc = 2` are evaluated at each step; those that evaluate to `true` cause the group of assignments to the right of their arrow to be eligible for execution, and one of those eligible groups will be selected arbitrarily and executed. Notice that the assignments to the right of an arrow are grouped by means of semicolons.

Next, we need to indicate the types and status of the variables appearing in the `TRANSITION`. Variables may be declared as `OUTPUT`, meaning they are controlled (i.e., assigned) by this module alone; `INPUT`, meaning they are observed (i.e., read) by this module but controlled by some other module that will be composed with it; `LOCAL`, meaning they are used by this module alone; or `GLOBAL`, meaning they are both controlled and observed by this module and others. Here, `reading` is controlled by `reader`; `latest`, `slot`, and `index` are observed (they will be controlled by the `writer`); and `rpc`, `out`, `rindex`, and `rpair` are local to the `reader`. Hence, suitable declarations for the variables appearing in our specification of the `reader` are the following (where `item` is a user-defined type, containing the value 0, that we will consider later).

```
INPUT
  latest: BOOLEAN,
  index: ARRAY BOOLEAN OF BOOLEAN,
  slot:  ARRAY BOOLEAN OF ARRAY BOOLEAN OF item
OUTPUT
  reading: BOOLEAN
LOCAL
  rpair, rindex: BOOLEAN,
  out: item,
  rpc: [0..3]
```

Notice that SAL keywords must be written in uppercase and that arrays are restricted to single dimension, but their elements may be arrays also.

Finally, we must initialize the local and output variables. This can involve guarded commands or, as here, be accomplished by definitions. It is important that `rpc` is initialized to `0`, but the initial values of `out`, `rpair`, and `rindex` are irrelevant (because they are assigned before they are used), so we may as well set them to `0`, `FALSE`, and `FALSE`, respectively. The initial value of `reading` may be important, so we initialize it nondeterministically to either `TRUE` or `FALSE` using a set assignment (the SAL `IN` construct): the SALenv model checker will then explore all choices.

```
INITIALIZATION
  reading IN {X: BOOLEAN | TRUE};
  rpair = FALSE; rindex = FALSE;
  out = 0;
  rpc = 0
```

We can now specify the `writer` module in a similar manner. First we declare and initialize the variables.

```
writer: MODULE =
BEGIN
INPUT
  reading: BOOLEAN,
OUTPUT
  latest: BOOLEAN,
  index: ARRAY BOOLEAN OF BOOLEAN,
  slot:  ARRAY BOOLEAN OF ARRAY BOOLEAN OF item
LOCAL
  wpair, windex: BOOLEAN,
  wpc: [0..4]
INITIALIZATION
  latest IN {X: BOOLEAN | TRUE};
  wpair = FALSE; windex = FALSE;
  wpc = 0;
  (FORALL (i: BOOLEAN): index[i] IN {X: BOOLEAN | TRUE});
  (FORALL (x, y: BOOLEAN): slot[x][y] IN {X: item | TRUE})
```

Observe that the variables that were input variables for the `reader` are outputs for the `writer`, and vice versa. Observe, also, the initialization for the `index` and `slot` arrays: we use `FORALL` expressions to iterate over the array indices, and a nondeterministic set assignment to provide the value.

The guarded commands for the writer are derived straightforwardly from the pseudocode shown earlier. The exception is the assignment to `slot` when `wpc = 2`. Here, we need to simulate the larger environment of the `writer` that produces the values it is required to write. The `([] (val:  item):  ...)` construction nondeterministically selects any value from the type `item` and assigns it to `val`. Notice how this value is assigned to the correct location in `slot` using the `WITH` construct; this is equivalent to the assignment `slot’[wpair][windex] = val` and was chosen simply for variety.

```
TRANSITION
  [
    wpc = 0 --> wpair’ = NOT reading;
                wpc’ = 1
  []
    wpc = 1 --> windex’ = NOT index[wpair];
                wpc’ = 2
  []
   ([] (val: item):
    wpc = 2 --> slot’ = slot WITH [wpair][windex] := val;
                wpc’ = 3)
  []
    wpc = 3 --> index’[wpair] = windex;
                wpc’ = 4
  []
    wpc = 4 --> latest’ = wpair;
                wpc’ = 0
  ]
END;
```

Now that we have the `reader` and the `writer`, we can compose them to yield the full system. This is accomplished by the following specification. This indicates the asynchronous composition of the two modules; by default the inputs of one are connected to the outputs of the other by matching the names of the variables.

```
system: MODULE = reader [] writer;
```

It might now seem that we have specified the algorithm and can proceed to examine it by model checking, but this overlooks a crucial point. Suppose the reader is at its location `rpc = 0` (where it reads the value of `latest`) while the writer is at its location `wpc = 4` (where it assigns to `latest`). We know that the `writer` has not completed its write (because we would then have `wpc = 0`) but can imagine that it is somewhere in the process of performing it. Similarly, the `reader` is somewhere in the process of performing its read. Because `latest` is not assumed to be an atomic register, but merely a typesafe one, its concurrent access by the `reader` and the `writer` can result in the `reader` obtaining *any* value. Thus, this section of the reader should be rewritten as follows.

```
  rpc = 0 AND wpc /= 4 --> rpair’ = latest;
                           rpc’ = 1
[]
 ([] (arb: BOOLEAN):
  rpc = 0 AND wpc = 4 --> rpair’ = arb;
                          rpc’ = 1)
```

This specifies that if `reader` accesses `latest` when the writer is *not* concurrently writing it, then it obtains the value most recently written; but when the accesses are concurrent, the `reader` may obtain any value.

Similar constructions are required for concurrent access to `index` when `rpc = 2` and `wpc = 3` and to `reading` when `rpc = 1` and `wpc = 0`. Since each module now refers to the program counter and some other local variables of the other, these need to be changed from local to input and output variables as appropriate.

2.2 Formal Examination: Mutual Exclusion

Now that we have finally completed specification of the algorithm, we can proceed to examine it by model checking. The first property that we want to examine is what Simpson calls *data coherence*: this is the requirement that the `reader` should not read from a slot that the `writer` is updating concurrently. Data coherence can be interpreted as mutual exclusion on slot access, and we use that term to refer to the property.

The `reader` is in the process of reading its slot when `rpc = 3`, and the `writer` is similarly engaged when `wpc = 2`; the `reader` accesses the slot at position `[rpair][rindex]` while the `writer` uses `[wpair][windex]`, so we need to be sure that whenever `wpc = 2 AND rpc = 3` then either `wpair /= rpair` or `windex /= rindex`. We specify this by the following invariant.

```
mutex: LEMMA system |-
        G((wpc = 2 AND rpc = 3) =>
             (wpair /= rpair OR windex /= rindex));
```

SALenv uses linear temporal logic (LTL) as its assertion language, where the symbol `G` stands for the *always* modality (often written $\Box$). Notice that some of the variables appearing in this formula are local to the modules concerned. Unfortunately, SALenv assertions cannot refer to local variables (because it is difficult to name them uniquely in complex module constructions), so we have to modify the specifications of `reader` and `writer` to make these outputs of their respective modules.

Before we can model check the specification, we need to fix the `item` type. Notice that the `mutex` property does not depend on the values assigned to the `slot` variable, so the exact range of this type is irrelevant to the property. To reduce the state space, we should set `item` to be as small as possible. A suitable definition is the following, which has just the single value `0`.

```
  item: TYPE = [0..0];
```

We then enclose the whole specification in a `CONTEXT` called `fourslot` as follows.

```
fourslot: CONTEXT =
BEGIN
% rest of specification
END
```

The complete specification is then saved in a file called `fourslot.sal` that is shown in Appendix A.

To model check the `mutex` property, we first compile the SAL specification with the following command.

```
sal-model-checker --output=check fourslot mutex
```

Here, `fourslot` is the name of the `CONTEXT` (and, by convention, of the file), and `mutex` is the assertion to be checked. The `--output=check` flag causes the output file to be named `check`. We can then run that file with the following command.

```
./check --verbose
```

This causes SALenv to explore all possible behaviors of the `system` we have specified, and to check the `mutex` property at every reachable state. It produces the following output.

```
Checking...
verified
Number of visited states = 1248
Maximum depth = 574
```

Because the aspects of the algorithm examined in the `mutex` property are accurately represented in the SAL specification (e.g., no "downscaling" was necessary to make it finite state), this output indicates formal verification of the `mutex` property for Simpson's algorithm. Because this property ensures there is no concurrent access to the `slot` variables, these need only have the properties of persistent registers.

2.3 Formal Examination: Freshness

Beyond the data coherence ensured by mutual exclusion, Simpson [Sim97] identified *data freshness* as an important requirement for his asynchronous communication mechanism. The idea is that the `reader` should always return the value written by the most recent activation of the `writer`.

There is a necessary delay between the `writer` writing a value in a slot and it updating the control variables to reflect that fact. Thus, we should require only that the `reader` obtain either the latest value written, or the one prior to that. To specify this, we introduce two new output variables, `current` and `recent`, to the `writer`: `current` saves the value most recently written while `recent` saves the value before that.

```
([] (val: item):
      wpc = 2 --> current’ = recent;
                  slot’ = slot WITH [wpair][windex] := val;
                  recent’ = val;
                  wpc’ = 3)
```

In the `reader`, we can then specify that the value read is equal to either `recent` or `current`—but should it be the values of these variables now or at the point when the `reader` began execution? Both choices seem reasonable, so we introduce local variables `prev_recent` and `prev_current` in the `reader` and use these to latch the values of `recent` and `current`, respectively, as the `reader` starts.

```
  rpc = 0 AND wpc /= 4 --> rpair’ = latest;
                           prev_current’ = current;
                           prev_recent’ = recent;
                           rpc’ = 1
[]
 ([] (arb: BOOLEAN):
  rpc = 0 AND wpc = 4 --> rpair’ = arb;
                          prev_current’ = current;
                          prev_recent’ = recent;
                          rpc’ = 1)
```

Then, we set another local variable `ok` to record whether the value read is equal to one of these saved values.

```
  rpc = 3 --> out’ = slot[rpair][rindex];
              ok’ = out’=recent OR out’=current
                   OR out’=prev_recent OR out’=prev_current;
              rpc’ = 0
```

Our requirement is that `ok` is always `TRUE`.

```
freshness: LEMMA system |- G(ok);
```

We initialize `current` and the other similar variables to `0`, and we also change the initialization of `slot` so that all its locations start with `0`, too. We must also modify the type of `item` so that it has at least one value different from `0`.

```
item: TYPE = [0..1];
```

We compile and model check the modified specification.

```
sal-model-checker --output=check fourslot freshness
./check
```

```
Counter example detected:
Number of visited states = 5242
Maximum depth = 13
----------------
out = 0
ok = TRUE
reading = TRUE     latest = TRUE
rindex = FALSE     rpair = FALSE
wpair = FALSE      windex = FALSE
prev_recent = 0    prev_current = 0
recent = 0         current = 0
wpc = 0            rpc = 0
index = [true := TRUE, false := TRUE]
slot = [true := [true := 0, false := 0], false := [true := 0, false := 0]]
----------------
Transition: <[fourslot:85]>.1
wpc = 1
----------------
Transition: <[fourslot:92]>.1
wpc = 2
----------------
Transition: <[fourslot:96] with [val := 1]>.1
slot = [true := [true := 0, false := 0], false := [true := 0, false := 1]]
current = 1
wpc = 3
----------------
Transition: <[fourslot:102]>.1
index = [true := TRUE, false := FALSE]
wpc = 4
----------------
Transition: <[fourslot:105]>.1
latest = FALSE
wpc = 0
----------------
Transition: <[fourslot:85]>.1
wpc = 1
----------------
Transition: <[fourslot:92]>.1
windex = TRUE
wpc = 2
----------------
Transition: <[fourslot:96] with [val := 1]>.1
slot = [true := [true := 0, false := 0], false := [true := 1, false := 1]]
recent = 1
wpc = 3
----------------
Transition: <[fourslot:102]>.1
index = [true := TRUE, false := TRUE]
wpc = 4
----------------
Transition: <[fourslot:35] with [arb := TRUE]>.0
prev_recent = 1
prev_current = 1
rpc = 1
rpair = TRUE
----------------
Transition: <[fourslot:39]>.0
rpc = 2
----------------
Transition: <[fourslot:42]>.0
rpc = 3
rindex = TRUE
----------------
Transition: <[fourslot:50]>.0
rpc = 0
ok = FALSE
----------------
Transition: <[fourslot:35] with [arb := FALSE]>.0
out = 0
ok = FALSE
reading = TRUE     latest = FALSE
rindex = TRUE      rpair = FALSE
wpair = FALSE      windex = TRUE
prev_recent = 1    prev_current = 1
recent = 1         current = 1
wpc = 4            rpc = 1
index = [true := TRUE, false := TRUE]
slot = [true := [true := 0, false := 0], false := [true := 1, false := 1]]
```

This generates a long counterexample, so we run it again using breadth-first search and compact trace output (the flag `--detailed-trace` causes local variables to be printed in the trace, the `--compact-trace` prints only *differences* from one state to the next).

```
./check --bfs --verbose --detailed-trace --compact-trace
```

This produces a short counterexample, but it is rather difficult to explain. Disabling the nondeterministic choice at `wpc = 0` produces the counterexample shown above (which has been edited, by combining lines, to make it fit on the page).

The behavior exposed by this counterexample is the following. All slots initially contain `0`; the `writer` executes twice and writes `1` into both left-hand (say) slots but, while it is performing the final step of its second execution (where it assigns to `latest`), the `reader` executes twice. Notice that the value being assigned to `latest` is the same value it has already (indicating that the `writer` has written the left-hand slots on both occasions). The first execution of `reader` gets the correct value of `latest` (perhaps because the `writer` has not actually started to write the variable yet), but the second execution gets the wrong value (perhaps because the `writer` is now actually in the middle of its write). This causes it to read from the right-hand slots, which have never been written by the `writer` and still contain their initial values.

Notice that, although it was exposed by model checking a freshness property, this counterexample actually manifests violation of a more important property: that which Simpson calls *data sequencing*. This requires that the `reader` should never return data *older* than that which it has returned previously. This property is more important than freshness because it is in the nature of asynchronous operation that the `reader` and the `writer` operate at different speeds, so the components that receive data from the `reader` must be prepared to tolerate loss of some values (when the `writer` is writing them faster than the `reader` is reading) and repetition of others (when the pace of their relative progress is reversed). What it should not expect, and may be unable to tolerate, is receiving values out of their correct temporal sequence.

The root of the problem manifested by the counterexample is that the control registers used in Simpson's algorithm are not atomic, merely safe. This means that they can return *any* value when read and written concurrently, and this is what causes the algorithm to fail. Notice, however, that the counterexample manifested the problem when the *same* value was written to a control register as it had already. Surely such a write is unnecessary, and if it were not performed, this particular problem could not arise: there would be no concurrent read and write (since there would be no write), and the reader would get the correct value (since the control register already had the value required). But what if the control register does *not* already contain the desired value? The write must then take place and surely this provokes the same problem of concurrent reads and writes to a nonatomic register. The vital insight that vitiates this concern is that if a safe *Boolean* register is written only when the new value is *different* from the old, then it behaves as an atomic register.[1] The value before

[1]We will later see that this insight is incorrect.

the write may be TRUE (say), so that afterward it will be FALSE; a concurrent read can produce any value, but there are only two choices for a Boolean register—hence, the value obtained must be consistent with the read occurring either atomically before the write, or atomically afterward.

Simpson possessed this insight and states [Sim90, Page 20, Section 2.3]

> “*The co-operative access control technique is only sound if we can rely completely on the integrity of the control variables.*
>
> ...
>
> “*Fortunately, there is a variable type whose value is always guaranteed to be coherent, and where reading and writing operations can be concurrent and need not interfere with each other; this is the bit variable.*
>
> ...
>
> “*The integrity of any fully asynchronous mechanisms rests ultimately on this property of the bit variable. Practical implementations of such variables, when used for control within an asynchronous mechanism, must ensure the following:*
>
> ...
>
> “*(b) A write operation which does not change the value must not induce any disturbance which might cause a concurrent read to obtain an incorrect value.*”

The simplest way to ensure the absence of “any disturbance” with a “write operation which does not change the value” is to eschew such writes: simply modify the algorithm so that Boolean control variables are written only if they change the value of the register. Our hope is that this modification corrects the problem identified by the counterexample. The modified algorithm is shown below.

```
item: TYPE                                                          3

slot: ARRAY [boolean, boolean] OF item;
index: ARRAY [boolean] OF boolean;
latest, reading: boolean;

PROCESS writer(in: item);
  wpair, windex: boolean;
BEGIN
  wpair := !reading;
  windex := !index[wpair];
  slot[wpair][windex] := in;
  index[wpair] := windex;
  IF latest /= wpair THEN latest := wpair ENDIF;
END;

PROCESS reader(out: item);
  rpair, rindex: boolean;
BEGIN
  rpair := latest;
  IF reading /= rpair THEN reading := rpair ENDIF;
  rindex := index[rpair];
  out := slot[rpair][rindex];
END;
```

Notice that the assignment `index[wpair] := windex` in the fourth line of the `writer` does not require the `IF...THEN ...` protection because these variables are written only by the `writer`, and `windex` is earlier assigned to the negation of `index[wpair]`. We examine this modified algorithm in the following section.

2.4 Formal Examination: The Modified Specification

The SAL specification can be adjusted to correspond to the corrected presentation of the algorithm given in [3]. For example, the `TRANSITION` for the `reader` is shown below (assignments to the verification variables such as `recent`, `prev_current` have been omitted). The full specification is given in Appendix B.

```
TRANSITION
  [
    rpc = 0 AND wpc /= 4 -->
                rpair' = latest;
                rpc' = IF reading = rpair' THEN 2 ELSE 1 ENDIF
  []
   ([] (arb: BOOLEAN):
    rpc = 0 AND wpc = 4 -->
                rpair' = arb;
                rpc' = IF reading = rpair' THEN 2 ELSE 1 ENDIF)
  []
    rpc = 1 --> reading' = rpair;
                rpc' = 2
  []
    rpc = 2 AND (wpc /= 3 OR rpair /= wpair) -->
                rindex' = index[rpair];
                rpc' = 3
  []
   ([] (arb: BOOLEAN):
    rpc = 2 AND wpc = 3 AND rpair = wpair -->
                rindex' = arb;
                rpc' = 3)
  []
    rpc = 3 --> out' = slot[rpair][rindex];
                rpc' = 0
  ]
END;
```

If we model check the revised specification with SALenv, it again reports failure. Examination of the counterexample reveals that it indicates a weakness in our specification of freshness, rather than a true flaw in the algorithm. The problem is that a single activation of the `reader` can be concurrent with multiple activations of the `writer`, and the interleaving can be such that the value obtained by the `reader` is one that was written in the middle of the series of writes—whereas our verification variables (as used in the test `ok`) capture only those written at the beginning or end. Since there can be any number of writes overlapping a read, it looks as if we need an unbounded number of verification variables to capture these legitimate outputs by the `reader`. We could bound the number by limiting the number of overlapping writes, or by putting a bound on the model checker search depth, but these solutions seem ad hoc. Furthermore, freshness is merely a desirable property; as explained earlier, the property that is really essential is data sequencing.

A very good way to examine this property is to cause the `writer` to write consecutive integers and then require that the reader return nondecreasing values.[2] To specify this, we

[2]This idea is due to N. Shankar.

first increase the range of the type `item` so that it will permit an interesting number of different values to be written.

```
item: TYPE = [0..8];
```

Then we modify the selection of values written by the `writer` so that they become strictly increasing. This can be done as follows (where the local variable `prev` is used to remember the previous value and is initialized to `0`).

```
 ([] (val: item):
    wpc = 2 AND val > prev -->
           slot' = slot WITH [wpair][windex] := val;
           prev' = val;
           wpc' = 3)
```

Finally, we specify that the values obtained by the `reader` must be nondecreasing.

```
atomic: LEMMA system |- G(out' >= out);
```

The expression specifies that the value of `out` after the assignments in any command must be no less than its value before. This property is a manifestation of that called "atomicity" in the computer science literature, hence the name used in the specification. (Atomicity is the property that any sequence of concurrent reads and writes behaves as if each had been performed indivisibly, and all had been performed in some definite sequence.)

The state space of this model is larger than previously, so we turn on some optimizations.

```
sal-model-checker --output=check --optimize fourslotmodified atomic
./check --bfs --verbose --detailed-trace --compact-trace
```

To our astonishment, this exercise returns a counterexample.

The essence of the counterexample is as follows. The `writer` starts writing in (say) the top left-hand slot. The `reader` starts while the `writer` is at `wpc = 3` (i.e., in the process of changing `index[rpair]` so that the left index indicates the top slot rather than the bottom). Because this is merely a safe register, the `reader` may obtain any value for `index[rpair]` (at `rpc = 2`), and it obtains the new value (i.e., top) and so returns the value most recently written by the `writer`. Immediately following the completion of this activation of the `reader`, a new one begins (the `writer` remains at `wpc = 3`). At `rpc = 2`, this activation obtains the old value for `index[rpair]` and so returns the value from the slot *previous* to the one most recently written, thereby violating the data sequencing requirement.

Whether this is a real bug or not depends on the assumptions from which one approaches the problem. From the computer science perspective, in which the problem is formulated as that of constructing an atomic register from merely safe registers, it is a genuine bug. The source of the bug is the observation on Page 14:

"The vital insight that vitiates this concern is that if a safe Boolean *register is written only when the new value is* different *from the old, then it behaves as an atomic register.*

This insight is incorrect: it is true that a concurrent read of a safe Boolean register that is written only when its new value is different from its old must always return the old or the new value, but a *second* concurrent read may produce the *old* value when the first produced the *new* value. This is not the behavior of an atomic register—it is the behavior of merely a *regular* register.

Simpson's algorithm depends on atomicity of its Boolean control registers. As we have seen, the construction suggested by Simpson (writing the control variables only when their new value is different from their old value) is inadequate to achieve this requirement, given only safe control registers. However, Simpson does not claim that his algorithm assumes only that the control registers are safe. Rather, he assumes them to be atomic and his discussion in [Sim90, Page 20, Section 2.3] (from which comes the construction that control variables should be written only when their new value is different from their old value) can be seen as suggestions to an implementor, rather than a complete algorithm.

Thus, to be accurate to Simpson's intent, we should examine correctness of his algorithm when the control registers are assumed to be atomic. This is undertaken in the following section.

2.5 Formal Examination: Assuming Atomic Control Registers

If atomic control registers are assumed, we could revert to the original specification $\boxed{2}$ from Page 6. It seems useful, however, to include at least the detail that the control registers are written only when their value will be changed. The `TRANSITION` for the `reader` then appears as follows. Notice that we have eliminated all the `([] (arb: BOOLEAN): ...)` constructions, since these are needed only to model safe registers. The full specification is given in Appendix C.

```
TRANSITION
  [
    rpc = 0 --> rpair’ = latest;
                rpc’ = 1
  []
    rpc = 1 AND reading /= rpair --> reading’ = rpair;
                rpc’ = 2
  []
    rpc = 1 AND reading = rpair -->
                rpc’ = 2
  []
    rpc = 2 --> rindex’ = index[rpair];
                rpc’ = 3
  []
    rpc = 3 --> out’ = slot[rpair][rindex];
                rpc’ = 0
  ]
END;
```

We can model check the `atomic` property for this specification with the following commands (the Unix `time` command measures the CPU time used).

```
sal-model-checker --output=check --optimize fourslotatomic atomic
time ./check --bfs --verbose
```

The following results are produced.

```
Checking.............................................verified
Number of visited states = 427378
Maximum depth = 42
87.770u 0.410s 1:28.91 99.1%    0+0k 0+0io 605pf+0w
```

The timing data indicates that model checking used just over 87 seconds (on an 800 MHz Pentium III running Red Hat Linux V7.1) to examine 427,378 states. The atomicity property is verified for this version of the specification.

Chapter 3

Conclusion

We have examined Simpson's four-slot fully asynchronous communication mechanism [Sim90] by model checking with SALenv. Simpson's mechanism can be seen as an algorithm for constructing a lock-free and wait-free atomic register from registers satisfying weaker assumptions. Atomic registers are crucial for reliable communication of data between separately clocked components.

We showed that Simpson's algorithm ensures mutual exclusion (i.e., the reader and writer never access the same data slot concurrently), and hence coherence or integrity of the data, when its registers satisfy the weakest of all assumptions: those of merely safe registers.

However, we found that the algorithm ensures atomicity (i.e., the property that the values read are as if reads and writes do not overlap), and hence correct data sequencing, only when its control registers are already atomic. We also found that modifying the algorithm so that the Boolean control registers are written only when their values will be changed is inadequate to overcome this limitation.

It seems unsatisfactory to base a construction for atomic registers on the assumption that its control registers are already atomic. It may be that this assumption can be discharged with careful electrical design (particularly for the modified algorithm, where the circumstances in which it fails may be considered to require rare electrical events). Nonetheless, it would seem preferable to use a construction that truly requires merely safe control registers. The algorithm of Anderson and Gouda [AG92] may satisfy this requirement.

Our formal examination used model checking (actually, explicit-state reachability analysis). For the mutual exclusion property, our modeling was exact and therefore provides true verification of this property. For atomicity, our modeling is indirect. We identify atomicity with the requirement that writes of strictly increasing integer values should lead to reads whose values are nondecreasing. This requirement is certainly necessary for atomicity, but may not be sufficient. However, Hesselink [Hes02], identifies a criterion for atomicity that explicitly associates sequence numbers with values written and read; we can regard the integers used in our experiments as being their own sequence numbers. Thus, although we lack a proof, it seems very plausible that our requirement is sufficient as well as necessary. Our model checking is compromised in that we consider sequences of no more than eight writes, but intuitively this seems sufficient to explore all behaviors (the counterexamples that exposed flawed assumptions were all found using only two values). Thus, we consider our model checking of the atomicity property to provide very strong, but not conclusive, evidence for correctness of Simpson's algorithm with atomic control registers.

Clark [Cla00] has examined several atomic register constructions, including Simpson's, using Petri nets. His Petri net models are much further removed from the natural presentation of the algorithms than our SALenv specifications, and the required properties are also stated indirectly. Clark's treatment of data freshness and sequencing derive from the "role models" of Simpson [Sim97], which are themselves abstracted descriptions that are not fully independent of the algorithms under examination. Clark assumes atomic control variables and verifies the properties of data coherence (mutual exclusion), freshness, and sequencing. He also considers a certain model of metastability that can be seen as weakening the assumption of atomicity on the control variables, and shows that Simpson's algorithm fails in this case. Clark's work (which was not known to the author when this work was performed) is similar to that reported here, but our examination of atomicity assumptions on the control registers is more comprehensive, and we also consider our modeling in SAL to be much more direct and accessible than that using Petri nets.

Henderson and Paynter [HP02] have formally verified the mutual exclusion property of Simpson's algorithm using theorem proving in PVS, but they have not yet examined the atomicity property. Hesselink [Hes02] has formally verified some multi-reader atomic register constructions (that build on single-reader constructions of the kind considered here) using theorem proving in ACL2. Both theorem proving approaches are substantially more complex than ours. As noted above, model checking is exact for the mutual exclusion property and it is plausibly so for the atomicity property, so theorem proving provides additional benefit only for the latter property, and at rather large cost. It would be interesting to see if theorem proving can be used to justify the approximation used in model checking the atomicity property; if it can, this combination of theorem proving and model checking might provide the most cost-effective approach to formal verification for this class of algorithms.

We have seen that Simpson's algorithm can fail if its control registers are not atomic. Although it includes suggestions for realizing atomic Boolean control registers, Simpson's paper does not provide an algorithm for realizing them, and the suggestion that the control registers are written only when their value will be changed reduces but does not eliminate the failure mode. Unless a verifiable construction for realizing Boolean control registers is employed, Simpson's algorithm cannot be considered verifiably correct. It requires expertise in the application domain (to determine the consequences of possible loss of atomicity), in circuit design (to determine the plausibility of the circumstances that lead to its failure), and in safety analysis to decide whether the algorithm is fit for deployment in a particular safety-critical context.

Because Simpson's algorithm is not unconditionally correct, it seems desirable to examine other atomic register constructions that share its good features (it is short and uses few control registers) and may remove its limitations. The algorithm of Anderson and Gouda [AG92] seems promising and it would be interesting to subject it to examination similar to that performed here. Clark's analysis [Cla00] found no flaws in this algorithm, but our examination of the version given by Clark [Cla00, Figure 5.19] finds that it fails even the `mutex` property when the control variables are merely safe.

Bibliography

[AG92] J. H. Anderson and M. G. Gouda. A criteron for atomicity. *Formal Aspects of Computing*, 4:273–298, 1992. 21, 23

[And01] James H. Anderson. Lamport on mutual exclusion: 27 years of planting seeds. In *20th ACM Symposium on Principles of Distributed Computing*, pages 3–12, Newport, RI, August 2001. Association for Computing Machinery. 2

[BGL$^+$00] Saddek Bensalem, Vijay Ganesh, Yassine Lakhnech, César Muñoz, Sam Owre, Harald Rueß, John Rushby, Vlad Rusu, Hassen Saïdi, N. Shankar, Eli Singerman, and Ashish Tiwari. An overview of SAL. In C. Michael Holloway, editor, *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, pages 187–196, Hampton, VA, June 2000. NASA Langley Research Center. Proceedings available at `http://shemesh.larc.nasa.gov/fm/Lfm2000/Proc/`. 5

[BP87] James E. Burns and Gary L. Peterson. Constructing multi-reader atomic values from non-atomic values. In *6th ACM Symposium on Principles of Distributed Computing*, pages 222–231, Vancouver, British Columbia, Canada, August 1987. Association for Computing Machinery. 2

[CB97] Jing Chen and Alan Burns. A three-slot asynchronous reader-writer mechanism for multiprocessor real-time systems. Technical Report YCS-286, Department of Computer Science, University of York, May 1997. 4

[Cla00] Ian G. Clark. *A Unified Approach to the Study of Asynchronous Communication Mechanisms in Real Time Systems*. PhD thesis, King's College, London University, May 2000. 2, 22, 23

[DHOS01] David Dill, Thomas Henzinger, Sam Owre, and N. Shankar. The SAL language. Technical Report SRI-CSL-01-02, Computer Science Laboratory, SRI International, Menlo Park, CA, March 2001. 5

[Hes02] Wim H. Hesselink. An assertional criterion for atomicity. *Acta Informatica*, 28(5):343–366, 2002. 21, 22

[HP02] N. Henderson and S. E. Paynter. The formal classification and verification of Simpson's 4-slot asynchronous communication mechanism. In Peter Lindsay, editor, *Formal Methods Europe (FME'02)*, Lecture Notes in Computer Science, Copenhagen, Denmark, July 2002. Springer-Verlag. To appear. 2, 3, 22

[HV92] S. Haldar and K. Vidyasankar. Counterexamples to a one writer multireader atomic variable construction of Burns and Peterson. *ACM Operating Systems Review*, 26(1):78–87, January 1992. 2

[KG94] Hermann Kopetz and Günter Grünsteidl. TTP—a protocol for fault-tolerant real-time systems. *IEEE Computer*, 27(1):14–23, January 1994. 2

[Kop99] Hermann Kopetz. Elementary versus composite interfaces in distributed real-time systems. In *The Fourth International Symposium on Autonomous Decentralized Systems*, Tokyo, Japan, March 1999. IEEE Computer Society. 1

[KR93] Hermann Kopetz and Johannes Reisinger. The non-blocking write protocol NBW: A solution to a real-time synchronization problem. In *Real Time Systems Symposium*, pages 131–137, Raleigh-Durham, NC, December 1993. IEEE Computer Society. 2

[Lam77] Leslie Lamport. Concurrent reading and writing. *Association for Computing Machinery*, 20(11):806–811, November 1977. 1, 2

[Lam86] Leslie Lamport. On interprocess communication—Part I: Basic formalism, Part II: Algorithms. *Distributed Computing*, 1(2):77–101, 1986. 2, 3

[Pet87] Gary Peterson. Concurrent reading while writing. *ACM Transactions on Programming Languages and Systems*, 5(1):46–55, 1987. 2

[Rus02] John Rushby. An overview of formal verification for the time-triggered architecture. In Werner Damm and Ernst-Rüdiger Olderog, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Lecture Notes in Computer Science, Oldenburg, Germany, November 2002. Springer-Verlag. To appear. 1

[Sim90] H. R. Simpson. Four-slot fully asynchronous communication mechanism. *IEE Proceedings, Part E: Computers and Digital Techniques*, 137(1):17–30, January 1990. 2, 4, 15, 19, 21

[Sim97] H. R. Simpson. Role model analysis of an asynchronous communication mechanism. *IEE Proceedings, Part E: Computers and Digital Techniques*, 144(4):232–240, July 1997. 11, 22

[Tro89] John Tromp. How to construct an atomic variable. In *Distributed Algorithms (3rd International Workshop WDAG '89)*, volume 392 of *Lecture Notes in Computer Science*, pages 292–302, Nice, France, September 1989. Springer-Verlag. 2

[TTT01] Time-Triggered Technology TTTech Computertechnik AG, Vienna, Austria. *Specification of the TTP/C Protocol (version 0.6p0504)*, May 2001. 2

Appendix A

Simpson's Algorithm in SAL

The following presents Simpson's algorithm, modeled in the SAL specification language. The modeling of control variables reproduces the behavior of safe registers. The `mutex` property is verified for this specification, but the `freshness` property fails.

```
fourslot: CONTEXT =
BEGIN
item: TYPE = [0..0];

reader: MODULE =
BEGIN
INPUT
  latest: BOOLEAN,
  index: ARRAY BOOLEAN OF BOOLEAN,
  slot:  ARRAY BOOLEAN OF ARRAY BOOLEAN OF item,
  recent, current: BOOLEAN,
  wpair: BOOLEAN,
  wpc: [0..4]
OUTPUT
  reading: BOOLEAN,
  rpair, rindex: BOOLEAN,
  ok: BOOLEAN,
  rpc: [0..3]
LOCAL
  prev_recent, prev_current: item,
  out: item
INITIALIZATION
  reading IN X: BOOLEAN | TRUE;
  rpair = FALSE; rindex = FALSE;
  ok = TRUE;
  prev_recent = 0; prev_current = 0;
  out = 0;
  rpc =0
```

```
TRANSITION
  [
    rpc = 0 AND wpc /= 4 --> rpair’ = latest;
                prev_current’ = current; prev_recent’ = recent;
                rpc’ = 1
  []
    ([] (arb: BOOLEAN):
    rpc = 0 AND wpc = 4 --> rpair’ = arb;
                prev_current’ = current; prev_recent’ = recent;
                rpc’ = 1)
  []
    rpc = 1 --> reading’ = rpair;
                rpc’ = 2
  []
    rpc = 2 AND (wpc /= 3 OR rpair /= wpair) -->
                rindex’ = index[rpair];
                rpc’ = 3
  []
    ([] (arb: BOOLEAN):
    rpc = 2 AND wpc = 3 AND rpair = wpair --> rindex’ = arb;
                prev_current’ = current; prev_recent’ = recent;
                rpc’ = 3)
  []
    rpc = 3 --> out’ = slot[rpair][rindex];
                ok’ = out’=recent OR out’=current
                       OR out’=prev_recent OR out’=prev_current;
                rpc’ = 0
  ]
END;

writer: MODULE =
BEGIN
INPUT
  reading: BOOLEAN,
  rpc: [0..3]
OUTPUT
  latest: BOOLEAN,
  index: ARRAY BOOLEAN OF BOOLEAN,
  slot:  ARRAY BOOLEAN OF ARRAY BOOLEAN OF item,
  wpair, windex: BOOLEAN,
  recent, current: item,
  wpc: [0..4]
INITIALIZATION
    latest IN X: BOOLEAN | TRUE;
    wpair = FALSE; windex = FALSE;
    wpc = 0;
    (FORALL (i: BOOLEAN): index[i] IN X: BOOLEAN | TRUE);
    recent = 0; current = 0;
    (FORALL (x, y: BOOLEAN): slot[x][y] = 0)
TRANSITION
  [
    wpc = 0 AND rpc /= 1 --> wpair’ = NOT reading;
                wpc’ = 1
  []
  ([](arb:BOOLEAN):
    wpc = 0 AND rpc = 1 --> wpair’ = NOT reading;
                wpc’ = 1)
  []
    wpc = 1 --> windex’ = NOT index[wpair];
                wpc’ = 2
  []
    ([] (val: item):
    wpc = 2 --> recent’ = current;
                slot’ = slot WITH [wpair][windex] := val;
                current’ = val;
                wpc’ = 3)
  []
    wpc = 3 --> index’[wpair] = windex;
                wpc’ = 4
  []
    wpc = 4 --> latest’ = wpair;
                wpc’ = 0
  ]
END;

system: MODULE = reader [] writer;

mutex: LEMMA system |-
        G((wpc = 2 AND rpc = 3) => (wpair /= rpair OR windex /= rindex));

freshness: LEMMA system |- G(ok);

END
```

Appendix B

Modified Simpson's Algorithm in SAL

The following presents Simpson's algorithm, modified so that control registers are written only when the write will change the stored value. Control variables are modeled as safe registers. The `atomic` property fails for this specification.

```
fourslot: CONTEXT =
BEGIN
item: TYPE = [0..1];

reader: MODULE =
BEGIN
INPUT
  latest: BOOLEAN,
  index: ARRAY BOOLEAN OF BOOLEAN,
  slot:  ARRAY BOOLEAN OF ARRAY BOOLEAN OF item,
  wpair: BOOLEAN,
  wpc: [0..4]
OUTPUT
  reading: BOOLEAN,
  rpair, rindex: BOOLEAN,
  rpc: [0..3],
  out: item
INITIALIZATION
  reading IN X: BOOLEAN | TRUE;
  rpair = FALSE; rindex = FALSE;
  out = 0;
  rpc =0
TRANSITION
  [
    rpc = 0 AND wpc /= 4 --> rpair' = latest;
```

```
                rpc’ = IF reading = rpair’ THEN 2 ELSE 1 ENDIF
  []
    ([] (arb: BOOLEAN):
    rpc = 0 AND wpc = 4 --> rpair’ = arb;
                rpc’ = IF reading = rpair’ THEN 2 ELSE 1 ENDIF)
  []
    rpc = 1 --> reading’ = rpair;
                rpc’ = 2
  []
    rpc = 2 AND (wpc /= 3 OR rpair /= wpair) --
> rindex’ = index[rpair];
                rpc’ = 3
  []
    ([] (arb: BOOLEAN):
    rpc = 2 AND wpc = 3 AND rpair = wpair --> rindex’ = arb;
                rpc’ = 3)
  []
    rpc = 3 --> out’ = slot[rpair][rindex];
                rpc’ = 0
  ]
END;

writer: MODULE =
BEGIN
INPUT
  reading: BOOLEAN,
  rpc: [0..3]
OUTPUT
  latest: BOOLEAN,
  index: ARRAY BOOLEAN OF BOOLEAN,
  slot:  ARRAY BOOLEAN OF ARRAY BOOLEAN OF item,
  wpair, windex: BOOLEAN,
  wpc: [0..4]
LOCAL
  prev: item
INITIALIZATION
    latest IN X: BOOLEAN | TRUE;
    wpair = FALSE; windex = FALSE;
    prev = 0;
    wpc = 0;
    (FORALL (i: BOOLEAN): index[i] IN X: BOOLEAN | TRUE);
    (FORALL (x, y: BOOLEAN): slot[x][y] = 0)
TRANSITION
  [
    wpc = 0 AND rpc /= 1 --> wpair’ = NOT reading;
                wpc’ = 1
  []
  ([](arb:BOOLEAN):
    wpc = 0 AND rpc = 1 --> wpair' = arb;
                wpc' = 1)
  []
    wpc = 1 --> windex' = NOT index[wpair];
                wpc' = 2
  []
    ([] (val: item):
    wpc = 2 AND val > prev --> slot' = slot WITH [wpair][windex] := val;
                prev' = val;
                wpc' = 3)
  []
    wpc = 3 --> index'[wpair] = windex;
                wpc' = IF latest = wpair THEN 0 ELSE 4 ENDIF
  []
    wpc = 4 --> latest' = wpair;
                wpc' = 0
  ]
END;

system: MODULE = reader [] writer;

mutex: LEMMA system |-
        G((wpc = 2 AND rpc = 3) => (wpair /= rpair OR windex /= rindex));

atomic: LEMMA system |- G(out' >= out);

END
```

Appendix C

Simpson's Algorithm with Atomic Control Registers in SAL

The following presents Simpson's algorithm, modified so that control registers are written only when the write will change the stored value. Control variables are modeled as atomic registers. The `atomic` property is verified for this specification.

```
fourslot: CONTEXT =
BEGIN
item: TYPE = [0..8];

reader: MODULE =
BEGIN
INPUT
  latest: BOOLEAN,
  index: ARRAY BOOLEAN OF BOOLEAN,
  slot:  ARRAY BOOLEAN OF ARRAY BOOLEAN OF item,
  wpair: BOOLEAN,
  wpc: [0..4]
OUTPUT
  reading: BOOLEAN,
  rpair, rindex: BOOLEAN,
  rpc: [0..3],
  out: item
INITIALIZATION
  reading IN X: BOOLEAN | TRUE;
  rpair = FALSE; rindex = FALSE;
  out = 0;
  rpc =0
TRANSITION
  [
    rpc = 0 --> rpair' = latest;
```

```
                rpc’ = 1
  []
    rpc = 1 AND reading /= rpair --> reading’ = rpair;
                rpc’ = 2
  []
    rpc = 1 AND reading = rpair -->
                rpc’ = 2
  []
    rpc = 2 --> rindex’ = index[rpair];
                rpc’ = 3
  []
    rpc = 3 --> out’ = slot[rpair][rindex];
                rpc’ = 0
  ]
END;

writer: MODULE =
BEGIN
INPUT
  reading: BOOLEAN,
  rpc: [0..3]
OUTPUT
  latest: BOOLEAN,
  index: ARRAY BOOLEAN OF BOOLEAN,
  slot:  ARRAY BOOLEAN OF ARRAY BOOLEAN OF item,
  wpair, windex: BOOLEAN,
  wpc: [0..4]
LOCAL
  prev: item
INITIALIZATION
    latest IN X: BOOLEAN | TRUE;
    wpair = FALSE; windex = FALSE;
    prev = 0;
    wpc = 0;
    (FORALL (i: BOOLEAN): index[i] IN X: BOOLEAN | TRUE);
    (FORALL (x, y: BOOLEAN): slot[x][y] = 0)
TRANSITION
  [
    wpc = 0 --> wpair’ = NOT reading;
                wpc’ = 1
  []
    wpc = 1 --> windex’ = NOT index[wpair];
                wpc’ = 2
  []
    ([] (val: item):
    wpc = 2 AND val>prev --> slot’ = slot WITH [wpair][windex] := val;
```

```
                prev’ = val;
                wpc’ = 3)
  []
    wpc = 3 --> index’[wpair] = windex;
                wpc’ = 4
  []
    wpc = 4 AND latest /= wpair --> latest’ = wpair;
                wpc’ = 0
  []
    wpc = 4 AND latest = wpair -->
                wpc’ = 0
  ]
END;

system: MODULE = reader [] writer;

mutex: LEMMA system |-
        G((wpc = 2 AND rpc = 3) => (wpair /= rpair OR windex /= rindex));

atomic: LEMMA system |- G(out’ >= out);

END
```